A survey on systematic construction of benchmarks for Static Analysis tools

Supervisor: Michael Eichberg
eichberg@cs.tu-darmstadt.de

Kasra Qasempour Soleimany
TU DARMSTADT
k.qasempour@gmail.com

Vivin Joseph Christoph
TU DARMSTADT
vivinjc@gmail.com

Abstract

Benchmarks have been used in computer science to compare the performance of computer systems. They provide an experimental basis for evaluating software engineering processes or techniques in an objective and repeatable manner. Static Analysis Tools (SAT) have been become popular recently, and whenever a new tool is developed, it comes with a benchmark to show how the tool performs in comparison to other options. However, there is not unified and unique approach for evaluating these tools. In this paper, we looked over some approaches on how to systematically build a benchmark for SATs. We explained different benchmarking approaches and the characteristics of a good benchmarking approach. At the end, we summarize the paper by pointing to differences and similarities of these approaches.

1. Introduction

Static Analysis Tools (SAT) have been become popular recently. The British Computer Society 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 artifacts which builds from source code, like binary code. 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 2011b).

Benchmarks provide an experimental basis for evaluating software engineering theories, represented by software engineering techniques, in an objective and repeatable manner (Tichy 1998). Based on IEEE, a benchmark is defined as "a procedure, problem, or test that can be used to compare systems or components to each other or to a standard" (IEEE 1990). A benchmark represents a problem and possible solutions to it, by defining the motivating comparison, a task sample and evaluation measures (Sim et al. 2003). The task sample can contain programs, tests, and other artifacts dependent on the benchmarks motivating comparison. A benchmark controls the task sample reducing result variability, increasing repeatability, and providing a basis for comparison (Sim et al. 2003).

In this paper we study recent studies on how to systematically create and build benchmarks for static analysis tools and we try to express commonalities, differences, shortcomings and advantages. In the rest of this paper, in Section 2 we provide the foundation for benchmarking criterion and what are the characteristics of a good SAT and a benchmark. Next, in Section 3 we study **Scanstud: A Methodology for Systematic, Fine-grained Evaluation of Static Analysis Tools** (Johns and Jodeit 2011b), then we study **DeltaBench: Differential Benchmark for Static Analysis Security Testing Tools** (Pashchenko et al. 2017). Next we see how MuBenchPipe works which proposes **A Systematic Evaluation of Static API-Misuse Detectors** (Amann et al. 2018). Then in Section 6 we talk about **Hermes: Assessment and Creation of Effective Test Corpora** (Reif et al. 2017). Afterwards, in Section 7 we look at **FOSS Version Differentiation as a Benchmark for Static Analysis Security Testing Tools** (Pashchenko 2017) and finally we summerize in Section 8.

2. Foundations

In this section we define some basic concepts and definitions that we will use in the rest of paper.

2.1 Success Criteria for SATs

Before we dive in to discussing different approaches, we take look over some criteria that a good SAT should meet. They were introduced by Chess and West (Chess and West 2007):

- **C1 - Quality of the analysis** which indicates 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 tradeoff between analysis duration and unwanted results, or memory usage vs depth of data flow to capture.
- **C3 - Vulnerability types coverage** Different types of vulnerabilities is the tool aware of.
- **C4 - Usability of the tool** How easy the tool can be used.

Chess and West introduced these criteria for Static Analysis Security testing tools, so we can generalize the C3 to *"Issue types coverage"*. Except C4, other criteria should be considered in the design of a benchmark.

2.2 Benchmarking approaches

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

- **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 therefore needs more effort in comparison to test on 1 project that has all the issues in one place.
- **Educational applications** There are many testing application for educational purposes which contain 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 in a context. Moreover they are well documented and with regard to size they are small but rich in issues, so the test time is less than the one for 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.

2.3 Characteristics of a good benchmark approach

In order to compare the upcoming systematic benchmark studies we use some characteristics which are well defined by Johns and Jodeit, and vital for any systematic benchmark approach (Johns and Jodeit 2011b):

- **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.

Beside these 3 characteristics, we point to 8 more properties for a successful benchmark, the upcoming properties are described in (Sim et al. 2003):

- **Accessibility** A benchmark should be easy to obtain and use. The test materials and results need to be publicly available. Anyone should be able to apply the benchmark to a tool or techniques and compare their results with others. The test and results should be understandable with software engineers at any level of expertise.
- **Affordability** There should be a trade off between cost and result of benchmark. The cost contains human resource, software and hardware.
- **Clarity** Benchmark's specification should be clear and concise, in order to easily adopt and extend it.
- **Relevance/Representativeness** The collection and the goal of tasks in the benchmark must be selected

in a way that system might behave in real world sample. This means they should not be meaningful in just the benchmark settings. Also the performance measures must be relevant to comparison being made. This is not an easy property to satisfy and also important. Because there is a need to make a decision about the amount of context to include, that makes the problem realistic.

- **Solvability** Completing the task sample and obtaining correct metrics should be possible.
- **Portability** A benchmark should be usable by different techniques without bias. This means it should specified with enough level of abstractions. As an example, the benchmark may need to implemented more than once for different platforms or architectures.
- **Scalability** It should be scalable to any technique with any level of maturity. Adding new techniques and adding other targets should be possible. For example it should support commercial products beside less mature products like a research prototype.
- **Fairness** A benchmark should not bias towards a specific technique. For example supporting only specific programming language (Lu et al. 2005).

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 tests 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 had and how they solved them.

One of the important parts in automating a benchmark is evaluating the results. The main challenge is how to map the tools findings with mined issues in tests. Using line number to address the issues is not enough, because different tools propose different line numbers for a certain issue. As an example, take a look at the Listing 1, the code has an off-bounds writing operation issue. But which line should be reported, line 2 which issue stems from or line 3 which executes the issue. Each of the decision is reasonable so each tool may decide to report line 2 or 3. . They addressed 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 differentiate the *testcode* and *host program*. Host program is the testcode added by some other codes to make it parse-able, compile-able and execute-able. In their test design they have defined to terms, *test* and *testcase*. The testcase is the smallest unit, it is designed to capture only one issue, or to be one false positive. It is passed if the issue

Listing 1: Off-by-one vulnerability caused by insecure loop condition (Johns and Jodeit 2011b)

```
1 [...]
2 for (i = 0; src[i] && (i <= sizeof(dst)); i++) {
3         dst[i] = src[i];
4 }
5 [...]
```

found by the tool, or if it is ignored in case of being a false positive. On the other hand, a test contains one or more testcases, and it is passed if all the testcases passed. A test could be used to verify that if a tool can capture a series of issue semantically related. A test could be used to double check a capability of a tool, by defining a vulnerable testcase and a false positive testcase. This way if the test passed, it shows that the tool is working perfectly in finding the issue. The ability to define different testcases and test enables ones to check C1 and C3 criteria mentioned in 2.1 which supports CH1.

The host program might produce some false positives. To address this issue they create two versions 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 tests, they have a list of template issues saved in files, for example opening a socket. Then there is a script that could insert each template in code, and automatically create different version of tests. Moreover, they have created scripts for automatic tool execution, result diffing, and result evaluation. This greatly supports CH3.

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 and provides means to interact with the testcode. This way test writer can activate the crafted vulnerability to make sure that the written code is insecure. For instance for Java test suite, the test application is indeed a full J2EE application, which use the codes in form of servlet, and then the host program provides a web UI which enables the test designer to access the vulnerable code. This supports CH1.

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 many false positives(Pashchenko et al. 2017). Ground truth means a set of known issues in a context that a tool must capture them and false

positives happens when a tool incorrectly reports that a static analysis rule is violated. The paper provides a *synthetic benchmark* (Micro benchmarking) 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 - Supports CH1-. 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 benchmarking context, and 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 selected 4 tools and they set up different experiments in order to evaluate many aspects of these tools. Some of these projects accept source code as input and others accept binary code, therefore for part of constructing their benchmark, they made binary out of the source codes available in MuBench dataset.

MuBenchPipe automates many parts of their experiments:

- **checkout**: each test item in MuBench dataset has a recorded Id for the respective commit of the project on version control system. It supports SVN and Git repositories, source archives (zip), also hand-crafted examples that come with MuBench.
- **Compile** For some of the detectors, we need binary as input, therefore MuBenchPipe, is able to build the code by getting all the dependencies and base on the configuration found on MuBench dataset.
- **Validation** MuBenchPipe automatically release the result to a review website for manual validation. For every detector finding, the website shows the source code it is found in along with any metadata the detector provides, such as the violated pattern, the properties of the violation, and the detectors confidence(Amann et al. 2018).

It provides a command line interface to control every part, 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 authenti-

cation. 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. Hermes

The Hermes(Reif et al. 2017) is a tool, that can analyze collection of Java projects and return a optimal set, which covers all requirements, whilst omitting the repetition.This is one of the prime feature of a benechmarking suite, which would satisfy the defined success criteria C1, C2 and C3. Also the characteristics CH1 and relevance would be addressed. In the Qualitas Corpus(Tempero et al. 2010) (Dingsøyr and Moe 2013) collection considered, from which Hermes tries to extract the most suitable subset, as per the requirement. However collections like Qualitas Corpus and others can be outdated or built for other purposes, or may have features that are no longer relevant. A subset is composed by taking projects from various collections, across domains which would be developed by various people, thus covering an entirety of the specification as mentioned in characteristic relevance. Therefore, these projects vary in style, size and methodology. Hence understanding the properties within each of the project that comprises the subset is difficult. Without this understanding it is not possible to establish conclusive results. Hermes helps to overcome these issues with a two step approach, that is:

1. Help to understand the properties within the projects and
2. Help build an optimal collection from the projects which satisfies the requirements.

The queries in Hermes can be extended which enables to further optimize and customize the result set. Hermes is extensible and configurable. However, the requirement is that the collection have to be in java bytecode. The collection can be java programs or libraries. Here the extensibility offered by Hermes is noteworthy. As we mentioned, a good benchmarking tool must have a justifiable, tradeoff between precision and scalability as mentioned in success criteria C2.

Hermes works by searching across the projects looking for the specified feature. Then it collects and sorts the projects that have those specific feature at least once. Hermes is built on OPAL. OPAL can compute and control flow graphs, call graphs, support low level queries, high abstraction, here Scala choco (constraint programming library) is used for selecting optimal collection. Since the tool searches through the collection of libraries to find specified feature it not only helps better understanding of the project set, but also makes sure that the project with the specified feature is chosen from the whole collection.

Hermes Configuration: every selected project has id, class-path and attributes(optional). Queries specified by default are executed. However, their execution can be decided as per the requirements, with possibility to add newer queries. Once queries are executed, Hermes shows if the project contains the feature or not, if the project contains the feature, then the navigation to that particular code portion within the project is enabled. Also these details can be managed enabling easier management of larger collections of projects like Qualitas Corpus. Feature query is a static analysis run on a project which outputs presence of the specified feature or closely related feature. Eg: having only java7 class files. Every feature query has Id of the respective feature and query. Once all the queries are executed, Hermes will select the projects with atleast one occurrence of the feature. If two projects have the same feature occurrence then it will pick the project with lesser number of methods (smaller project). This is niche feature as it tries to keep the overall size of the collection to bare minimum, whilst coveriing the requirement.This is user friendly feature as it helps the user effort to be minimum and hence we can say the criteria C4 is clearly satisfied by the tool. Apart from this the results can be exported to the external file and post processing can be done. This can be a useful feature for academic and scientific purposes.

Some more tool set which analysed were meeting some of the criteria for a successful benchmarking tool. Like, Blackburns DaCapo benchmark suite(Blackburn et al. 2006) is about developing java performance evaluation using static and dynamic software metrics. And, how to develop the collection for the evaluation. Tempero et als Qualitas Corpus is about empirical study of code based on size, content, representativeness and permanence. Livshits et als SecuriBench(Livshits) is about static and dynamic security analysis. All these collections are topic specific and are not updated regularly and that is an issue. Dujmovic presented parameterized approach to generate fully synthetic programs that allow benchmarking and testing. But this approach is not applicable to the real world application. Dujmovic et als automatic benchmark management(Dujmović 2010) can extract and update a collection, but it cannot find the differences between the projects. It is important to note that, these above mentioned benchmarking tools are not as comprehensive as Hermes. Hermes tries to solve the problem by becoming one stoop solution, within the benchmarking domain where tools are scarce.

Evaluation methodology of Hermes :

1. Understanding the test collection and

	CH1: Testing tool capabilities individually	CH2: Easy, correct, and iterative testcase creation	CH3: Automatic test execution and result evaluation
Scanstud	Possible	Possible	Possible
Delta-Bench	Possible	No information	No information
MuBenchPipe	Possible	No information	Possible
Hermes	Possible	Possible	Possible
Differential Benchmark	Possible	Partially Possible	No information

Table 1: Characteristics available within each approach

2. Generating effective integration testing suite.

1) Understanding the test collection: Hermes when run on Qualitas Corpus from 2013, it was able to find out, that none of the project used java 8 features. None of the projects used java FX framework Only one project used java 7 feature. Hereby summarizing that collection from September 2013 cannot be used to test for latest java features.Thus we have a clear proof of Hermes satisfying C1, C2, C3 and characteristics CH1, CH2.

2) Generating effective integration testing suite: Hermes generated the subset from Qualitas Corpus that is optimal(small, yet effective, means to say it meets the requirement specified) and covering all scenarios mentioned. Whilst creating a optimal subset.Therefore reducing the run time from 16.77 minutes (time required to run on hundred projects that comprises Qualitas Corpus) to 2.82 minutes(time required to run on selected subset). And the code coverage was only 1.09% less. This could be bettered by adding the missing features in the queries and re-running Hermes. And the size still would remain considerably smaller than original test collection built. Thus Hermes fulfils its purpose of better understanding the test set and automatic creation of subsets using the derived understanding. This fulfils the characteristic CH3.

7. Differential Benchmark

Both the academic and Industrial surveys do not provide substantial information on the benchmark construction(Johns and Jodeit 2011a) (Li and Cui 2010) . Through the project static analysis tool exposition that was run from 2008 to 2016, software assurance reference dataset was created. However, it is important to note, bug collections in the following dataset may not represent all vulnerabilities in the real world(Greiman 2016) (Kupsch and Miller 2009) (Wilander and Kamkar 2002). This is the reasons for false alarms. False alarm is a background noise that is generated, which can cause attention deviation from the original issue. And this a major source of reason in skipping over certain vulnerabilities during testing.

Juliet test suite, suggests benchmarking real-world application is challenging. And the challenge is due to variability within the real world ecosystem. Dolan gavitt et al (Dolan-Gavitt et al. 2016), suggests method to artificially inject vulnerabilities into source code of real-world application, Large scale automated vulnerability addition (LAVA methodology). However LAVA methodology cannot perform automatic detection and correction of false alarms in real world application. Also this methodology cannot inject some specific vulnerabilities. Cardar and Donaldson (Cadar and Donaldson 2016) suggested program transformation like mutation and metamorphic testing to increase code coverage of SAST tools. But this cannot find bugs in the source code of real world application. Thus the issue with only using SAST tools to analyze real world application is that false alarms for bugs analyzed might be true alarams for bugs that are yet to be analyzed.

So, free open source society (FOSS) (Pashchenko 2017) differential vulnerability fixing methodology was developed to overcome these defects. Here empahsis is on ground truth. Ground truth is the expected correct output. Nuyen et al (Do et al. 2016) suggests to consider code snippet that should produce expected correct output when it had bug and the same snippet after the bug is fixed. Therefore here there are two versions that are being compared which forms the core idea of the evaluation described. This would the fullfil the success criteria C1 and C2 as idea is focused on helps in finding different types of vulnerability with good precision.

Generic methodology for a testing tool evaluation:

1. Run tool on vulnerable version.
2. Extract tool finding.
3. Check for correctness in the finding.

This methodology is suitable for custom collection of projects for which, it is generally know where the errors in the code are present. But this is not the case in the real world applications as there might be multiple bugs. Bugs are usually fixed one after the other and therefore the resultant set must have difference only the in the corrected area, compared to the earlier version prior to running the tool on it.

Proposed methodology of differential benchmarking in FOSS:

1. Find the changed lines of code by comparing fixed and version with bug.
2. Run the tool on the version with the bug

	Accessibility	Affordability	Clarity	Relevance	Solvability	Portability	Scalability	Fairness
Scanstud	Not able to find any repository	Yes	Yes	Not easy to judge	Yes	No, they were not able to adopt all tools	Yes	Yes
Delta-Bench	Yes, they use open source applications	Yes	Not enough info about automation and reproduction	Not easy to judge	Yes	No, they just use tools the supports Java	Yes	It just support Java tools and those that pecifically created for finding security vulnerabilities
MuBenchPipe	Yes, The MuBench (the dataset) is available online	Yes	Yes	Not easy to judge	Yes	No, for example they can not use DroidAssist because its implementation only supports Dalvik Bytecode	Yes	It just support Java API misses
Hermess	No, sufficient information is available on the public availability of the benchmark	Yes	Yes, it is a well defined benchmark framework	Yes, it covers the entirety of tests	Yes, it solves the problem at hand	Yes	Yes, as it supports further query specifications	No, as it currently only supports java bytecode
Differntial Benchmark	No, sufficient information is available on the public availability of the benchmark	No, because it requires a lot of re runs	Yes	Yes, as it tries to address most vulnerabilities individually	Yes	Yes, as it is well defined	Yes	Yes, as it is not bound to a particular programming language

Table 2: Properties within each approach

3. Run the tool on the fixed version
4. Ignore the tool findings that occur in fixed and version with the bug that are irrelevant to the current bug in consideration, as this can be considered as background noise (false alarms)
5. Classify the remaining finding

Here only one bug is fixed at a given run, therefore any other finding not related to the discovered bug is ignored even if it might be right. This is major disadvantage, as it leads to redundant reruns. Here characteristic relevance is not satisfied because the tool is not covering the complete bug spectrum, and we have less information to determine the characteristic CH3. However due to the various re-runs conducted features can be tested individually satisfying the CH1 described above.

Synthetic test suite typically contains only flaw within a testcase. Differential benchmarks has sufficient complexity to identify false alerts. Current tools require some additional input data whilst differential benchmark needs only the fixed and bugged version of projects to generate expected correct code. Differential benchmark is independent of programming languages and bug types. Auto classification of true positive, false negative, false positive, and true negative is also possible. Thus we can say the tool might partially satisfy characteristic CH2, as it cannot be said to be easy due to the re-runs required to arrive at the final conclusion.

8. Conclusion

To summarize, it can be seen from Table 1 that in all the approaches it is possible to test tools capabilities individually. On the other hand, iteratively creating easy and correct testcase, is only possible in Hermes and Scanstud. With regard to automatic execution of benchmark, all the approaches can do that, except Delta bench and differential-benchmark.

Table 2 compares the rest of properties we mentioned in Section 2. The table shows that all approaches are affordable, solvable and scalable. Approaches are not generally portable, this means not all tools in that context could be tested by approaches. They are generally accessible, the sources, tools are available online.

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. .

S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. . URL `http://doi.acm.org/10.1145/1167473.1167488`.

C. Cadar and A. F. Donaldson. Analysing the program analyser. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 765–768, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4205-6. . URL `http://doi.acm.org/10.1145/2889160.2889206`.

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.

T. Dingsøyr and N. B. Moe. Research challenges in large-scale agile software development. *SIGSOFT Softw. Eng. Notes*, 38(5):38–39, Aug. 2013. ISSN 0163-5948. . URL http://doi.acm.org/10.1145/2507288.2507322.

L. N. Q. Do, M. Eichberg, and E. Bodden. Toward an automated benchmark management system. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2016, pages 13–17, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4385-5. . URL http://doi.acm.org/10.1145/2931021.2931023.

B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, May 2016. .

J. Dujmović. Automatic generation of benchmark and test workloads. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, WOSP/SIPEW '10, pages 263–274, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-563-5. . URL http://doi.acm.org/10.1145/1712605.1712654.

T. Greiman. *ICCWS 2016 11th International Conference on Cyber Warfare and Security: ICCWS2016*. ACPIL, 2016. ISBN 9781910810828. URL https://books.google.de/books?id=XD7QCwAAQBAJ.

IEEE. Ieee standard glossary of software engineering terminology, 1990.

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 2011a. .

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 2011b. .

J. A. Kupsch and B. P. Miller. Manual vs. automated vulnerability assessment: A case study. 2009.

P. Li and B. Cui. A comparative study on software vulnerability static analysis techniques and tools. In *2010 IEEE International Conference on Information Theory and Information Security*, pages 521–524, Dec 2010. .

B. Livshits. Defining a set of common benchmarks for web application security.

S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5, 2005.

I. Pashchenko. Foss version differentiation as a benchmark for static analysis security testing tools. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 1056–1058, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5105-8. . URL http://doi.acm.org/10.1145/3106237.3121276.

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. .

M. Reif, M. Eichberg, B. Hermann, and M. Mezini. Hermes: Assessment and creation of effective test corpora. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, pages 43–48, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5072-3. . URL http://doi.acm.org/10.1145/3088515.3088523.

S. Sim, S. Easterbrook, and R. Holt. Using benchmarking to advance research: A challenge to software engineering. pages 74– 83, 06 2003. ISBN 0-7695-1877-X. .

E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference*, pages 336–345, Nov 2010. .

W. F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, May 1998. ISSN 0018-9162. .

J. Wilander and M. Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Nordic Workshop on Secure IT Systems NordSec,2002*, pages 68– . Karlstad University Studies, 2002.