# FOSS Version Differentiation as a Benchmark for Static Analysis Security Testing Tools

Ivan Pashchenko
University of Trento
Via Sommarive, 9 I-38123 Povo
Trento, Italy
ivan.pashchenko@unitn.it

## ABSTRACT

We propose a novel methodology that allows automatic construction of benchmarks for Static Analysis Security Testing (SAST) tools based on real-world software projects by differencing vulnerable and fixed versions in FOSS repositories. The methodology allows us to evaluate "actual" performance of SAST tools (without unrelated alarms). To test our approach, we benchmarked 7 SAST tools (although we report only results for the two best tools), against 70 revisions of four major versions of Apache Tomcat with 62 distinct CVEs as the source of ground truth vulnerabilities.

## CCS CONCEPTS

• **Security and privacy → Software security engineering**;

## KEYWORDS

Static Analysis, Static Application Security Testing Tool, Vulnerability, Software Security, Large-scale Benchmark

**ACM Reference format:**
Ivan Pashchenko. 2017. FOSS Version Differentiation as a Benchmark for Static Analysis Security Testing Tools. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany , September 4–8, 2017 (ESEC/FSE'17),* 3 pages.
https://doi.org/10.1145/3106237.3121276

## 1 RESEARCH PROBLEM & MOTIVATION

Various static analysis security testing (SAST) tools have been proposed to facilitate automatic detection of bugs. Whilst SAST tools are typically sound, they are well known to generate misleading (or irrelevant) alerts [3, 4, 26].

Given this limitation, the selection of an appropriate tool based on empirical grounds becomes extremely important from an industrial perspective. Yet, we only find few *comparative empirical studies* [2, 11, 14]. A typical comparison reports how a single tool performs when applied to different software projects [7]. Such comparisons usually consider only the ability of a tool to produce correct findings, while tools are known to generate false alarms [24] (*Background Noise*), which significantly decreases usability of SAST tools.

In this paper we propose a novel methodology to automatically construct a SAST tool benchmark using real-world software. Our methodology filters *Background Noise*, and therefore, allows us to evaluate "actual" performance of SAST tools.

## 2 BACKGROUND & RELATED WORK

There are several papers providing surveys on academic [14] and industrial [10] tools, but they do not discuss the issue of benchmark construction. On the latter topic in 2008 NIST initiated the Software Assurance Metrics And Tool Evaluation (SAMATE) project dedicated to improving software assurance by measuring the effectiveness of tools and techniques, and identifying gaps in the existing tools and methods. The first large-scale public event named Static Analysis Tool Exposition (SATE), aiming to accumulate test data, was also conducted in 2008. From 2008 to 2016 there were five different stages of the event [5, 19–22]. The main outcome of this project is the creation of the Software Assurance Reference Dataset (SARD) – a collection of synthetic test suites for SAST tool comparison.

There exist several other synthetic bug corpora [1, 12, 15, 28]. However, synthetic test suites may contain bugs, which do not reflect real-world vulnerabilities [9], and therefore, tool evaluation results on synthetic benchmarks may not correspond to the behavior of SAST tools on a real-world software, i.e., the large number of false alarms that are reported in the literature [3, 4, 26].

According to the user guide on the Juliet test suite [1], using real-world software for benchmarking purposes is challenging. Dolan-Gavitt et al. [9] proposed a technique for the artificial injection of vulnerabilities into the source code of real applications (LAVA). However, LAVA does not provide a way to perform automatic false positive evaluation of SAST tool findings. Also, there exist several vulnerability types, which cannot be injected using LAVA.

Cadar and Donaldson [6] proposed to apply program transformations (such as mutation and metamorphic testing) to increase the code coverage of SAST tools, i.e., to identify more data and control flows. Although such techniques increase the applicability of SAST tools for real-world software, they do not help to locate flaws in the source code of an application.

The biggest problem is that the source code of software may contain several different flaws, and therefore, the "false" alarms for the analyzed vulnerability may be "true" alarms for other vulnerabilities, which also present in the code. Hence, only running a SAST tool against real-world software does not allow us to perform the evaluation of tool's findings in terms of false positives.

**Table 1: The SAST tools used in this paper**

| SAST | License | Version | Description |
|---|---|---|---|
| FindBugs | Free | 3.01 | Supports any JVM language and can detect 113 different vulnerability types with over 689 unique API signatures. |
| Fortify SCA | Commercial | 4.42 | Supports 23 programming languages and detects over 700 vulnerabilities. |
| Jlint | Free | 3.1.2 | Works only with Java language. It helps to find more than 50 semantic and syntactic bugs. |
| OWASP LAPSE+ | Free | 2.8.1 | Works only with Java language. The tool is able to identify 12 vulnerability types. |
| PMD | Free | 5.5.1 | Supports 20 programming languages and facilitates finding more than 25 bug types. |
| SonarQube | Free | 5.6 | Supports more than 20 programming languages and covers OWASP Top 10 and the CWE/SANS Top 25 vulnerability types. |
| OWASP YASCA | Free | 2.2 | Supports 14 programming languages and aggregates results from 11 static analysis tools. |

**Table 2: Relative Rankings due to benchmark choices**

*The last column illustrates the relative performance of two tools first by running them on a vulnerable Tomcat version (Direct) vs the relative performance captured by Differential benchmark (Differential).*

| Vuln type | Total vulns | Benchmark | Tool A | Tool B | Ranking |
|---|---|---|---|---|---|
| Bypass | 12 | Direct | 12 | 10 | $A \geq B$ |
| | | Differential | 12 | 1 | $A \gg B$ |
| Cross-Site Scripting | 11 | Direct | 11 | 6 | $A > B$ |
| | | Differential | 10 | 2 | $A \gg B$ |
| Denial of Service | 15 | Direct | 15 | 11 | $A > B$ |
| | | Differential | 13 | 8 | $A > B$ |
| Directory Traversal | 3 | Direct | 3 | 2 | $A \geq B$ |
| | | Differential | 3 | 0 | $A \gg B$ |
| Exec code | 2 | Direct | 2 | 2 | $A = B$ |
| | | Differential | 2 | 0 | $A > B$ |
| Information Disclosure | 23 | Direct | 22 | 23 | $A \leq B$ |
| | | Differential | 22 | 13 | $A \gg B$ |
| Session Fixation | 1 | Direct | 1 | 1 | $A = B$ |
| | | Differential | 1 | 0 | $A \geq B$ |
| Text Injection | 1 | Direct | 3 | 3 | $A = B$ |
| | | Differential | 2 | 0 | $A \geq B$ |

## 3 APPROACH & UNIQUENESS

We intend to use large open source software projects – these projects are well documented, their source code is publicly available, and their software repositories contain many historical vulnerabilities [8, 25]. They can be used for identifying the ground truth – the expected correct output of a tool.

In particular, the information about security fixes in such projects points to the code locations that are likely to be responsible for vulnerabilities. Following the work by Nguyen et al. [17], we identify ground truth code fragments[1] as a set of code lines changed between a revision, which contains a fix for a vulnerability (i.e., fixed), and a revision, which precedes the fixed revision (i.e., vulnerable).

A Direct approach, which researchers typically use for tool evaluation, contains the following steps: (i) run a tool on a vulnerable revision, (ii) extract tool findings, and (iii) identify whether the tool was correct or missed the vulnerable code fragment. The classification of tool findings can be rather simple for synthetic test suites, since a typical synthetic test case contains only one vulnerable flow. Hence, it is only necessary to identify if a tool produced a finding in a test case or missed it. However, for real-world software the classification of findings is not trivial. We say that finding is correct if there is at least one finding that corresponds to the set of lines of code from the ground truth for that particular vulnerability.

Although real-world software projects typically contain several flaws simultaneously, several observations [13, 16] suggest that developers generally fix one vulnerability at a time and do not introduce other changes besides the vulnerability fix itself. Therefore the difference between tool findings returned on a vulnerable and fixed revisions should be only due to the fix. Therefore, we count all the different findings between the two revisions as related to vulnerability, and all other findings as *Background Noise*.

This idea yields the following benchmark conclusion:

(1) identify the ground truth set of code lines by extracting the changed lines of code between vulnerable and fixed revisions of a software project;
(2) run a tool on a vulnerable revision of a software project;
(3) run a tool on a fixed revision of a software project;
(4) eliminate tool findings, which occurs in both vulnerable and fixed revisions (*Background Noise*);
(5) classify the remaining tool findings.

Since we perform analysis of one vulnerability at a time, findings not related to the analyzed vulnerability (even if they are "correct" due to some other issues) may be distracting. Moreover, we are interested in an automatic identification of the code lines related to the vulnerability, and therefore, we measure tool performance regarding a particular vulnerability, rather than assess correctness of all the findings at once.

The approach has several advantages: Synthetic test suites typically contain only one flaw within a test case, instead, the Differential benchmark provides sufficient complexity for tool analysis in terms of false alerts. Current SAST tool benchmarks generally require some additional manually generated input information (i.e. a pattern or a signature) to create a test case, while the proposed benchmark requires only a pair of vulnerable and fixed revisions to extract the ground truth. Such pair may be extracted for every historical vulnerability fix in any software project, and therefore, Differential benchmark is independent on vulnerability types and program languages. Finally, the proposed benchmark allows automatic classification of the cleaned tool findings (without *Background Noise*) to perform *true positive*, *false negative*, *false positive*, and *true negative* evaluations.

## 4 RESULTS & CONTRIBUTIONS

To evaluate our proposal we chose Apache Tomcat, since it is well documented and contains a large number of historical vulnerabilities that can be easily identified in its source code repository.

[1]Such ground truth definition depends significantly on "what else" happens during vulnerability fixes. Several studies observed that for disciplined projects such as Google Chrome, Mozilla's Firefox [16], and Apache project line [13] the majority of security fixes are rather "local". Hence, we assume that the majority of all the changes introduced between vulnerable and fixed revisions are there due to the vulnerability fix.

This project is mainly written in Java, and has more than 800 thousands of lines of code, more than 15 thousands of commits, and 30 contributors[2].

In order to check whether the benchmark was able to discriminate among the tools, we applied the Differential benchmarking procedure to evaluate 7 SAST tools (see Table 1) out of the lists created by OWASP [23], SAMATE [18], and D.Wheeler [27].

Table 2 shows the tool findings in Tomcat by vulnerability types for the two best tools. Although tools identified almost the same number of vulnerabilities according to the Direct benchmark, there is a significant difference in their performance for Bypass, Cross-Site Scripting, Directory Traversal, and Information Disclosure vulnerability types according to the Differential benchmark. This happens, since a SAST tool may identify an actually fixed line of code in a fixed revision as vulnerable. Such findings would be eliminated by the Differential Benchmark: Tool A distinguishes the majority of vulnerability fixes, while Tool B does not really recognize those fixes, and therefore, generates significant *Background Noise*.

In this paper we demonstrate the ability of the Differential benchmark to distinguish tools with similar performance on their ability to spot various vulnerability types (*true positive* evaluation) according to the Direct approach. We believe, that Differential benchmarks would report interesting tool evaluations from a practical perspective. These results may help software development companies to select the most appropriate tools for their projects, and inspire tool developers to improve SAST tools to produce "clean" results.

## ACKNOWLEDGMENTS

This work was performed under advisement of prof. Fabio Massacci at the University of Trento. Such results would not have been possible to achieve without his guidance and mentorship.

Additionally I would like to thank Stanislav Dashevskyi from the University of Trento for his technical expertise in static analysis and generous support on all stages of the current research.

## REFERENCES

[1] National Security Agency Center for Assured Software (NSA CAS). 2012. Juliet Test Suite v1.2 for Java User Guide. (2012).

[2] Nuno Antunes and Marco Vieira. 2015. Assessing and Comparing Vulnerability Detection Tools for Web Services: Benchmarking Approach and Examples. 8, 2 (2015), 269–283.

[3] Joao Eduardo M. Araujo, Silvio Souza, and Marco Tulio Valente. 2011. Study on the relevance of the warnings reported by Java bug-finding tools. 5, 4 (2011), 366–374.

[4] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. 2007. Evaluating static analysis defect warnings on production software.

[5] Paul E. Black and Athos Ribeiro. 2016. *SATE V Ockham Sound Analysis Criteria.* Technical Report. National Institute of Standards and Technology (NIST).

[6] Cristian Cadar and Alastair F. Donaldson. 2016. Analysing the Program Analyser *(ICSE '16).* ACM, New York, NY, USA, 765–768. https://doi.org/10.1145/2889160.2889206

[7] Aurelien Delaitre, Bertrand Stivalet, Elizabeth Fong, and Vadim Okun. 2015. Evaluating Bug Finders–Test and Measurement of Static Code Analyzers.

[8] Lisa Nguyen Quang Do, Michael Eichberg, and Eric Bodden. 2016. Toward an automated benchmark management system. ACM, 13–17.

[9] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-scale automated vulnerability addition.

[10] Pär Emanuelsson and Ulf Nilsson. 2008. A comparative study of industrial static analysis tools. 216 (2008), 5–21.

[11] Martin Johns and Moritz Jodeit. 2011. Scanstud: a methodology for systematic, fine-grained evaluation of static analysis tools.

[12] James A Kupsch and Barton P Miller. 2009. Manual vs. automated vulnerability assessment: A case study. 83–97.

[13] Daoyuan Li, Li Li, Dongsun Kim, Tegawendé F Bissyandé, David Lo, and Yves Le Traon. 2016. Watch out for This Commit! A Study of Influential Software Changes. *arXiv preprint arXiv:1606.03266* (2016).

[14] Peng Li and Baojiang Cui. 2010. A comparative study on software vulnerability static analysis techniques and tools.

[15] Benjamin Livshits. 2005. Stanford SecuriBench. *Online: http://suif. stanford. edu/livshits/securibench* (2005).

[16] Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. 2015. An automatic method for assessing the versions affected by a vulnerability. 21, 6 (2015), 2268?–2297.

[17] Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. 2016. An automatic method for assessing the versions affected by a vulnerability. *Empirical Software Engineering* 21, 6 (2016), 2268–2297.

[18] NIST. 2016. SAMATE list of Source Code Security Analyzers. (2016). https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

[19] Vadim Okun, Aurelien Delaitre, and Paul E. Black. 2010. The second static analysis tool exposition (SATE) 2009. (2010), 500–287.

[20] Vadim Okun, Aurelien Delaitre, and Paul E. Black. 2011. Report on the Third Static Analysis Tool Exposition (SATE 2010). (2011), 500–283.

[21] Vadim Okun, Aurelien Delaitre, and Paul E. Black. 2013. Report on the static analysis tool exposition (SATE) IV. 500 (2013), 297.

[22] Vadim Okun, Romain Gaucher, and Paul E. Black. 2009. Static analysis tool exposition (SATE) 2008. 5, 00-2 (2009), 79.

[23] OWASP. 2017. OWASP list of Source Code Analysis Tools. (2017). https://www.owasp.org/index.php/Source_Code_Analysis_Tools

[24] Latifa Ben Arfa Rabai, Barry Cohen, and Ali Mili. 2015. Programming Language Use in US Academia and Industry. 14, 2 (2015), 143.

[25] Michael Reif, Michael Eichberg, Ben Hermann, and Mira Mezini. 2017. Hermes: assessment and creation of effective test corpora. ACM, 43–48.

[26] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. 2008. Predicting accurate and actionable static analysis warnings: an experimental approach.

[27] David Wheeler. 2015. Static analysis tools for security. (2015). http://www.dwheeler.com/essays/static-analysis-tools.html

[28] John Wilander and Mariam Kamkar. 2002. A comparison of publicly available tools for static intrusion prevention. (2002).

[2]This information is taken from https://www.openhub.net/ (last accessed on June 2017).