Delta-Bench: Differential Benchmark for Static Analysis Security Testing Tools

Ivan Pashchenko
University of Trento, Italy
Email: ivan.pashchenko@unitn.it

Stanislav Dashevskyi
University of Trento, Italy
Email: stanislav.dashevskyi@unitn.it

Fabio Massacci
University of Trento, Italy
Email: fabio.massacci@unitn.it

Abstract—**Background: Static analysis security testing (SAST) tools may be evaluated using synthetic micro benchmarks and benchmarks based on real-world software.**

Aims: The aim of this study is to address the limitations of the existing SAST tool benchmarks: lack of vulnerability realism, uncertain ground truth, and large amount of findings not related to analyzed vulnerability.

Method: We propose Delta-Bench – a novel approach for the automatic construction of benchmarks for SAST tools based on differencing vulnerable and fixed versions in Free and Open Source (FOSS) repositories. To test our approach, we used 7 state of the art SAST tools against 70 revisions of four major versions of Apache Tomcat spanning 62 distinct Common Vulnerabilities and Exposures (CVE) fixes and vulnerable files totalling over 100K lines of code as the source of ground truth vulnerabilities.

Results: Our experiment allows us to draw interesting conclusions (e.g., tools perform differently due to the selected benchmark).

Conclusions: Delta-Bench allows SAST tools to be automatically evaluated on the real-world historical vulnerabilities using only the findings that a tool produced for the analyzed vulnerability.

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

I. Introduction

Designing a benchmark with real-world software is a challenging task [1]. Therefore, existing approaches either insert bugs artificially [2], [3], or use historical bugs from the software repository of a project [4]. Artificial bug injection is often difficult to verify (see [2, p.2]), whilst historical vulnerabilities may represent only a subset of the ground truth.

Purely synthetic benchmarks [5] eliminate the above problems by isolating vulnerabilities into atomic tests that represent small applications, so that each of them contains only the code relevant to a vulnerability to be tested or a deliberately inserted false positive, and some other closely related code which may be required for the vulnerable code to compile.

Still, for practical purposes one would like to know how a tool scales when moving from synthetic to real-world software. The biggest problem of using real-world software for benchmarking is that the code usually contains several "issues" simultaneously. Hence, the tool may produce many alarms not related to the vulnerability type for which we would like to use the software as a benchmark. Some of those alerts may be wrong but others may be "true" for other issues (see the discussion on the Juliet test suite [1, p.2]).

Developers perceive these large amounts of alerts (*Background Noise*) to be a "pain in the neck" that goes along with the practical usage of static analysis security testing (SAST) tools [6]. Therefore, our goal is to devise a methodology for benchmarking SAST tools that would combine both benefits of synthetic benchmarks and real-world software: (1) "isolate" findings relevant to the ground truth vulnerabilities, and (2) assess the *Background Noise* that various tools may generate in practice.

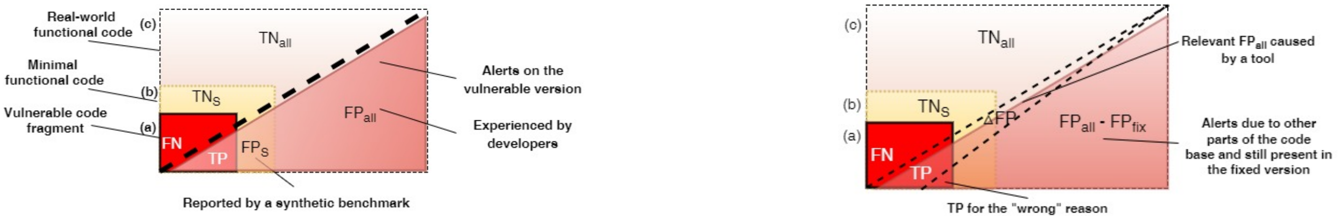
The next section (§II) provides an overview of the existing SAST tool benchmarks. Then we describe the Delta-Bench approach on how to use real-world software for benchmarking SAST tools (§III) and discuss our data collection process for the Delta-Bench evaluation (§IV), and check whether our assumptions on the data are satisfied (§V). Finally, we discuss the results of the empirical evaluation (§VI), the threats to validity (§VII) and conclude (§VIII).

II. SAST benchmarks

Several studies survey the performance of SAST tools. For example, Li and Cui [7] provided a technical description of seven open source tools, describing their experience with three of them in terms of false positive and false negative rates on their own test code. Emanuelsson and Nilsson [8] described three commercial tools, providing case studies on their evaluation at Ericsson. Such comparisons provide valuable insights for security researchers, but cannot be easily replicated.

The first large-scale public event named Static Analysis Tool Exposition (SATE), aiming to accumulate test data, was conducted in 2008 [9] and is now at the fifth edition [10]. One of the main outcomes of this project is the creation of the Software Assurance Reference Dataset (SARD), which contains synthetic test suites for SAST tool comparison. In the academic domain Johns and Jodeit [5] introduced a common methodology for systematic evaluation of SAST tools using a benchmark composed of small programs that contain artificially injected vulnerabilities. However, artificial vulnerabilities may differ from the real-world ones, and therefore the evaluation results may differ.

A possible way to adapt real-world software for benchmarking purposes is to modify the original source code of an application to increase the potential coverage of SAST tools. Examples are mutation [11] and metamorphic [12] testing. Although such techniques may expand the applicability of


Real-world functional code
(c)
TN$_{all}$
Minimal functional code
(b)
TN$_S$
Vulnerable code fragment
(a)
FN
TP
FP$_S$
FP$_{all}$
Alerts on the vulnerable version
Experienced by developers
Reported by a synthetic benchmark
(c)
TN$_{all}$
Relevant FP$_{all}$ caused by a tool
(b)
TN$_S$
ΔFP
FP$_{all}$ - FP$_{fix}$
Alerts due to other parts of the code base and still present in the fixed version
(a)
FN
TP
TP for the "wrong" reason

(a) Directly running the tool on the vulnerable version

(b) Considering different alerts on vulnerable and fixed versions

The tool output after analyzing a vulnerable version would likely contain many alerts not related to the analyzed vulnerability (Figure 1a). Such alerts should be also present in the tool output on a fixed version. Hence, the alert subtraction may significantly decrease the amount of irrelevant alerts (Figure 1b).

Fig. 1. An example of findings of a SAST tool

static analysis tools to real-world software, they do not help automatic warning classification and do not solve a problem on how to compare outputs of different SAST tools [1].

The solution proposed by Dolan-Gavitt et al. [3] (LAVA) suggests an artificial injection of vulnerabilities into the source code of real applications. Although this technique allows benchmarks to be created automatically, it does not allow false positive evaluation of SAST tools [3, §VIII]. The current implementation of LAVA is limited only to one vulnerability type (buffer overflow), and some vulnerability types cannot be injected using LAVA approach (e.g., logic errors, crypto flaws, and side-channel vulnerabilities).

III. Benchmark construction

Similarly to Livshits and Lam [13], and Delaitre et al. [14], 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. Therefore, they can be used for identifying the ground truth – the expected correct output of a tool.

In a software repository we typically have available:

- C_{fixed} – the source code of a revision of a software project that was created to fix a security vulnerability.
- C_{vuln} – the source code of the last vulnerable revision that precedes the C_{fixed}.

Figure 1a demonstrates a typical situation regarding the alerts of a SAST tool when running it on C_{vuln}. Ideally, the tool output for C_{vuln} should contain only alerts related to the vulnerability (TP area of the (a) square in Figure 1a). A tool may not identify all the code related to the vulnerability in C_{vuln} (FN area in the (a) square in Figure 1a).

SAST tools tend to generate many false alerts [15], so the tool output may contain false positive alerts (FP_S) related to the vulnerable set of files in the squared area (b)[1]. In case of a synthetic benchmark there would be no other alarms.

Unfortunately, real-world software projects usually contain many "issues" distributed between all the project files. Hence, a SAST tool would generate many more alarms (some of them may correspond to other flaws present in a project) not related to the analyzed vulnerability (the FP_{all} area in the (c) square). The false alerts FP_{all} may be distracting, and therefore unwanted by developers [6].

Since the false alarms FP_{all} are unrelated to the vulnerable code fragment of the benchmark test, they are likely to be present in the tool outputs for both C_{vuln} and C_{fixed}. The successful fix of a vulnerability implies that the source code does not contain the vulnerable code anymore. Hence, the tool output on C_{fixed} should not contain alerts related to the vulnerability and observed in C_{vuln}. We can then subtract common alerts and evaluate the "actual" tool performance considering only the alerts relevant to the analyzed vulnerability. Figure 1b shows the alert distribution in the vulnerable code base after eliminating the alerts common for C_{vuln} and C_{fixed}.

The above intuition corresponds to our proposed process to generate a benchmark. The first three steps determine the ground truth, the others separate the *Background Noise* from the findings related to the specific vulnerability – *Signal*:

1) Identify a suitable project that provides sufficient information about security fixes, so that they can be identified in the source code (e.g., Common Vulnerabilities and Exposures (CVE) entries in the Git logs).
2) For each fixed vulnerability, identify a pair $\langle C_{vuln}, C_{fixed} \rangle$ – this information can be obtained either from the repository commit logs, vulnerability databases, or security notes.
3) Extract the source code constructs (files, and lines of code) modified during a fix (thus, likely vulnerable): we use the *diff* tool of a version control system.
4) Run a tool on a vulnerable version of the software C_{vuln}, and on the fixed version C_{fixed} (the fix must be the only difference between the two versions);
5) The $Signal$ are the alerts differing between the tool outputs on C_{vuln} and C_{fixed}. Metrics (TP, FP, etc.) are only calculated on the alerts related to $Signal$. As *Background Noise* we consider the code lines from the same files that were reported for both C_{vuln} and C_{fixed}.

The next step in the process is to assess the tool findings and classify them as true or false positives.

Unfortunately, various tools may return different code lines for the same issue [1]. Moreover, a security fix may not touch the exact vulnerable line, but may modify a line that is relevant to the vulnerable one and is located "closely" to it (i.e., within the same method). An insertion of a sanitization mechanism for the user input may be an example of such a fix. Hence, a direct comparison of the lines reported by a tool with the code lines changed during a security fix would be misleading.

[1] We consider such alerts as false positives, since we concentrate only on one vulnerability at a time.

Algorithm 1: Differential tool assessment

```
input : A vulnerable revision $C_{vuln}$ and a fixed revision $C_{fixed}$
output: Differential assessment of tool findings on file-level
// identification of the ground truth
$GTF \leftarrow \{file | file \in diff(C_{fixed}, C_{vuln})\}$ // $diff(C_1, C_2)$ is
    a diff tool of a version control system
$BackgroundNoise \leftarrow \emptyset$ ;
// $Alerts(C)$ represents a tool output after
    running on $C$ and returns a set of $< file, line >$.
for each $< file, lile > \in Alerts(C_{fixed})$ do
    // $Adjust(file, line, C_1, C_2)$ converts positions of
        lines in $C_1$ into relative positions in $C_2$
    $line^* \leftarrow Adjust(file, line, C_{fixed}, C_{vuln})$;
    if $< file, line^* > \in Alerts(C_{vuln})$ then
        $BackgroundNoise \leftarrow BackgroundNoise \cup$
                                $\{< file, line^* >\}$;
    end
end
$Signal \leftarrow Alerts(C_{vuln}) \backslash BackgroundNoise$;
// identification of a set of correct findings
$TP_\Delta \leftarrow \emptyset$;
for each $< file, line > \in Signal$ do
    if $file \in GTF$ then
        $TP_\Delta \leftarrow TP_\Delta \cup \{file\}$;
    end
end
// classification of all the remaining findings
$FN_\Delta \leftarrow GTF \backslash TP_\Delta$;
$FP_\Delta \leftarrow Alerts(C_{vuln}) \backslash GTF$;
$TN_\Delta \leftarrow file(C_{vuln}) \backslash (GTF \cup TP_\Delta)$;
```

TABLE I
SOFTWARE PROJECTS USED FOR EVALUATION IN THIS PAPER

The table shows the characteristics of an average vulnerable version (C_{vuln}) extracted from both Scanstud and Apache Tomcat: the total number of files in the revision, the number of vulnerable files, and the Prevalence rate (the ratio of vulnerable files in the revision). For Scanstud each vulnerable revision consists from one vulnerable file, while for Apache Tomcat an average revision may contain more than 1600 files with only 2-3 actually vulnerable files.

	#Files	#Vuln files	Prevalence Rate
	μ ($\pm$ σ)	μ ($\pm$ σ)	μ ($\pm$ σ)
Scanstud	1 (± 0)	1 (± 0.0)	1.0 (± 0.0)
Tomcat	1626 (± 318)	2.54 (± 3.34)	0.0011 (± 0.0024)

In this short paper we use files as a first approximation for finding classification: a TP is a file that has been changed during the security fix and for which there exists an alert pointing to that file. We extended our approach to work with methods[2], and plan to extend it to hunks and program slices.

Algorithm 1 shows how to filter *Background Noise* and classify the "clean" tool findings. Since a line of code in a vulnerable revision may have a different position in a fixed revision, we have to convert the positions of the identified lines obtained after running a tool on the fixed version, in order to make the set of code lines comparable (we used *lhdiff* [16] for this purpose).

IV. Data selection for evaluation

In order to understand what could be the difference in results when comparing the performance of various tools, we had to select an appropriate synthetic benchmark, as well as a real-world software project. To have a fair comparison, both the synthetic benchmark and the project should be written in the same programming language (we selected Java, which is the most popular programming language since 2004[3]).

TABLE II
THE SAST TOOLS TESTED FOR THIS RESEARCH

SAST	License	Version	Description
FindBugs	Free	3.01	Supports any JVM language and can detect 113 different vulnerability types.
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 can identify 12 vulnerability types.
OWASP YASCA	Free	2.2	Supports 14 programming languages and aggregates results from 11 static analysis tools.
PMD	Free	5.5.1	Supports 20 programming languages and facilitates finding more than 25 bug types.
SonarQube	Free	5.6	Supports 20 programming languages and covers OWASP Top 10 vulnerability types.

We used Scanstud by Johns and Jodeit [5] as a synthetic benchmark, since it contains a large number of tests and provides both "vulnerable" and "fixed" versions of each test.

We used Apache Tomcat as a real-world application, since it is mainly written in Java, and contains a large number of historical vulnerabilities that can be easily identified in its source code repository. This project has more than 800 thousands of lines of code, more than 15 thousands of commits and 30 unique contributors.

To demonstrate our approach we identified 38 vulnerable-fixed file pairs from Scanstud. From Apache Tomcat we extracted 70 revisions with 62 distinct CVEs, which contain 178 vulnerable files out of the total amount of 113842 files. There are some common CVEs for different versions of the project. A revision was selected if it was possible (i) to precisely identify that the particular CVE was fixed, and (ii) to successfully build the project version. Table I shows the averages and standard deviations of total number of files, number of vulnerable files, and the prevalence rate in one experimental unit extracted from both Apache Tomcat and Scanstud, and Table III lists the vulnerability types present in both code bases.

To select SAST tools for benchmarking we considered the lists created by OWASP[4] and SAMATE[5]. Out of these lists we selected the tools that (1) support Java, (2) are specifically created for finding security vulnerabilities, and (3) can be easily automated. From the commercial tools, we could obtain

[2] We do not show the results on methods due to the space constraints. However, we plan to report them in the extended version of this paper.

[3] According to the two indexes used by IEEE Spectrum (http://spectrum.ieee.org/) to assess popularity of a programming language: (i) Tiobe index (http://www.tiobe.com/tiobe-index/), which combines data about search queries from 25 most popular websites of Alexa; and (ii) PYPL index (http://pypl.github.io/PYPL.html), which uses Google search queries.

[4] OWASP Source Code Analysis Tools list: https://www.owasp.org/index.php/Source_Code_Analysis_Tools

[5] SAMATE Source Code Security Analyzers list: https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

an academic license for Fortify SCA (Checkmarx asked for several thousands euros a year). Table II contains the list of the selected SAST tools. All the tools were used in their default configuration. Due to the licensing issues, we obfuscate the real names of the tools while presenting their results.

We could not use all the tools from Table II for evaluation. One tool generated many issues both on Scanstud and Tomcat, but there were no security issues among them. FindBugs identified 21 out of 38 issues on Scanstud, but was not able to spot any vulnerabilities in Tomcat. This might happen because Tomcat contains many different vulnerability types, not all supported by FindBugs. However, the most likely reason is that Apache Tomcat developers actually used FindBugs (and also Coverity)[6], hence they may have already fixed the findings before committing to the source code repository. We assume that this also caused the absence of findings from three other tools on Tomcat. Moreover, two of them are unable to identify the vulnerability types present in Scanstud.

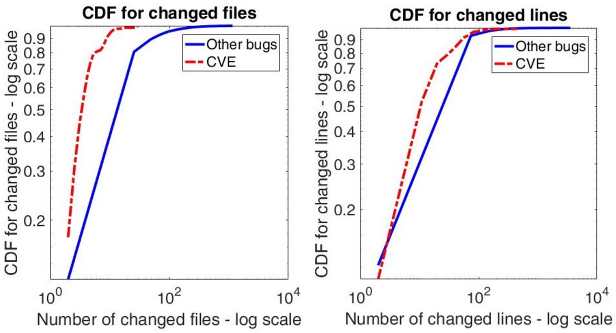
Instead, Tool A and Tool B use smart algorithms of data and control flow analysis and are constantly updated, and therefore, they identified some vulnerabilities in both Scanstud and Apache Tomcat. Hence, we will use them to demonstrate the preliminary evaluation of our approach.

V. Vulnerability fixes are "local"

Similarly to the *Defects4J* benchmark proposed by Just et al. [4], our benchmark construction methodology depends on "what else" happens during vulnerability fixes. Vulnerability fixes must not contain other changes that are not relevant to the purpose of the fix (e.g., refactorings or new features). Regular bug fixing may involve several files and several little "polishing" touches in several parts of the code base [17], [18]. If this was the case for security bugs, our ground truth could hardly be classified as such, as it would include a large amount of irrelevant changes.

Several studies observed that for disciplined projects such as Google Chrome, Mozilla's Firefox [19], and Apache Commons [20] the majority of security fixes are rather "local", which allows us to assume that in many cases the vulnerable code consists of closely related chunks located within a single file (or a handful of files).

To check if our assumption holds for Apache Tomcat, we performed a comparative analysis of known security fixes versus other commits not related to security vulnerabilities (Figure 2). The distributions of the numbers of changed files and lines of code suggest that non-security changes are likely to be significantly larger (e.g., may spread to hundreds of files and involve thousands of lines), while security fixes are rather "local" (mostly a couple of files and less than 100 lines). Therefore, it is not likely that security fixes from our sample would contain irrelevant changes.



The cumulative distribution function (CDF) for CVE fixes demonstrates that CVE fixes tend to be much more local than all other fixes in terms of changed files. Also in terms of changed lines CVE fixes are more local than all other fixes: CDF for changed lines during usual fixes is sharper than CDF for changed lines during CVE fixes.

Fig. 2. Comparing vulnerability fixes with non-security changes from the Apache Tomcat source code repository

TABLE III
Relative Rankings due to benchmark choices

The last column illustrates the relative performance of the two best tools in Table II first by running them on the vulnerable version (Direct row) versus the relative performance captured by Delta-Bench by removing the Background Noise according to the Algorithm 1 (Delta-Bench row).

Vuln type	Total vulns	Benchmark	Tool A	Tool B	Ranking
Cross-Site Scripting	35	Scanstud	7	35	$A \ll B$
SQL injection	3	Scanstud	3	3	$A = B$
Bypass	12	Direct	12	10	$A \geq B$
		Delta-Bench	12	1	$A \gg B$
Cross-Site Scripting	11	Direct	11	6	$A > B$
		Delta-Bench	10	2	$A \gg B$
Denial of Service	15	Direct	15	11	$A > B$
		Delta-Bench	13	8	$A > B$
Directory Traversal	3	Direct	3	2	$A \geq B$
		Delta-Bench	3	0	$A \gg B$
Exec code	2	Direct	2	2	$A = B$
		Delta-Bench	2	0	$A > B$
Information Disclosure	23	Direct	22	23	$A \leq B$
		Delta-Bench	22	13	$A \gg B$
Session Fixation	1	Direct	1	1	$A = B$
		Delta-Bench	1	0	$A \geq B$
Text Injection	3	Direct	3	3	$A = B$
		Delta-Bench	2	0	$A \geq B$

VI. Empirical evaluation

From the perspective of "finding a vulnerable file", SAST tools are conceptually similar to defect predictors [21]. They provide the likelihood of the presence of software defects in a file (or a method) based on source code metrics (e.g., the size of the source code, cyclomatic complexity, etc.), development history (code churn, number of contributors, etc.), or other features. For example, Neuhaus et al. [22] used the information about past vulnerabilities in software components of Mozilla Firefox to identify the components that will likely cause vulnerabilities in the future; whereas, Shin et al. [23] assessed the defect prediction capabilities of traditional source code metrics versus developer activity metrics.

[6] FindBugs is integrated into Apache Tomcat build scripts. Also, Apache Tomcat is listed among the projects that use Coverity Scan service (https://scan.coverity.com/projects/apache-tomcat).

TABLE IV
AVERAGES OF FILE-LEVEL FINDINGS

There is a difference between tool performances on Scanstud and real-world benchmarks: Tool B shows better results on Scanstud, but on real-world software Tool A performs better. Delta-Bench allows us to see this difference even better: the distance between means of tool metrics becomes bigger. In some cases there even occurs inversions, i.e., Tool B produces more false positives when executed on the vulnerable version (Direct row), while after subtracting $BackgroundNoise$ Tool A starts to produce more False alarms (Delta-Bench row).

Metric	Benchmark	Mean of # Files		Ranking
		Tool A	Tool B	
TP	Scanstud	0.3	1.0	$A < B$
	Direct	2.2	1.7	$A > B$
	Delta-Bench	1.8	1.2	$A > B$
FN	Scanstud	0.7	0.0	$A > B$
	Direct	0.4	0.9	$A < B$
	Delta-Bench	0.7	1.4	$A < B$
FP	Scanstud	n/a	n/a	One file
	Direct	554.0	677.0	$A < B$
	Delta-Bench	402.0	254.0	$A > B$
Signal	Scanstud	0.3	1.0	$A < B$
	Delta-Bench	403.0	252.0	$A > B$
Background Noise	Scanstud	0.0	0.0	$A = B$
	Delta-Bench	152.0	426.0	$A < B$

Whilst defect predictors mostly use statistical methods for identifying the relationships between the source code features and software defects, SAST tools use semantic-based information, and should have better performance [24].

Obviously, Tool A and Tool B (the two tools from Table II selected for evaluation) did not produce any *Background Noise* on Scanstud. Hence, we will report only one result for Scanstud, while there will be two results for Apache Tomcat.

At first we assessed whether a tool is able to identify a particular type of vulnerability. A tool succeeded, if there is at least one correct finding (i.e., at least one TP). Table III reports tool performances by vulnerability types extracted from Scanstud and Apache Tomcat.

On Scanstud Tool B identified all 38 vulnerabilities, while Tool A found only 10 vulnerabilities (7 Cross-Site Scripting and 3 SQL injection). The analysis of tool findings in Tomcat by vulnerability types showed different results. According to the Direct approach (running a tool on the vulnerable version) for finding classification, Tool A was able to find almost all vulnerabilities, while Tool B missed some of them. However, after removing the *Background Noise* the difference in tool performances changed significantly: Tool A still identified the majority of the vulnerabilities, but Tool B spotted only several of them. Some TP were made by chance, and they were filtered by Delta-Bench (removing the *Background Noise* according to the Algorithm 1). The significant change happened for Bypass, Cross-Site Scripting, Directory Traversal, and Information Disclosure vulnerability types.

Table IV shows the average results for each tool. There is a significant difference in the relative ranking of the tools. On Scanstud Tool B was able to spot all the vulnerable files, while Tool A missed some of them. Hence, Tool B performs better in terms of both TP and FN. By design, synthetic benchmarks have no non-vulnerable files in C_{vuln}, hence the "n/a" for FP in Table IV for Scanstud.

According to the Direct approach on Tomcat, Tool A produced more TP, less FN and FP comparing to Tool B, which shows that Tool A performs better. Delta-Bench increased the difference between the two tools, and therefore, made it possible to distinguish the two tools better. However, there is an inversion in the amount of FP: Tool B shows more FP according to the Direct approach, and Tool A shows more FP according to the Delta-Bench. This happens due to the fact, that Tool B produced much more warnings (i.e., *Background Noise*) than Tool A. Therefore, when we subtracted this *Background Noise* from the tool findings, this eliminated the majority of FP produced by Tool B.

TABLE V
AVERAGES OF PRECISION, RECALL AND NEGATIVE PRECISION ON FILE-LEVEL

Running tools on different types of benchmarks showed different performances in terms of Precision, Recall, and Negative Precision. In some cases even a ranking reversal. Noise removal allows a better discrimination between tools, since the distance between metrics becomes more pronounced.

Metric	Benchmark	Tool A	Tool B	Result
Precision	Scanstud	0.3	1.0	$A < B$
	Direct	0.0039	0.0025	$A > B$
	Delta-Bench	0.0065	0.0044	$A > B$
Recall	Scanstud	0.3	1.0	$A < B$
	Direct	0.9	0.7	$A > B$
	Delta-Bench	0.7	0.4	$A > B$
Negative Precision	Scanstud	n/a	n/a	No TN
	Direct	0.9998	0.9995	$A > B$
	Delta-Bench	0.9995	0.9991	$A > B$

Table V shows the averages of *Precision*, *Recall*, and *Negative Precision* for Tool A and Tool B. As it was mentioned for the average tool findings (Table IV), the tools perform differently when executed on synthetic and real-world software. This is also visible for *Precision* and *Recall*. By design, there were only vulnerable files for C_{vuln} in Scanstud, and therefore, we cannot report any results for *Negative Precision*. Similarly to the observations on the average tool findings, Delta-Bench allows tools to be better differentiated than the Direct approach.

Shaha et al. [25] in their study of bug reports showed that low-severity bugs can be very important (e.g., due to classification errors), therefore for our analysis we considered all warnings regardless of their severity, as we believe they can be also a subject to similar classification errors.

We also selected only the findings with the top two severity levels. Both tools produced a negligible amount of TP, when limited to high severity findings. As it was mentioned in section IV, Apache Tomcat developers used other SAST tools, and therefore, they may have already fixed all the high severity findings produced by those tools.

VII. THREATS TO VALIDITY

Our results may be affected by errors in the data collection process, the accuracy of the information about security fixes in Apache Tomcat, and the mechanism for extracting either ground truth or code fragments pointed by alerts.

Bias in the data collection: although static analysis tools produce different kinds of output, we bring them to a common denominator by reducing the output to vulnerability warnings mapped to the source code locations. In this way we might overlook some other features of tools that, for example, can enhance user experience and may influence the selection.

Bias in the information about vulnerability fixes: there are few fixes that span over several commits (e.g., CVE-2009-3555), for which we used only the last commit that concluded the fix to reconstruct the vulnerable code fragment. It might be possible, that both ground truth and warning code fragments that we extract do not reflect the full vulnerable code sample.

Bias in code base selection: Our private communications with an industrial SAST specialist suggest that such tools may be optimized towards finding vulnerabilities specific to web applications (e.g., XSS or SQLi). Although Apache Tomcat is a web server, it still has a handful of vulnerabilities specific to web applications. Hence, we believe that this threat is limited.

Bias in SAST tool selection: we present results obtained only from two SAST tools. However, we use these tools to demonstrate the methodology without making claims about the overall performance of these tools and only show how different benchmarking methods may change the results.

VIII. Conclusions

We propose Delta-Bench – a novel approach that uses fixes of historical vulnerabilities from the existing FOSS projects as a ground-truth set of vulnerabilities to automatically construct benchmarks for SAST tools by (suitably) differencing SAST alerts from vulnerable and fixed versions. The approach allows us to evaluate SAST tools using only the findings that a tool produced for the analyzed vulnerability (without considering the *Background Noise*). For benchmark construction Delta-Bench requires only a pair of vulnerable and fixed versions of a software code as an input.

We demonstrated Delta-Bench on a synthetic benchmark Scanstud and a set of historical vulnerabilities extracted from Apache Tomcat. Our experiments already showed significant insights between the two tools: we found that a relative tool ranking may be reverted by a different benchmarking method.

As for the future work, we plan to demonstrate Delta-Bench by using it for evaluation of different commercial and open-source SAST tools, and on a larger set of real-world software projects as a source of historical vulnerabilities (beyond Java). Due to space constraints in this short paper we show the initial results only at a file-level granularity. We have already extended Delta-Bench to work with methods, and are starting to extend it to hunks and program slices. The approach could be also applied to other types of bugs provided the assumption on the locality of fixes also applies to those bugs (as we have shown in Section V for security bugs).

By using Delta-Bench software development companies may select the most appropriate tool for their projects and tool developers may improve SAST tools for sharper results.

Acknowledgments

We would like to thank Achim D. Brucker, Paolo Tonella, and the members of the security group in Trento for their helpful comments on preliminary versions of this work. They greatly helped to improve this paper. We would also like to thank Katsiaryna Labunets from the TU Delft for her useful suggestions on the final stages of this work.

References

[1] National Security Agency Center for Assured Software (NSA CAS), "Juliet Test Suite v1.2 for Java user guide," 2012.

[2] J. Dahse and T. Holz, "Static detection of second-order vulnerabilities in web applications," in *Proc. of USENIX'14*, 2014.

[3] 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 *Proc. of SSP'16*, 2016.

[4] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. of ISSTA'14*, 2014.

[5] M. Johns and M. Jodeit, "Scanstud: a methodology for systematic, fine-grained evaluation of static analysis tools," in *Proc. of ICSTW'11*, 2011.

[6] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *Proc. of ASE'16*, 2016.

[7] P. Li and B. Cui, "A comparative study on software vulnerability static analysis techniques and tools," in *Proc. of ICITIS'10*, 2010.

[8] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *ENTCS*, vol. 216, pp. 5–21, 2008.

[9] V. Okun, R. Gaucher, and P. E. Black, "Static analysis tool exposition (SATE) 2008," *NIST SP*, vol. 5, no. 00-2, p. 79, 2009.

[10] P. E. Black and A. Ribeiro, "SATE V Ockham sound analysis criteria," NIST SP, Tech. Rep., 2016.

[11] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *TSE*, vol. 37, no. 5, pp. 649–678, 2011.

[12] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," HKUST-CS98-01, Hong Kong University of Science and Technology, Tech. Rep., 1998.

[13] B. V. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis." in *Proc. of USENIX'13*, 2005.

[14] A. Delaitre, V. Okun, and E. Fong, "Of massive static analysis data," in *Proc. of SERE'13*, 2013.

[15] L. Rabai, A. Ben, B. Cohen, and A. Mili, "Programming language use in us academia and industry," *Inf. in Education*, vol. 14, no. 2, p. 143, 2015.

[16] M. Asaduzzamad, R. K. Chanchal, K. A. Schneider, and M. Di Penta, "Lhdiff: A language-independent hybrid approach for tracking source code lines," *Proc. of ICSME'13*, 2013.

[17] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proc. of ICSE'11*, 2011.

[18] K. Herzig, S. Just, and A. Zeller, "The impact of tangled code changes on defect prediction models," *Emp. Soft. Eng.*, vol. 21, no. 2, pp. 303–336, 2016.

[19] V. H. Nguyen, S. Dashevskyi, and F. Massacci, "An automatic method for assessing the versions affected by a vulnerability," *Emp. Soft. Eng.*, vol. 21, no. 6, pp. 2268–2297, 2015.

[20] D. Li, L. Li, D. Kim, T. F. Bissyandé, D. Lo, and Y. L. Traon, "Watch out for this commit! a study of influential software changes," *arXiv preprint arXiv:1606.03266*, 2016.

[21] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *TSE*, vol. 38, no. 6, pp. 1276–1304, 2012.

[22] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. of CCS'07*, 2007.

[23] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *TSE*, vol. 37, no. 6, pp. 772–787, 2011.

[24] H. Tang, T. Lan, D. Hao, and L. Zhang, "Enhancing defect prediction with static defect analysis," in *Proc. of INTERNETWARE'15*, 2015.

[25] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry, "Are these bugs really normal?" in *Proc. of MSR'15*, 2015.