A Novel Approach for Comparing Automated Vulnerability Detection Techniques

Emanuele Iannone[1], Giulia Sellitto[1], Valeria Pontillo[1], Filomena Ferrucci[1] and Fabio Palomba[1]

[1]Software Engineering (SeSa) Lab – University of Salerno, Fisciano, 84084, Italy

Abstract

Finding vulnerabilities in applications' source code is a paramount task to guarantee the security of software systems. With the multitude and variety of available tools for vulnerability detection, practitioners are puzzled when selecting the most suitable approach to adopt in the projects they work on. Unfortunately, the tools cannot be easily compared as they belong to inherently different categories, i.e., static analysis, dynamic analysis, and machine learning, which makes it hard to understand the adequate solution for a project. We recognize the lack of a detailed comparison among existing approaches for vulnerability detection and propose a new way to compare the tools with each other. Specifically, we envision a "tournament-like" comparison of the best tools in different categories under several aspects, such as the number of vulnerabilities detected, execution time elapsed, and false alarms raised. By performing a multivocal literature review, we plan to gather information about the existing tools for vulnerability detection and the related benchmarks; then, we design a comparison study among the reproducible tools to determine the best solution under different scenarios. We expect our contribution to be impactful for practitioners struggling with selecting a tool to employ among the many available options.

Keywords

Software Security, Software Vulnerability, Vulnerability Detection, Automated Tool

1. Introduction

Software is becoming more and more critical for performing basic daily tasks. Software companies are committed to developing high-quality products that can reliably and efficiently fulfill all requirements. However, the sole assessment of functional aspects is not enough [1, 2]; non-functional requirements must also be considered to ensure the creation of high-quality software [3]. Over the last decades, several models for measuring and monitoring such non-functional aspects have been defined [4, 5]. In particular, security, i.e., the software's ability to withstand malicious attacks [6], has drawn great attention from researchers and practitioners. The building of highly-secure applications often translates into detecting and removing as many security issues as possible to minimize the risk of being attacked by malicious users. Such issues, a.k.a. software vulnerabilities, appear in the source code due to improper design decisions, e.g., sending sensitive data over an insecure channel,[1] or overlooked programming mistakes, e.g., using an inadequate sanitization of user-supplied inputs when querying an SQL database.[2] Vulnerabilities represent entry points that malicious attackers can exploit to reach their goals, such as denying the correct operation of the system or stealing the data it manages.

Finding as many vulnerabilities as possible before deployment is crucial to minimize the risk of exploitation during software evolution. The task of deciding whether a given program contains security issues or not is called *vulnerability detection* (a.k.a. *discovery* or *identification*), and it is far from being trivial: it has been demonstrated to be undecidable [7, 8]. This means that a "perfect" method that properly detects all the vulnerabilities in the code and marks as non-vulnerable all the safe components can never exist [9, 10]. Thus, one can only propose partial or approximate solutions to such a problem. Currently, many approaches and automated tools are available in the state of the art and practice, each with its peculiarities, strengths, and weaknesses [11, 12, 13].

Although the availability of several automated vulnerability detection tools is a valuable advantage for practitioners, it might pose a serious threat. Namely, developers might opt for the most-adopted solutions, resulting in sub-optimal choices as the selected tools might not fit the specific project needs and constraints. For instance, in web-based applications with a short release cycle, developers should opt for tools that can be executed rapidly and continuously, disregarding tools that make through-

SATToSE'23: 15th Seminar Series on Advanced Techniques & Tools for Software Evolution, June 12–14, 2023, Fisciano, Italy

eiannone@unisa.it (E. Iannone); gisellitto@unisa.it (G. Sellitto); vpontillo@unisa.it (V. Pontillo); fferrucci@unisa.it (F. Ferrucci); fpalomba@unisa.it (F. Palomba)

https://emaiannone.github.io/ (E. Iannone); https://giuliasellitto7.github.io/ (G. Sellitto); https://valeriapontillo.github.io/ (V. Pontillo); https://docenti.unisa.it/001775/en/home (F. Ferrucci); https://fpalomba.github.io/ (F. Palomba)

0000-0001-7489-9969 (E. Iannone); 0000-0002-5491-0873 (G. Sellitto); 0000-0001-6012-9947 (V. Pontillo); 0000-0002-0975-8972 (F. Ferrucci); 0000-0001-9337-5116 (F. Palomba)

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

[1]CWE-311: https://cwe.mitre.org/data/definitions/311.html

[2]CWE-89: https://cwe.mitre.org/data/definitions/89.html

out scans. We hypothesize that this scenario comes true due to the absence of a detailed comparison of all the currently-available techniques for vulnerability detection, particularly when their type of analyses, i.e., *static*, *dynamic* and *machine-learning-based*, is considered. Indeed, if developers were aware of the existence of *benchmarks*, they could be better supported in the tool selection. Such benchmarks would represent a catalog highlighting their peculiarities supported by empirical evidence. However, to the best of our knowledge, such empirical evidence does not exist yet.

We aim to fill this gap by comparing existing automated vulnerability detection techniques to assess their effectiveness and differences from a practical standpoint. We envision two main phases in our study, i.e., (1) a Multivocal Literature Review for collecting the available tools and benchmarks and (2) a comparison study performed in a *tournament-like* fashion. Namely, we plan to compare the tools within their belonging category (i.e., static analyzers, dynamic analyzers, and machine-learning-based prediction models) first and then compare the "winners" of each category with each other. We envision that our work will provide impactful knowledge to both researchers and practitioners, as the former can get empirical evidence of the characteristics of each tool, and the latter can leverage the results to properly choose the best solution to employ in their real-case scenarios.

In the following, we first summarize the relevant literature on automated vulnerability detection; afterward, we present our research method idea and then we conclude by reflecting on the expected impact of this work.

2. Background & Related Work

2.1. Automated Vulnerability Detection

Software security researchers and practitioners have proposed many automated solutions to detect vulnerabilities in the code [11, 12, 13].

Most of them rely on static code analysis to spot recurring weak code patterns [14] or detect unusual data flows via taint analysis [15] without executing the code under analysis. Some notable examples are Fortify [16], Flawfinder [17], and FlowDroid [18]. These tools perform particularly well at finding specific vulnerabilities, such as buffer overflow or SQL injection flaws, while struggling for others, like those related to authentication mechanisms and dynamic access control. Static analyzers are known to raise many warnings, most of which are false alarms, having a negative impact on the inspection workload required to developers [19].

Another relevant portion of solutions leverages concrete executions of the system under analysis—i.e., dynamic analysis—to identify unusual behaviors that hint at the presence of vulnerabilities. For instance, OWASP ZAP [20] stimulates the tested system with specially-crafted inputs and interprets the response to assess the presence of a vulnerability. American Fuzzy Lop [21] seeds the system under test with random inputs that are continuously mutated to maximize the chance of detecting crashes, buffer overflows, or other anomalies—such a technique is known as *fuzzing* [22]. Fuzzing tools require a corpus of seed inputs provided by the analysts themselves; this drawback has been addressed by DART [23], which is entirely automated and does not need any starting set of inputs. Yet, the execution time of dynamic analyzers represents their most serious disadvantage, making the tools much more difficult to employ in IDEs or CI pipelines—though initiatives are aimed at simplifying this process, such as OSS-Fuzz [24]. Despite having a lower chance of raising false positives, they still struggle to uncover vulnerabilities that demand a very restricted set of inputs to be observed.

Hybrid vulnerability detection solutions leverage static and dynamic analysis to take advantage of their benefits and mitigate their drawbacks. This is the case of *symbolic execution* [25, 26]: rather than supplying the target program with real input values, symbolic execution tools execute the code—with an interpreter—with "symbolic inputs", representing multiple possible inputs at once. The objective is to assign each execution path a constraint expressed on the input values so that its resolution will generate an input that will cover that execution path. CUTE [27], KLEE [28], and FuzzBall [29] are some of the most popular tools implementing symbolic execution.

More recently, machine learning has been employed to discover software vulnerabilities affecting files or functions [30, 31], though no solutions have been adopted outside academic or industrial research. Relevant proposals available in the literature are μVulDeePecker [32], VCCFinder [33], and TROVON [34, 35]. Similarly to static analysis approaches, vulnerability prediction models (VPMs) still report many false positives, likely caused by poor diffuseness of vulnerable components compared to non-vulnerable ones.

2.2. Survey & Comparison Studies

Shahriar and Zulkernine [11] surveyed many vulnerability detection approaches available at the time of the study (2011), leveraging a wide range of techniques pertaining to static, dynamic, and hybrid analysis. The paper compared dozens of techniques by analyzing the content of the papers where they are described without providing a benchmark—i.e., executing them on a dataset to observe their performance. The approaches were compared under several aspects, like their core technique, the granularity of analysis, the targeted vulnerability type, and the supported languages. The work also reviews exist-

ing secure programming mitigation strategies to remove vulnerabilities from the source code. Despite this, the paper does not describe how the surveyed techniques were selected in detail, i.e., the adopted selection method seems not systematic.

Ghaffarian and Shahriari [13] built upon the described review by adding machine-learning-based approaches in the loop, which have been largely adopted in for detecting software vulnerabilities since the last decade. The authors identified four main types of ML-based vulnerability detection techniques: (i) supervised prediction models leveraging features derived from traditional software metrics, e.g., function's complexity or file size; (ii) prediction models leveraging features extracted automatically from source code, e.g., text or graph representations; (iii) unsupervised anomaly detection techniques, e.g., association rule mining and nearest neighbors; and (iv) miscellaneous approaches, such as classifiers trained on the output of taint analysis tools. Analogously to the work by Shahriar and Zulkernine [11], there is no indication of how the surveyed techniques were selected.

Lin et al. [36] surveyed the literature to find approaches adopting deep-learning-based solutions to detect software vulnerabilities with the goal of understanding the extent to which such models can capture the underlying semantics of vulnerabilities. Specifically, the authors categorized the previous work according to the feature representation adopted, i.e., (i) graph-based representations rely on Abstract Syntax Trees (ASTs) and/or Program Dependence Graphs (PDGs), etc.; (ii) sequence-based representations rely on structures like stack traces, call chains, or data flows; (iii) text-based representations rely directly on the code text; and (iv) mixed representations that combine different representations.

Zaazaa and Bakkali [25] focused on dynamic analysis approaches alone, describing the advantages and disadvantages of the most popular techniques, like KLEE [28], TaintScope [37] and AFLFast [38], belonging to three (possibly overlapping) classes: (i) dynamic taint analyzers, (ii) fuzzers, and (iii) symbolic execution engines. The study also reports an overview of the performance scored by a subset of the surveyed approaches based on the evaluation already available in the literature. For instance, KLEE [28], a dynamic symbolic execution engine, evaluated on 90 programs in GNU Coreutils managed to generate test cases covering an average of 82% statements. On the same set of programs, the AFLFast [38] fuzzer discovered much more vulnerabilities than KLEE at the price of longer running time. The AEG (Automatic Exploit Generation) tool [39] discovered 16 control flow hijacking vulnerabilities—two of which were never discovered before—in 14 open-source C/C++ projects.

From an empirical perspective, Austin et al. [40] compared four vulnerability discovery technique types based on manual and automated analysis. They conducted three case studies on three Electronic Health Record (EHR) systems evaluating the effectiveness of (i) exploratory manual penetration testing, (ii) systematic manual penetration testing, (iii) automated penetration testing, and (iv) automated static analysis. Exploratory manual penetration testing involves exploring the application without any precise schema and aided by tools like debuggers or network traffic monitors. The systematic manual penetration testing, proposed by Smith [41], selects and generates test cases starting from functional requirements. Automated penetration testing relies on a tool—like IBM Rational AppScan—that crawls the web application and tries several pre-made tests to uncover vulnerabilities. Automated static analysis—in this case, Fortify 360—consists in running a traditional static analyzer. They discovered that the evaluated techniques have variable performance. Static analysis has the highest discovery rate—at the cost of a large number of false positives, i.e., false alarms—while all the other techniques detect way fewer vulnerabilities, most of which pertain to injection-type vulnerabilities, e.g., SQL Injection and Cross-site Scripting (XSS). The techniques have some orthogonality, i.e., one technique is able to discover vulnerabilities that other approaches miss.

Arusoaide et al. [14] went deeper into investigating the performance of static analyzing in C/C++ code. They run more than 10 static analyzers, including popular tools like Flawfinder [17] and Cppcheck [42], on the Toyota ITC dataset, containing 639 synthetic C/C++ pairs of snippets, one affected by a security bug and the other without it. The study reports a similar outcome to Austin et al. [40]: different tools perform differently depending on the vulnerability type. The only exception stands in the static analyzer integrated with Clang[3], which found many vulnerabilities at the cost of less precision.

Antunes and Viera [43] benchmarked eight automated vulnerability detection tools, namely four dynamic analyzers, three static analyzers, and one anomaly detector, for discovering SQL Injection vulnerabilities in SOAP-based web services application. The anomaly detector and two static analyzers—i.e., FindBugs [44] and Yasca[4]—resulted in having the best F-measures. Due to the mediocre results scored by the dynamic analyzers, they were reassessed on another application, obtaining analogous results for all four tools.

Framing the current research with our goal. We observe that the current research focused on describing the various approaches but only a minority of them compared the effectiveness of existing tools on realistic benchmarks. Despite this, the comparison studies are restricted to analyzing a single class of techniques and vulnerabilities. Our work addresses such a knowl-

[3] Clang: https://clang.llvm.org/
[4] Yasca: https://github.com/scovetta/yasca

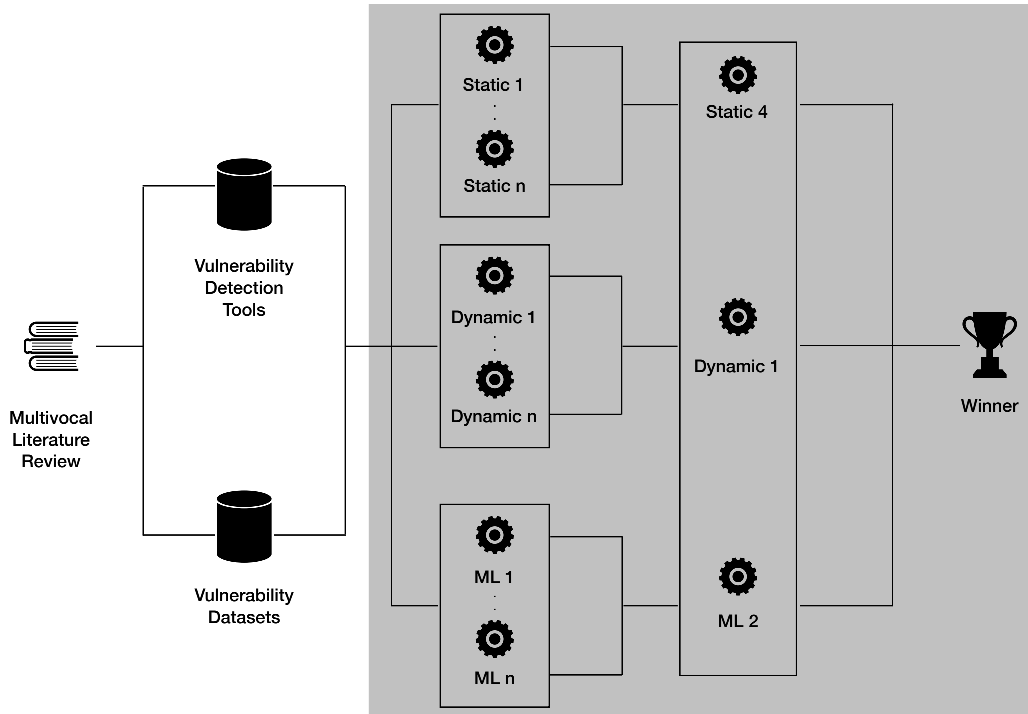

Figure 1: Graphical description of our research method proposal. The gray box represents an iteration of the tournament-like comparison for a given benchmark dataset.

edge gap by performing a comprehensive comparison of existing tools for automated vulnerability detection, evaluated in a tournament-like fashion, under multiple publicly-available benchmarks.

3. Research Method

3.1. Research Statement

The *goal* of this study is to perform a detailed comparison of existing automated vulnerability detection approaches, with the *purpose* of providing developers with empirical evidence on the suitability of such solutions for their specific real-case scenarios. The *perspective* is of both researchers and practitioners; the former are interested in leveraging comprehensive and sound work to assess the characteristics of the solutions proposed, while the latter are concerned about the availability of a catalog of approaches evaluated and compared among each other, to select the most suitable in their use case. The *context* of our study consists of a set of automated tools for vulnerability detection and a set of evaluation benchmarks that have been employed in the existing literature. In particular, we aim to compare the reproducible tools to assess their performance on reliable benchmarks. First, we need to find the range of available tools; hence we ask:

> **RQ_1.** *What are the existing automated tools for vulnerability detection?*

By answering this research question, we aim to collect a comprehensive set of the available solutions that we plan to evaluate and compare on common ground. Therefore, we need reliable datasets and benchmarks where our experiments are performed, and we ask:

> **RQ_2.** *What are the datasets and benchmarks used to evaluate vulnerability detection tools?*

The first two research questions are meant to gather the experimental objects with which we can answer the third research question:

> **RQ_3.** *How do vulnerability detection tools perform under different benchmarks in terms of detection rates, analysis rates, and output understandability?*

The following section describes our plan for answering the research questions formulated.

3.2. Study Design

The research method we design involves two main steps, depicted in Figure 1. We first plan to perform a **Multivocal Literature Review** (using the guidelines provided by Garousi et al. [45]) with the double goal of answering $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$. On the one hand, we will collect all automated solutions from literature and online resources, with particular attention to static, dynamic, and machine-learning-based detectors. On the other hand, we will gather resources reporting datasets and benchmarks containing projects affected by recognized and mapped vulnerabilities, e.g., the Vulnerability History Project.[5] We plan to carry out the search on ACM Digital Library,[6] IEEE Xplore,[7] and Scopus,[8] for the formal literature, and Google search engine to search for the grey literature and other online resources. As for the search query, we propose the following:

Search Query

((“automat*” *OR* “tool” *OR* “techinique”) *AND* (“vulnerabilit*” *OR* (“security*” *AND* (“flaw” *OR* “bug” *OR* “issue”))) *AND* (“detect*” *OR* “analyze*” *OR* “analyse*” *OR* “identificat*”)

Once we gain information about tools and benchmarks that have been developed and released, we will select only those that are publicly accessible. From the tools perspective, we select only the tools that can be replicated without blocking issues. Namely, we consider only the tools with sufficiently detailed documentation (a README file can be enough) that explain how to install and run the tool. Then, we test if the tool functions correctly on toy projects to observe how it behaves and whether it raises runtime errors. In case of complex issues, we will contact the corresponding contact (if any) to help us solve the issue. Should the problems not be addressed, the tool is discarded from the study. From the benchmark perspective, we expect them to be collections of projects with labeled data, i.e., information about instances of vulnerabilities or weaknesses affecting the code. We assess the reliability of such data by analyzing their origin and how they have been labeled and validated. The first output of the study will be a catalog of tools organized by category, a collection of projects with labeled vulnerabilities, and a set of benchmarks to evaluate detection strategies.

Afterward, we compare the performance of the selected tools in a *tournament-like* comparison, which represents one of the main contributions of this work and will be fundamental for answering $\mathbf{RQ}_3$ adequately. Specifically, the tools are first compared within their belonging category, e.g., static analyzers are compared with other static analyzers to avoid unfair comparisons. For instance, comparing a static analyzer with a dynamic analyzer might penalize the former as they are generally less precise than the latter due to their nature. The comparisons consist of executing the tools on a given benchmark to assess their performance on a common set of projects and vulnerabilities. This means that for each N benchmark, there will be N reiterations of the tournament. Finally, the winners of each of the three categories—i.e., static analyzers, dynamic analyzers, and machine-learning models—will be compared in a “final” bout that will decree the winner on that benchmark. We point out that assessing the best one on all fronts might be difficult since the tools can be evaluated under many different perspectives, i.e., with diverse *performance metrics*. Hence, we plan to involve a wide range of metrics commonly adopted in literature for evaluating these kinds of tools, measuring aspects like (i) the discovery rate of vulnerabilities, (ii) the rate of analyzed components (e.g., code elements) per time unit, and (iii) the understandability of the output. Other than comparing the tools under each metric individually, we also plan to find the best trade-off among a combination of multiple metrics.

Other than declaring the winners of each category from the metrics perspective, we also analyze the complementarity of the tools, i.e., how the findings of one can integrate the findings of another. Namely, we aim to identify those cases where all the tools, or the tools belonging to a specific category, fail at recognizing certain vulnerabilities Such an analysis will be helpful in understanding the recurring patterns that lead the tools to fail. Symmetrically, we aim to spot those cases where the tools find no trouble detecting certain vulnerabilities, reflect on why this happened, and elaborate on the quality of currently available datasets and benchmarks.

4. Final Remarks

We presented our research plan to provide a detailed comparison of the currently-available techniques for vulnerability detection belonging to different categories. We expect this work can help researchers and practitioners better understand security tools’ real performance. Our research lays the foundations for an automated recommendation tool that supports developers in selecting the best suite of vulnerability detectors optimized for the projects’ specific needs. Besides, our work can encourage the development of next-generation vulnerability detectors that employ a smart combination of existing approaches, in hope of providing more accurate and actionable vulnerability reports.

[5] Vulnerability History Project: https://github.com/orgs/VulnerabilityHistoryProject
[6] ACM Digital Library: https://dl.acm.org
[7] IEEE Xplore: https://ieeexplore.ieee.org/Xplore/home.jsp
[8] Scopus: https://www.scopus.com/home.uri

References

[1] M. Glinz, On non-functional requirements, in: 15th IEEE international requirements engineering conference (RE 2007), IEEE, 2007, pp. 21–26.

[2] L. Chung, B. A. Nixon, E. Yu, J. Mylopoulos, Non-functional requirements in software engineering, volume 5, Springer Science & Business Media, 2012.

[3] R. Berntsson Svensson, T. Gorschek, B. Regnell, Quality requirements in practice: An interview study in requirements engineering for embedded systems, in: International Working Conference on Requirements Engineering: Foundation for Software Quality, Springer, 2009, pp. 218–232.

[4] ISO Standard 8402: Quality management and quality assurance, Standard, International Organization for Standardization, Geneve, CH, 1994.

[5] J. Bansiya, C. G. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Transactions on software engineering 28 (2002) 4–17.

[6] G. McGraw, Software security, IEEE Security & Privacy 2 (2004) 80–83.

[7] W. Landi, Undecidability of static analysis, ACM Lett. Program. Lang. Syst. 1 (1992) 323–337. URL: https://doi.org/10.1145/161494.161501. doi:10.1145/161494.161501.

[8] T. Reps, Undecidability of context-sensitive data-dependence analysis, ACM Trans. Program. Lang. Syst. 22 (2000) 162–186. URL: https://doi.org/10.1145/345099.345137. doi:10.1145/345099.345137.

[9] Y. Xie, M. Naik, B. Hackett, A. Aiken, Soundness and its role in bug detection systems, in: Proc. of the Workshop on the Evaluation of Software Defect Detection Tools, volume 7, 2005.

[10] R. Jhala, R. Majumdar, Software model checking, ACM Comput. Surv. 41 (2009). URL: https://doi.org/10.1145/1592434.1592438. doi:10.1145/1592434.1592438.

[11] H. Shahriar, M. Zulkernine, Mitigating program security vulnerabilities: Approaches and challenges, ACM Comput. Surv. 44 (2012). URL: https://doi.org/10.1145/2187671.2187673. doi:10.1145/2187671.2187673.

[12] A. Kaur, R. Nayyar, A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code, Procedia Computer Science 171 (2020) 2023–2029. URL: https://www.sciencedirect.com/science/article/pii/S1877050920312023. doi:https://doi.org/10.1016/j.procs.2020.04.217.

[13] S. M. Ghaffarian, H. R. Shahriari, Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey, ACM Comput. Surv. 50 (2017). URL: https://doi.org/10.1145/3092566. doi:10.1145/3092566.

[14] A. Arusoaie, S. Ciobâca, V. Craciun, D. Gavrilut, D. Lucanu, A comparison of open-source static analysis tools for vulnerability detection in c/c++ code, in: 2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2017, pp. 161–168. doi:10.1109/synasc.2017.00035.

[15] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Association for Computing Machinery, New York, NY, USA, 2014, p. 259–269. URL: https://doi.org/10.1145/2594291.2594299. doi:10.1145/2594291.2594299.

[16] M. F. CyberRes, Fortify static code analyzer, 2023. URL: https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer.

[17] D. A. Wheeler, Flawfinder, 2023. URL: https://dwheeler.com/flawfinder/.

[18] S. Arzt, Flowdroid, 2023. URL: https://github.com/secure-software-engineering/FlowDroid.

[19] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, A. Zaidman, How developers engage with static analysis tools in different contexts, Empirical Software Engineering 25 (2020) 1419–1457. URL: https://doi.org/10.1007/s10664-019-09750-5. doi:10.1007/s10664-019-09750-5.

[20] Owasp, Zed attack proxy, 2023. URL: https://www.zaproxy.org/.

[21] M. Zalewski, American fuzzy lop, 2023. URL: https://lcamtuf.coredump.cx/afl/.

[22] H. Liang, X. Pei, X. Jia, W. Shen, J. Zhang, Fuzzing: State of the art, IEEE Transactions on Reliability 67 (2018) 1199–1218. doi:10.1109/tr.2018.2834476.

[23] P. Godefroid, N. Klarlund, K. Sen, Dart: Directed automated random testing, in: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005, pp. 213–223.

[24] Google, Oss-fuzz, 2023. URL: https://google.github.io/oss-fuzz/.

[25] O. Zaazaa, H. El Bakkali, Dynamic vulnerability detection approaches and tools: State of the art, in: 2020 Fourth International Conference On Intelligent Computing in Data Sciences (ICDS), 2020, pp. 1–6. doi:10.1109/ICDS50568.2020.9268686.

[26] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, I. Finocchi, A survey of symbolic execution techniques, ACM Comput. Surv. 51 (2018). URL: https://

doi.org/10.1145/3182657. doi:10.1145/3182657.
[27] K. Sen, D. Marinov, G. Agha, Cute: A concolic unit testing engine for c, in: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, Association for Computing Machinery, New York, NY, USA, 2005, p. 263–272. URL: https://doi.org/10.1145/1081706.1081750. doi:10.1145/1081706.1081750.
[28] C. Cadar, D. Dunbar, D. Engler, Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, USENIX Association, USA, 2008, p. 209–224.
[29] D. Babić, L. Martignoni, S. McCamant, D. Song, Statically-directed dynamic automated test generation, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 12–22. URL: https://doi.org/10.1145/2001420.2001423. doi:10.1145/2001420.2001423.
[30] S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller, Predicting vulnerable software components, in: Conference on Computer and Communications Security, 2007, pp. 529–540.
[31] T. Zimmermann, N. Nagappan, L. Williams, Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista, in: International Conference on Software Testing, Verification and Validation, 2010, pp. 421–428.
[32] D. Zou, S. Wang, S. Xu, Z. Li, H. Jin, μvuldeepecker: A deep learning-based system for multiclass vulnerability detection, IEEE Transactions on Dependable and Secure Computing 18 (2021) 2224–2236. doi:10.1109/TDSC.2019.2942930.
[33] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, Y. Acar, Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, Association for Computing Machinery, New York, NY, USA, 2015, p. 426–437. URL: https://doi.org/10.1145/2810103.2813604. doi:10.1145/2810103.2813604.
[34] A. Garg, R. Degiovanni, M. Jimenez, M. Cordy, M. Papadakis, Y. L. Traon, Learning to predict vulnerabilities from vulnerability-fixes: A machine translation approach, 2020. arXiv:2012.11701.
[35] A. Garg, R. Degiovanni, M. Jimenez, M. Cordy, M. Papadakis, Y. Le Traon, Learning from what we know: How to perform vulnerability prediction using noisy historical data, Empirical Software Engineering 27 (2022) 1–30.
[36] G. Lin, S. Wen, Q.-L. Han, J. Zhang, Y. Xiang, Software vulnerability detection using deep neural networks: A survey, Proceedings of the IEEE 108 (2020) 1825–1848. doi:10.1109/JPROC.2020.2993293.
[37] T. Wang, T. Wei, G. Gu, W. Zou, Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection, in: 2010 IEEE Symposium on Security and Privacy, 2010, pp. 497–512. doi:10.1109/SP.2010.37.
[38] M. Böhme, V.-T. Pham, A. Roychoudhury, Coverage-based greybox fuzzing as markov chain, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 1032–1043. URL: https://doi.org/10.1145/2976749.2978428. doi:10.1145/2976749.2978428.
[39] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, D. Brumley, Automatic exploit generation, Commun. ACM 57 (2014) 74–84. URL: https://doi.org/10.1145/2560217.2560219. doi:10.1145/2560217.2560219.
[40] A. Austin, C. Holmgreen, L. Williams, A comparison of the efficiency and effectiveness of vulnerability discovery techniques, Information and Software Technology 55 (2013) 1279–1288. URL: https://www.sciencedirect.com/science/article/pii/S0950584912002339. doi:https://doi.org/10.1016/j.infsof.2012.11.007.
[41] B. Smith, Systematizing security test case planning using functional requirements phrases, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 1136–1137. URL: https://doi.org/10.1145/1985793.1986019. doi:10.1145/1985793.1986019.
[42] Cppcheck, Cppcheck – a tool for static c/c++ code analysis, 2023. URL: https://cppcheck.sourceforge.io/devinfo/.
[43] N. Antunes, M. Vieira, Assessing and comparing vulnerability detection tools for web services: Benchmarking approach and examples, IEEE Transactions on Services Computing 8 (2015) 269–283. doi:10.1109/TSC.2014.2310221.
[44] P. Arteau, Find security bugs, 2023. URL: https://find-sec-bugs.github.io/.
[45] V. Garousi, M. Felderer, M. V. Mäntylä, Guidelines for including grey literature and conducting multivocal literature reviews in software engineering, Information and software technology 106 (2019) 101–121.