Toward Automated Exploit Generation for Known Vulnerabilities in Open-Source Libraries

Emanuele Iannone[1], Dario Di Nucci[2], Antonino Sabetta[3], Andrea De Lucia[1]
[1]SeSa Lab - University of Salerno, Fisciano, Italy
[2]Tilburg University, JADS, 's-Hertogenbosch, The Netherlands
[3]SAP Security Research, France
eiannone@unisa.it, d.dinucci@uvt.nl, antonino.sabetta@sap.com, adelucia@unisa.it

***Abstract*—Modern software applications, including commercial ones, extensively use Open-Source Software (OSS) components, accounting for 90% of software products on the market. This has serious security implications, mainly because developers rely on non-updated versions of libraries affected by software vulnerabilities. Several tools have been developed to help developers detect these vulnerable libraries and assess and mitigate their impact. The most advanced tools apply sophisticated reachability analyses to achieve high accuracy; however, they need additional data (in particular, concrete execution traces, such as those obtained by running a test suite) that is not always readily available.**

In this work, we propose SIEGE, a novel automatic exploit generation approach based on genetic algorithms, which generates test cases that execute the methods in a library known to contain a vulnerability. These test cases represent precious, concrete evidence that the vulnerable code can indeed be reached; they are also useful for security researchers to better understand how the vulnerability could be exploited in practice. This technique has been implemented as an extension of EVOSUITE and applied on set of 11 vulnerabilities exhibited by widely used OSS JAVA libraries. Our initial findings show promising results that deserve to be assessed further in larger-scale empirical studies.

***Index Terms*—Exploit Generation, Security Testing, Software Vulnerabilities.**

I. INTRODUCTION

The adoption of software reuse, particularly of *third-party libraries* released under open-source licenses, has dramatically increased over the past two decades and has become pervasive in today's software, including commercial products. Recent analyses [1] estimate that over 90% of software products on the market include some form of OSS components. Like any other piece of software, third-party libraries may contain flaws [2], [3], whose negative effects are amplified by the fact that they occur in components that are broadly adopted [4], [5]. The complexity in the dependency structures of modern software systems makes things worse: the impact of the defects occurring deep in the dependency graph is difficult to assess [6] and to mitigate [7]. One of the primary forms of defect that regularly affect third-party libraries are *vulnerabilities* [8], which expose the software to potential attacks against its confidentiality, integrity, and availability (CIA) [9]. For these reasons, *third-party vulnerabilities* represent the main threat caused by inadequate dependency management practices [4] since they expose client applications (directly, or *transitively* through potentially long dependency chains) to abuse, as happened for the infamous HEARTBLEED bug. In that case, a "naive" vulnerability in OPENSSL 1.0.1 exposed almost half-million websites (17% of the total at the time), supposedly protected through SSL, to *buffer over-read* attacks [10]. As time goes by, more and more vulnerabilities of popular OSS libraries are being discovered [8] and publicly disclosed in vulnerability databases, among which the de-facto standard *National Vulnerability Database* (NVD) [11], where vulnerabilities are documented according to the *Common Vulnerabilities and Exposures* (CVE) standard. This growing trend motivated the inclusion of "Using components with known vulnerabilities" into the *OWASP Top 10 Web Application Security Risks* [12] in 2013. As of today, that risk is still in the OWASP top-ten.

Numerous detection and assessment tools have been developed to tackle this problem [13]–[17]. Almost all of them analyze a project searching for known vulnerable OSS dependencies. Whenever a vulnerable dependency is found, the common mitigation action consists in updating it to another non-vulnerable version. While this solution seems reasonable and easy to adopt, it can be difficult to implement in practice, particularly when the library to be updated is not a direct dependency but a transitive one, or when the affected system is operational in a productive environment and serves business-critical functions [3], [18]. Other tools have tackled this problem by providing fine-grained code analyses to reduce the number of false alerts (i.e., dependencies flagged as vulnerable but that do not expose the client application to any threat) [16], [19], [20] in an effort to prioritize library updates. In this regard, tools such as ECLIPSE STEADY provide a combination of both static (i.e., call graph-based) and dynamic analyses (i.e., test-based) to maximize the reachability of known vulnerable library constructs (e.g., method, class) starting from the client application code. In particular, the dynamic reachability analysis requires a significant amount of data from the client application test suite (i.e., execution traces) to make an effective vulnerability assessment. Unfortunately, many software projects are not adequately tested [21]. Furthermore, the test cases that an attacker would try to trigger to exploit vulnerabilities are inherently different from those needed for functional testing. Indeed, attackers would try to explore *corner cases* and *unusual* execution conditions.

Novelty. In this work, we propose **SIEGE** (*Search-based*

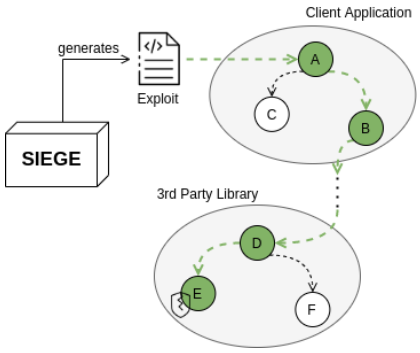

Fig. 1. SIEGE exploits functioning. A generated exploit starts its execution on the client class A, which in turn it calls all the classes along the way to the vulnerable class E (marked with a broken shield) belonging to a vulnerable third-party library (included in the application's classpath). The black dots between the two codebases indicates the (possible) presence of additional third-party libraries.

automatIc Exploit GenEration), a novel vulnerability assessment technique for the automatic generation of program executions of client applications to reach the vulnerable code fragments contained in a dependency. Such program executions provide the essence of *exploits* since they can reach and execute the vulnerable library constructs without the need of pre-existing test suites, thus solving the limitations of the previously mentioned tools. It is important to stress that the test cases generated by SIEGE have the only goal to execute the constructs affected by the vulnerability, not to cause damage (e.g., spawning a shell [22] or going out of the intended memory bounds [23]). Instead, they are meant to support security analysts and penetration testers, who may use the test cases generated by SIEGE to speed up the design and implementation of full exploits (in a similar fashion to [24]). In the following, for the sake of conciseness, we will call the test cases generated by SIEGE "exploits". Figure 1 shows the general principle of a SIEGE exploit in a typical usage scenario.

This work makes the following contributions: 1) SIEGE, an automatic exploit generation technique that could be integrated into existing library vulnerability assessment tools; 2) a preliminary study on 11 known vulnerabilities exhibited by widely used JAVA OSS libraries; 3) experimental raw data and SIEGE source code [25].

II. Related Work

Vulnerable code reachability. Several free (e.g., OWASP DEPENDENCY CHECK a.k.a., OWASP DC [13]) and commercial tools (e.g., WHITESOURCE [14], SNYK [15]) are available to detect vulnerabilities in open-source components. However, as shown in the study that Zapata et al. [20] conducted in the NODE.JS ecosystem, up to 73.3% of the projects depending on vulnerable libraries are actually safe. Thus, analyzing the impact of library vulnerabilities based on *inclusion* analysis leads to overestimations (i.e., high number of false alerts), and more accurate approaches are necessary to address the problem at a finer level of granularity, analyzing which *methods* of each dependency are actually reachable. Such an approach was adopted by Plate et al. [26], who first proposed the use of runtime monitoring (i.e., dynamic reachability analysis) to establish whether vulnerable methods are executed and made a first step beyond mere inclusion detection. This work was later extended [16] introducing a combination of dynamic and static reachability analyses. This enhancement could reach methods that static reachability analysis and runtime monitoring in isolation would not have detected. Their approach has been implemented in ECLIPSE STEADY [27], an open-source tool that SAP uses internally to scan its JAVA-based products. The empirical comparison between STEADY and OWASP DC (which does not use reachability analysis) showed that all STEADY findings are true positives. In contrast, 88.8% of the code-related vulnerabilities detected by OWASP DC are false positives. Simultaneously, for difficult to parse vulnerabilities (e.g., security misconfigurations) 63.3% of OWASP DC findings are true positives. To the best of our knowledge, no previous work and tool applied search-based algorithms to reach vulnerable code exhibited by libraries.

Automatic test case generation. EVOSUITE [28] is a state-of-the-art tool which leverages genetic algorithms (as well as other evolutionary approaches) for automatically selecting and generating test cases for JAVA classes. Genetic Algorithms (GA)s [29]–[31] are inspired by biological evolution. They rely on a population of candidate solutions (*individuals*) to derive next *generations* of solutions by iteratively applying *selection*, *crossover*, and *mutation* (a.k.a., search operators) with the aim to optimize a given *fitness function*. In the context of EVOSUITE, individuals are represented as JUNIT test methods, whereas search operators are applied on the list of statements they are made of [32]. The fitness function reflects a specific code coverage criterion (e.g., branch coverage) for the selected class under test (CUT). The goal of EVOSUITE is to generate both (i) minimal test suites that achieve high code coverage and (ii) an effective set of assertions to summarize the current behavior of the CUT. EVOSUITE is not meant to replace humans during testing activities but rather to ease the workload of developers when defining unit tests (i.e., it is necessary to assess the goodness of the generated tests manually).

III. SIEGE: Toward Automated Vulnerability Exploitation

SIEGE relies on a genetic algorithm to generate a set of exploits, which start from the client application's classes and reach the targeted library vulnerabilities included within the application's dependencies. Thus, SIEGE is implemented within the context of EVOSUITE, which provides all the infrastructure that our approach needs. In this preliminary implementation, SIEGE works at the *line* granularity level, i.e., it only deals with library vulnerabilities characterized by a single vulnerable source code line. As such, the GA's fitness function—to be *minimized*—rewards those individuals (JUNIT test methods) that best cover (1) the target vulnerable library method, (2) the block containing the vulnerable line, and (3) the vulnerable line itself. An individual that successfully

reaches a fitness score equals to 0 is labeled as "SIEGE exploit". Figure 2 shows the detailed workflow of SIEGE, where three distinct phases can be observed: (1) *Preprocessing*, (2) *Goal Preparation*, and (3) *GA Execution*.

(1) Preprocessing. The creation of the GA's objective is based on two constructs extracted from the client application bytecodes, i.e., (i) the global call graph—spanning across the entire client application's classpath—and (ii) the control flow graphs of each classes' methods [33]. SIEGE reuses EVOSUITE's program analyses features, as well as the bytecode instrumentation (which collects information from the individual's execution traces [33] in Phase 3).

(2) Goal Preparation. In addition to the entire client application, SIEGE needs a *description* of the specific vulnerability to target and locate the vulnerable statement. Thus, SIEGE asks for three pieces of information: (1) *Fully Qualified Name* of the target vulnerable class; (2) *Vulnerable Method Name*; (3) *Vulnerable Line Number*. These elements are obtained by a *manual inspection* of the fixing commits of publicly disclosed vulnerabilities, collected from CVE databases, by comparing the patched version with the immediately previously vulnerable snapshot, as described in the work of Plate et al. [26]. Afterward, SIEGE builds the *coverage goals*, i.e., the coverage requirements that will define the behavior of the fitness function. Such goals represent the sub-goals that an individual execution has to satisfy at best to achieve better fitness scores, increasing its chances to be improved and kept as the final solution. A SIEGE's coverage goal consists of three sub-goals: (1) *Target Call Context*, the list of method calls that start from a client method and reaches the given vulnerable method; (2) *Target Branches*, the list of branches to be evaluated `true` to reach the block containing the vulnerable line; (3) *Target Line*, the line of code number of the vulnerability. Given the coverage goal g—containing the target call context (cc), the target branches (b) and the target line (tl)—and the i-th individual's execution trace t_i, the GA fitness score of the i-th individual is computed as follows:

$$f_i(g, t_i) = \begin{cases} 3 - CS(g.cc, t_i) & \text{if } CS(g.cc, t_i) < 1 \\ 2 - \frac{size(g.b) - AL(g.cc, t_i)}{size(g.b)} & \text{if } CS(g.cc, t_i) = 1 \text{ and } AL(g.b, t_i) > 0 \\ 1 - \frac{CL(g.tl, t_i)+1}{g.tl+1} & \text{if } CS(g.cc, t_i) = 1 \text{ and } AL(g.b, t_i) = 0 \text{ and } CL(g.tl, t_i) < g.tl \\ 0 & \text{otherwise} \end{cases}$$

where (i) $CS(g.cc, t_i)$ computes the **context similarity**, i.e., the ratio of the number of method calls of $g.cc$ that t_i covered out of the total number of method calls of $g.cc$; (ii) $AL(g.b, t_i)$ computes the **approach level** [32], i.e., the number of branches that separate the last branch of $g.b$ and its closest branch that t_i covered in the vulnerable method; (iii) $CL(g.tl, t_i)$ returns the **closest line** that t_i covered compared to the target line $g.tl$.

(3) GA Execution. EVOSUITE allows the selection of different metaheuristics, search operators, and stopping conditions. The current SIEGE implementation relies on the *monotonic GA* [34] metaheuristic, which prevents the "degradation" of the best individuals across different generations, and on the following search operators:

- *Rank selection*, which creates an ordering of the individuals based on their fitness scores and selects them proportionately to their rank.
- *Single point crossover*, which combines individuals' statements by selecting a random split point to produce offsprings.
- *Uniform statement-level mutation*, which randomly mutates (inserts, deletes, or changes) a single statement in an individual by sampling from a uniform distribution.

The main stopping condition is the *time*, i.e., the evolution continues as long as there is enough budget (expressed in seconds) available. An early stop will occur if an individual minimizes the fitness score to 0. At the end of the evolution, the optimal individual is subject to a *minimization*, i.e., a procedure to reduce the length of a JUNIT test method (i.e., number of statements) but keeping the same fitness score.

IV. EXPLORATORY EVALUATION

Research Goals. We provide an exploratory evaluation to establish whether SIEGE can generate exploits for different third-party vulnerabilities.

> **RQ.** *Does SIEGE succeed in generating exploits of third-party vulnerabilities included within client applications?*

To answer this research question, we ran SIEGE on a set of client applications that include different known vulnerable dependencies (publicly disclosed and collected in NVD).

Context Selection. Being a preliminary study, we considered 11 known vulnerabilities—each pertaining to different JAVA OSS libraries—selected from the dataset of Ponta et al. [35]. The dataset contains a total of 624 publicly disclosed vulnerabilities mapped onto the corresponding 1,282 fixing commits, i.e., the commit that patched the security flaw. The selected vulnerabilities are those whose fixing commits consist of a single line only (excluding aesthetic changes) so that we could provide SIEGE with the needed vulnerability descriptions (as explained in *Goal Preparation* phase in Section III). For the sake of this preliminary investigation, we developed a set of 11 artificial client applications, each consisting of a JAVA class that calls a single public library method. The choice of which method to call was based on the *manual inspection* conducted during the *Goal Preparation* phase to identify an appropriate starting point for reaching the target vulnerability. The source code, along with the list of the 11 vulnerabilities, is available in our online appendix [25].

Results. When launched with 5 seconds of search budget, SIEGE could generate exploits for 6 out of 11 vulnerabilities successfully (54.55%). Listing 1 shows the generated exploit

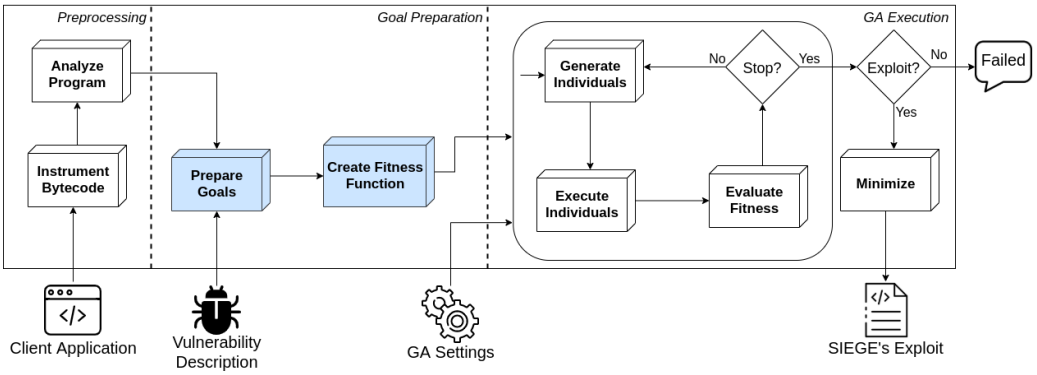

Fig. 2. The detailed workflow of SIEGE. The white blocks represent the reused EvoSuite components, while the blue ones are managed by SIEGE.

```
public void test0() throws Throwable {
  CallingClient1 callingClient1_0 = new CallingClient1();
  BasicHttpRequest basicHttpRequest0 = new
      ↪ BasicHttpRequest("", "");
  BasicHttpContext basicHttpContext0 = new
      ↪ BasicHttpContext((HttpContext) null);
  callingClient1_0.call(basicHttpRequest0,
      ↪ basicHttpContext0);
}
```

Listing 1. Exploit for the vulnerability affecting HttpComponents Client 4.1.

for the vulnerability affecting HttpComponents Client 4.1 library. Note that SIEGE may produce different results due to the random behavior of the GA, i.e., the actual code of the exploits and the number of elapsed generations may be different across runs. The remainder of the results of this run is available in our online appendix [25].

Unfortunately, SIEGE failed at generating exploits for the vulnerabilities affecting Commons Compress, Jenkins, Nifi, Mailer Plugin and Primefaces. We further investigated the reasons behind this behavior by increasing the search budget to 15, 30, and 60 seconds. The results of these new runs are reported in Table I. With a higher budget, SIEGE can exploit an additional vulnerability successfully, namely the one affecting Commons Compress, but still failing with the other four (so, reaching 63.64% coverage). In particular, we can see that SIEGE never went lower than 3.0 fitness score for Jenkins, Nifi, and Mailer Plugin. In other words, the candidate exploits never managed to call the vulnerable class at all. In contrast, the candidates targeting Primefaces's vulnerability failed at calling its vulnerable method.

The reasons behind these failures may be caused by external and not considered factors, such as: (i) the **intrinsic complexity** of exploiting specific vulnerabilities (i.e., the difficulty in providing the proper input values to reach the vulnerable line); (ii) the **way** the client application uses the vulnerable dependency (i.e., the extent to which the client "guards" the vulnerability); (iii) the **GA settings** (e.g., the metaheuristics, the search operators, the stopping conditions). All these aspects need additional and dedicated investigations, which is part of our future agenda.

TABLE I
SIEGE's performance on the 11 vulnerabilities when run with different search budgets. 'Fit.' column reports the fitness score of the best individual. 'Gen.' column reports the number of elapsed generations. 'Expl.' column shows whether the exploit generation succedeed in at least one case.

Library	Version	Search Budgets (sec) 5		15		30		60		Expl.
		Fit.	Gen.	Fit.	Gen.	Fit.	Gen.	Fit.	Gen.	
Commons Compress	1.15	0.18	38	0.00	21	0.00	29	0.00	302	✔
Tomcat	7.0.12	0.00	1	0.00	1	0.00	1	0.00	1	✔
Jasypt	1.9.1	0.00	1	0.00	1	0.00	1	0.00	1	✔
Jenkins	2.89.3	3.00	53	3.00	190	3.00	397	3.00	799	✖
Multijob Plugin	1.26	0.00	1	0.00	1	0.00	1	0.00	1	✔
Commons FileUpload	1.3.1	0.00	1	0.00	1	0.00	1	0.00	1	✔
HttpComponents Client	4.1	0.00	1	0.00	1	0.00	1	0.00	1	✔
Zeppelin	0.6.0	0.00	1	0.00	1	0.00	1	0.00	1	✔
Nifi	1.7.1	3.00	6	3.00	80	3.00	280	3.00	552	✖
Mailer Plugin	1.20	3.00	36	3.00	221	3.00	504	3.00	945	✖
Primefaces	6.1	2.00	23	2.00	93	2.00	218	2.00	492	✖

V. Final Remarks

This paper presented SIEGE, a novel automatic exploit generation technique that supports security analysts and penetration testers to assess the impact of vulnerabilities in third-party open-source libraries. The results of our preliminary investigation indicate that the proposed approach deserves further investigation. In particular, future directions aim at:

- Investigating the impact of using ad-hoc GA settings instead of general-purpose setups;
- Evaluating SIEGE on real-world client applications, possibly of industrial relevance, and a larger set of vulnerabilities;
- Automating the creation of the *vulnerability descriptions*, as implemented in Eclipse Steady [16], and overcome the *line* granularity level limitation;
- Investigating the extent to which client classes facilitate or hinder the exploitation;
- Using SIEGE's outcomes to provide an estimation of the risk of keeping a certain vulnerable dependency (i.e., if SIEGE succeeds, it means that attackers could easily build harmful exploits).

Acknowledgements

Dario Di Nucci is supported by the European Commission grant no. 825040 (RADON H2020). Antonino Sabetta thanks Serena E. Ponta and Henrik Plate for insightful discussions.

REFERENCES

[1] Synopsis, "Open source security and risk analysis report 2019," 2020. [Online]. Available: https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-19.pdf

[2] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An empirical study of usages, updates and risks of third-party libraries in java projects," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 35–45.

[3] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: An evolutionary study," *Empirical Softw. Engg.*, vol. 20, no. 5, p. 1275–1317, Oct. 2015. [Online]. Available: https://doi.org/10.1007/s10664-014-9325-9

[4] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 181–191. [Online]. Available: https://doi.org/10.1145/3196398.3196401

[5] A. Zerouali, V. Cosentino, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the impact of outdated and vulnerable javascript packages in docker images," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 619–623.

[6] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," *Proceedings 2017 Network and Distributed System Security Symposium*, 2017. [Online]. Available: http://dx.doi.org/10.14722/ndss.2017.23414

[7] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1513–1531. [Online]. Available: https://doi.org/10.1145/3372297.3417232

[8] J. Williams and A. Dabirsiaghi, "The unfortunate reality of insecure libraries," 2014. [Online]. Available: https://cdn2.hubspot.net/hub/203759/file-1100864196-pdf/docs/Contrast_-_Insecure_Libraries_2014.pdf

[9] CVE, "Terminology," 2020. [Online]. Available: https://cve.mitre.org/about/terminology.html

[10] T. H. Bug, "The heartbleed bug," 2021. [Online]. Available: https://heartbleed.com/

[11] NIST, "National vulnerability database," 2021. [Online]. Available: https://nvd.nist.gov/

[12] OWASP, "Owasp top 10 web application security risks," 2021. [Online]. Available: https://owasp.org/www-project-top-ten/

[13] ——, "Owasp dependency check," 2021. [Online]. Available: https://owasp.org/www-project-dependency-check/

[14] W. Software, "Whitesource," 2021. [Online]. Available: https://www.whitesourcesoftware.com/open-source-security/

[15] Snyk, "Snyk," 2021. [Online]. Available: https://snyk.io/

[16] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 449–460.

[17] J. Hejderup, A. van Deursen, and G. Gousios, "Software ecosystem call graph for dependency management," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 101–104. [Online]. Available: https://doi.org/10.1145/3183399.3183417

[18] R. Kula, D. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, pp. 1–34, 05 2017.

[19] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vuln4real: A methodology for counting actually vulnerable dependencies," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[20] R. Zapata, R. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages," 09 2018, pp. 559–563.

[21] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the ide: Patterns, beliefs, and behavior," *IEEE Transactions on Software Engineering*, vol. 45, no. 3, pp. 261–284, 2017.

[22] T. Avgerinos, S. Cha, B. Hao, and D. Brumley, "Aeg: Automatic exploit generation." vol. 57, 01 2011.

[23] L. Xu, W. Jia, W. Dong, and Y. Li, "Automatic exploit generation for buffer overflow vulnerabilities," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2018, pp. 463–468.

[24] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2329–2344. [Online]. Available: https://doi.org/10.1145/3133956.3134020

[25] E. Iannone, "Toward Automated Exploit Generation for Known Vulnerabilities in Open-Source Libraries," 3 2021. [Online]. Available: https://bit.ly/3f28c0U

[26] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 411–420.

[27] S. E. Ponta, H. Plate, and A. Sabetta, "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3175–3215, 2020.

[28] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," 09 2011, pp. 416–419.

[29] D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," 1988.

[30] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

[31] C. Von Lücken, B. Barán, and C. Brizuela, "A survey on multi-objective evolutionary algorithms for many-objective problems," *Computational optimization and applications*, vol. 58, no. 3, pp. 707–756, 2014.

[32] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[33] "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990.

[34] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," *Information and Software Technology*, vol. 104, 08 2018.

[35] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19. IEEE Press, 2019, p. 383–387. [Online]. Available: https://doi.org/10.1109/MSR.2019.00064