Discovering Authorship of Vulnerabilities in Open Source Software

***Abstract*—Software vulnerabilities are, often the outcome of poor programming practices in the software developmental process. Elimination of bugs increases the expenses of the software. Bugs are created during software development process by programmers. Mitigation of flawed code in the programs can be achieved by training programmers to adopt secure code practices. Therefore, identification of flawed code created by authors becomes critical. In this work, a novel network theoretic approach to investigate the relationships between programmers and flawed code in open source software projects is created. Experiments are conducted on multiple open source software to evaluate and validate the approach using multiple metrics motivated by concepts from natural language processing. The software, data and analysis is available at https://github.com/damonot/authorship-analysis.**

***Index Terms*—Static Code Analysis, Natural Language Processing, Network Theory, Vulnerabilities.**

I. Introduction

Static code analysis (SCA) tools provide useful information about a codebase on a variety of levels, ranging from critical security threats to minor software bugs [1]. SCA tools do not investigate the programmers who wrote *flaws* in the code with the same scrutiny as other aspects of these code flaws such as software vulnerabilities. Flaws are fragments of source code that lead to software vulnerabilities. Since programmers are the authors behind the flaws and authorship is largely ignored during SCA, ultimately these tools treat the symptoms manifested as code flaws rather than the root cause, the authors. The occurrences of software vulnerabilities during software development is an outcome insecure programming practices by the programmers and primarily, coding errors [2]. Hence, it is critical to evaluate the programmers to mitigate software vulnerabilities. Studies have been performed on prediction of software flaws [3]. Software security audits have been proposed ([4], [5]) to minimize software vulnerabilities during software development process. We focus on relationships between programmers and programmer-flawed code with the intent to discover trends in the distribution of flaws. We use the flaws in the programs to construct relationship in the form of networks of programmer profiles to track flaws and prevent them at the source. A flawed code is considered to be any line of code that contains a bug or security vulnerability. In this study, we define *code author* as the person who created the first version of the code.

We study the following research questions by creating a network of programmers and flaws from open source codebases:

RQ1: Is there any programmer or a set of programmers influence on the software team creating the codebase?

RQ2: Is there any correlation between subsets of programmers creating flawed codes?

The research questions are addressed with experiments on data from static code analysis. The data is modeled in the form of network using metrics adapted from natural language processing. The defined metrics are used to perform network theoretic analysis. To the best of our knowledge, this is a novel application of network theory in connecting the flawed files with code-authorship.

II. Related Work

The research supporting our proposed formalization of programmer-centric software analysis sits in the intersection of work in network theory analysis (NTA) ([6], [7]) and SCA. Though, the programmer-centric approach draws more heavily from NTA research than SCA research as SCA is only relevant to the initial data collection while NTA is used for data analysis. A method describing code authorship and interaction have evaluated the benefits and limitation of professional developer's authorship and interaction with code [8]. Data dependent approaches in prediction software components have been studied[9]. Walden et al. [10] proposed the open-source software repositories for PhpMyAdmin, Drupal, and Moodle repositories for vulnerability mining. The primary method for collecting flaw code represented by source code is the application of static code analysis, dynamic code analysis (DCA), or a combined application of SCA and DCA. [11] highlights the fact that compared to DCA, SCA requires less time, but is also less accurate. Notably, this difference has much to do with the fact that DCA finds flaws at execution time while SCA finds flaws while the code is "static" or not running. As such, DCA requires a supplement of input data to provide the program it analyzes during run-time. Given this constraint, SCA can be used in analysis where input data is not available [11]. When selecting software to perform SCA against the data, SonarQube [12] as a powerful option, given it's ability to categorize bugs and vulnerabilities and rank the flaws of each category according to their threat. Additionally, the Drupal PhPmyAdmin, and Moodle repositories worked synergistically with SCA analysis performed by SonarQube. SonarQube has been used for static code analysis for web-based languages such as Php and JavaScript ([12],[13]). Even when SonarQube is compared to other code analyzing tools such as Leopard [14] which possess a flaw-ranking system and can outperform other SCA techniques, SonarQube performs well based on it's intuitive workflow and greater control over the final SCA results. Although, there are disadvantages of using SonarQube's because of its false positives found in the SNA results [15]. SonarQube's flaw categorization permits flaws which are more likely to be false-positives, the *security hotspots* category, to be ignored and therefore reduce the false positive rate of the data that forms the basis of network theoretic analysis. In the SCA phase, several undirected networks of authors are constructed according to what flaws they have written or where they have written flaws.

III. Methodology

The work can be divided in three distinct phases:

Phase 1: Generation of flaw data from the repositories with SCA using SonarQube.
Phase-2: Conversion of the flaw data output by SCA to networks.
Phase-3: Applying of network theoretic analysis on author-flaw networks.
In the construction of network from flaw data, several undirected networks of authors were constructed according to what flaws they have written or where they have written flaws. The network theoretic metrics such as proximity measures, correlation and assortativity were evaluated on the generated networks. We proposed a unique metric, *Author-Influence* based on the node degree of authors and the location of their flaws within the code (which is similar to TF-IDF)

A. Network Theoretic Analysis (NTA)

The two key algorithms in SNA are the adaptations of Alupopaie's edge-weighting scheme and Lazareva's BiCon bi-clustering tool. Supplementing these are the correlation assortativity and node proximity measures. Unique to this research is the "Author Influence" metric. It's important to note why these adaptations are at the forefront of the analysis; the modifications made here to Alupopaie and Lazareva's work give us the clearest look into what the habits of authors are, while the remaining metrics only reinforce the findings of these adapted algorithms.

Alupopaie (2013) presents a bipartite system where the edge weights of user-object networks were calculated using TF-IDF, popularly used in text analysis. TF-IDF (term frequency-inverse document frequency) is a term-weighting scheme that calculates the weight of keywords within a document using the frequency of the keyword in the current document relative to the frequency of that keyword across a set of documents. From this, TF-IDF extracts which words are unique and important to each document. Using TF-IDF weights for edge values, Alupoaie (2013) demonstrated that this relationship can be represented as a social network between the keywords and documents. In addition, Alupoaie (2013) also found that using TF-IDF for the weighting scheme improved the quality of community structures when compared to the original networks. As an extension of Alupoaie's work, we apply his bipartite network scheme with TF-IDF weighting to bipartite networks of authors and flaws for code security analysis; *terms* is substitued by *flaws* and *documents* are replaced with authors with dictionaries of their associated flaws. Hence, we refer to the adaption of Alupoaie's edge-weighting scheme as *flaw frequency* × *inverse author frequency* (FF-IAF). The algorithm FF-IAF is built on can be described as

$$M = f(a,x) \cdot \log(\frac{1+n}{1+g(A,x)})$$

Here, the variables a represents a specific author, x represents a specific flaw, n represents the total number of occurrences of that specific flaw across all authors, and A represents the set of all authors. From these, we can build the the function f to calculate how often flaw x is connected to author a and function g to determine how many authors have written this particular flaw. As a result, FF-IAF generates a matrix M of numerical scores for how closely all flaws and authors are associated with each other. Finally, it's worth noting that FF-IAF should not be considered the absolute metric for the quality of an author's code as the algorithm does not account for the amount of "clean" code an author commits to the repository. So, an author may have a high flaw score, but this does not necessarily imply they are "bad" programmer as this may only account for a small portion of the total code they have written.

B. Proximity

Proximity is the shortest path between nodes. For answering if bugs and vulnerabilities have a causal relationship, we use the Author-Flaw network, but we only need to consider the proximity of flaw nodes to each other, not authors. Assuming that a bug b and vulnerability v, are given by a pair, $\langle b, v\rangle$ and $k \in \mathbb{N}$ is the distance between b and v, we can plot to show proximity distribution between the pairs $\langle b, v\rangle$ vs k. By the nature of authors or files being the central node in their communities, all bugs and vulnerabilities always have a maximum distance of 1 node between them. As such, networks of only bugs or only vulnerabilities will have $k = 0$, while communities containing both will have $k = 1$. Using the distribution of k from any pair $\langle b, v\rangle$ across the entire set can help show if bugs lead to vulnerabilities based on the ratio between $k = 0$ and $k = 1$ communities. In other words, if we keep track of how often a pair $\langle b, v\rangle$ occurs in the network, we can determine if there is a relationship between the two flaws. Similar to correlation assortativity, author influence also deals with an author's degree of flaws and their distribution. An author's influence refers to how widespread the distribution of their flaws are and how numerous they are relative to other authors. Using the Author-Flaw and Author-File networks,.An author's influence can be calculated with the function:

$$f(x,y) = \frac{x}{\bar{x}} \cdot \frac{y}{\bar{y}}$$

where x represents the number of flaws an author has contributed, $\bar{x}$ represents the total number of flaws in the entire network, y represents the number of files the author has contributed flaws towards, and $\bar{y}$ represents the total number of files containing flaws. An author with an influence score of zero would be responsible for no flaws in the network, while an author with an influence score of one would be responsible for every flaw.

IV. Data Preparation

The datasets that were used in the analysis were: Phpmyadmin[16],Moodle [17] and Drupal [18]. The number of authors that actually created flawed code and the number of files containing flawed code was what I told you in bullet 2. The total number of authors that contributed to Moodle is 563 and the total number of files is 19,874. For PHPmyAdmin, there are 1,070 total authors and 3,366 total files. For Drupal, there are 43 total authors and 15,315 total files. SonarQube was used to perform SCA analysis on the source code, after which the results can be exported to a .CSV format. For each flaw, the .CSV output contains the following information: creation date of the vulnerability; date of SonarQube analysis; rule violated by the flaw; severity of the flaw, whether it's critical, major, or minor; parent file of the flaw; line number; and a recommendation of how to repair the line. The programmer(author of the code) of flawed code is important information and was retrieved using git.

A *flaw* refers to the rule violation during software implementation unless otherwise specified. The proximity measures, author influence, and correlation assortativity metrics used during network theoretic analysis. Four network types are constructed from the formatted SonarQube CSV files using the codebases of PhpMyAdmin, Moodle and Drupal. The first network connects authors to flaws, the second connects authors to files, the third connects flaws to one another, and the last connects coworkers. The Author-Flaw network is a bipartite graph and only possesses edges between authors and the flaws they've generated. For this network and all subsequent ones, author nodes contains only one field, the author. However, flaw nodes contain attributes for the rule violation, the flaw's parent file, and the flaw's line number. Author-Flaw is the primary network layout used in SNA and is used to track the frequency of flaws for authors and determine which flaws are unique to certain authors and which are shared among a set of authors. Next is the Author-File network, with edges between authors and the parent files that of flaws they've written, hereafter referred to as "flawed files." In contrast to the former structure, this layout shows where flaws are located, but does not show the flaw's location within the file nor the quantity of flaws within a file. Instead, the flawed files node only contains attributes for where the file exists and what types of flaws, i.e. "bug" or "vulnerability" (or both), exist in the parent file. Although it may seem that this perspective withholds important information about the network's structure by discarding flaw attributes, this network layout is crucial to calculating an author's influence. The third network layout removes author nodes from the network and instead creates two networks, one with edges between flaws of the same author and another with edges between flaws that share the same parent file. Using this network layout, we can track which code flaws are consistently linked to each other. The fourth and final network configuration is a coworker network. Two authors are considered coworkers if they have both contributed a flaw to the same file. This layout is imperative to calculating network assortativity, which tackles whether or not author node of similar degrees generally work together or independently. Additionally, it can be used as a comparative tool to optimize team structure. Tracking the evolution of coworker structure in conjunction with the rate of flaw-generation through the project's development can highlight how flaws increase or decrease with different coworker structures. In turn, comparing these structures and flaw-generation rate against each other can be used to determine how teams of authors should be structured to reduce the number of flaws generated overall.

V. Results

The outcome from analyzing the datasets, Drupal, PhpmyAdmin, and Moodle let to the following observation: specific flaws consistently linked to specific authors. Through our FF-IAF metric, we were able to show exactly what flaws are unique to any given programmer programmer and which are common throughout the entire team. Beyond that, for each author and flaw pair, FF-IAF generates a score that tells us how closely associated they are to each other. The only shortcoming of FF-IAF is that it doesn't account for the number of clean contributions that a programmer has made. So just because there is a leader in flaw generation, but it does not mean the programmer created programs with flaws.

Camilo (2015) also drew upon bipartite analysis when determining the relationship between bugs and vulnerabilities. In Camilo's work (2015), bipartite bug-file networks for were compared with bipartite vulnerability-file networks through the iterations of Chromium software releases to determine if there was a relationship between bugs and vulnerabilities, though no strong connection was found.

RQ1: *Is there any programmer or a set of programmers influence on the software team creating the codebase?* Author influence and degree centrality are two metrics which gauge how severe the negative impact of an author's flaws are to the repository. Author influence is the product of an author's Author-File and Author-Flaw degree centrality, while degree centrality itself is a simple sum of the edges connected to an author. Degree centrality was scaled to a range of zero to one by dividing over the total number of flaws or flawed files in the network in order to standardize the output. Standardizing this output allowed for easier evaluation of each repository's degree centralities. In the figures in the Appendix, if an author's file degree centrality is higher than the degree centrality of their flaws, this signifies that an author's code flaws are spread out over a large set of files. Conversely, if an author's flaw degree centrality is higher than their file degree centrality, then this reveals that an author's code flaws are concentrated in a small set of files. These metrics are important as their product determines an author's influence, and therefore how widespread an author's code flaws are across the repository.

When comparing the author influence distribution of each repository, an average of 95% +- 4.6% of the influence exists with the leading five influencers as shown in Figure 2. Additionally, the percent difference in the remaining authors is at minimum 1800 times smaller than the percent difference with

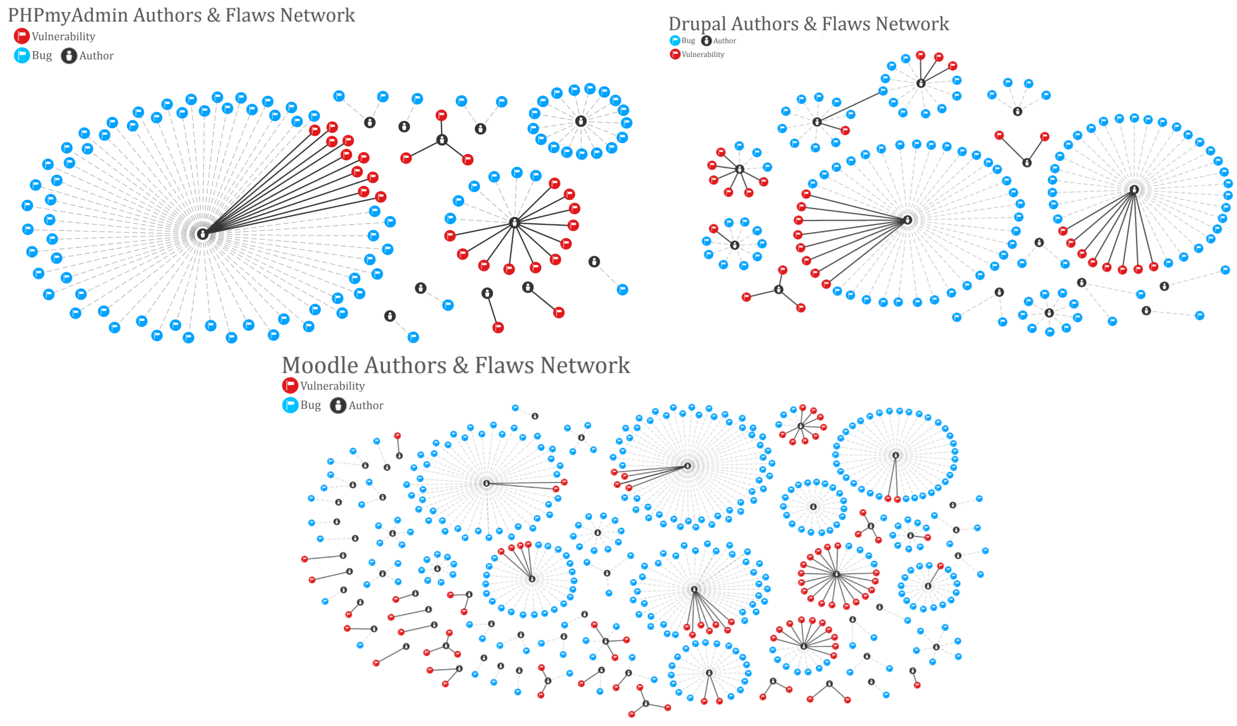
PHPmyAdmin Authors & Flaws Network
Vulnerability
Bug
Author
Drupal Authors & Flaws Network
Bug
Author
Vulnerability
Moodle Authors & Flaws Network
Vulnerability
Bug
Author

Fig. 1. Author-Flaw Network (a) PhpAdmin (b) Drupal (c) Moodle

the leading authors included, which suggests that eliminating the flaws caused by the leading five authors will resolve an overwhelming majority of the repositories code flaws. The number of authors that actually created flawed code and the number of files containing flawed code are: for Moodle, there is 54 authors across 227 files, for phpMyAdmin was 12 authors across 58 files and for Drupal, was 15 authors across 114 files.

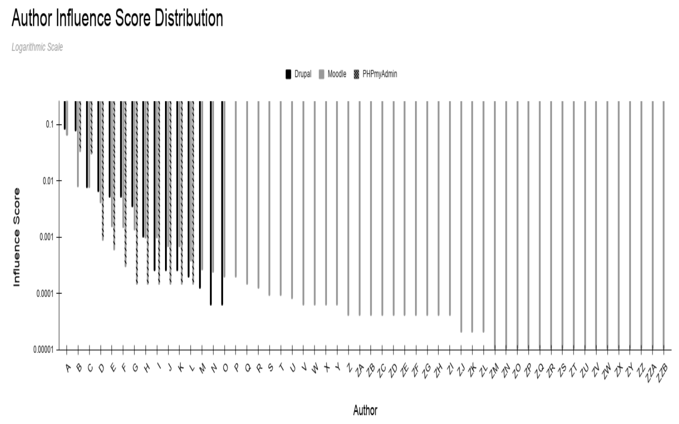

Fig. 2. Author Influence Score Distribution

RQ2 Is there any correlation between subsets of programmers creating flawed codes? Networks where nodes are connected to other nodes of similar degrees being considered assortative networks. Similarly, when nodes are mostly connected to other nodes of different degrees their network is considered disassortative. Analyzing any Author-Flaw network using degree correlation assortativity (DCA) results in a disassortative network, as the structure of Author-Flaw networks dictates that connections only exist between an author and a flaw leading to all flaws having a degree of one and an author whose degree equals the number of flawed code snippets written by the author. Likewise, File-Flaw networks are disassortative because all flaws connect to their parent files and files do not connect to each other. However, combining coworker networks with the author-flaw networks allows DCA to provide meaningful data. DCA suggests that there is no correlation, or at most a weak negative correlation, in the assortativity of coworkers. Drupal coworkers yielded -0.005 and Moodle yielded -0.042. The PhpMyAdmin result of +1.000 was not promising because it was based on insufficient data (only two sets of programmers). The extreme weak correlation between coworkers (programmers) suggests that there is no specific grouping among coworkers; authors who produce many code flaws work with each other just as frequently as they work with authors who produce few code flaws. Figure 3 shows the degree correlation assortativity computed from the networks. The distribution of author and flawed files 4 showed that there are few authors that contribute to the flaws. Similarly, Figure 5 shows the frequency of file node degree in each dataset, illustrating the distribution of flawed files with unflawed files.

VI. Conclusion

The identification of authors of flawed code is an useful investigation for mitigating software vulnerabilities and

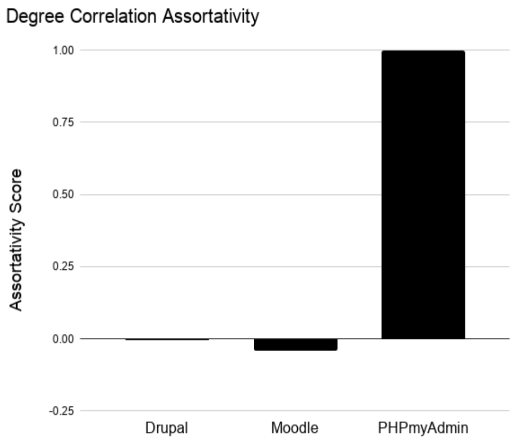

Fig. 3. Degree Correlation Assortativity

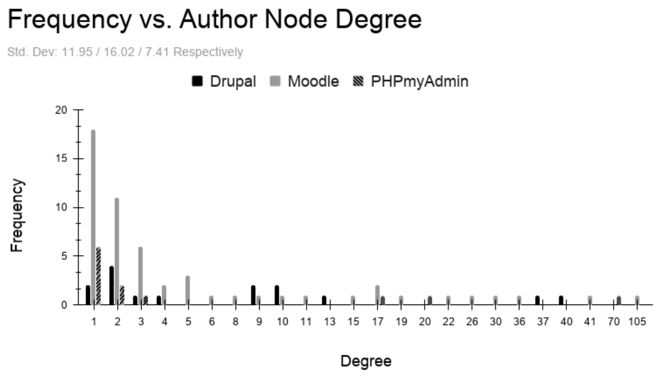

Fig. 4. Distribution of Author and Flawed Files

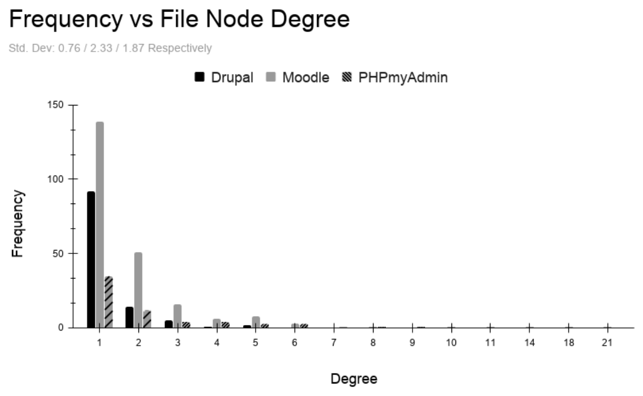

Fig. 5. Distribution of Flawed Files with Unflawed files

saving resources to fix the bugs. Network theoretic analysis of author-flaw network provides strong insights to pinpoint the programmers who can be trained so that the flawed files in the codebase during the software development process is minimized. The study provided several cues to understand the relationships between flaws and programmers. In future, rigorus analysis using advanced data mining approaches will be performed to reveal intricate and useful details from the author-flaw networks.

References

[1] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, no. 4, pp. 363–387, 2011.

[2] D. Plakosh, R. Seacord, R. Stoddard, D. Svoboda, and D. Zubrow, "Improving the automated detection and analysis of secure coding violations," Carnegie-Mellon University, Pittsburgh, PA Software Engineering Institute, Tech. Rep., 2014.

[3] L. Kanashiro, A. Ribeiro, D. Silva, P. Meirelles, and A. Terceiro, "Predicting software flaws with low complexity models based on static analysis data," *Journal of Information Systems Engineering & Management*, vol. 3, no. 2, p. 17, 2018.

[4] J. Heffley and P. Meunier, "Can source code auditing software identify common vulnerabilities and be used to evaluate software security?" in *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the*. IEEE, 2004, pp. 10–pp.

[5] S. Schmeelk, B. Mills, and L. Hedstrom, "Standardizing source code security audits," *International Journal of Software Engineering & Applications*, vol. 3, no. 1, p. 1, 2012.

[6] F. Menczer, S. Fortunato, and C. A. Davis, *A first course in network science*. Cambridge University Press, 2020.

[7] E. David and K. Jon, *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. USA: Cambridge University Press, 2010.

[8] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 385–394.

[9] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[10] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *2014 IEEE 25th international symposium on software reliability engineering*. IEEE, 2014, pp. 23–33.

[11] C. Artho and A. Biere, "Combined static and dynamic analysis," *Electronic Notes in Theoretical Computer Science*, vol. 131, pp. 3–14, 2005.

[12] G. A. Campbell and P. P. Papapetrou, *SonarQube in action*. Manning, 2014, vol. 392.

[13] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, and G. Pinto, "Are static analysis violations really fixed?: a closer look at realistic usage of sonarqube," in *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, 2019, pp. 209–219.

[14] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "Leopard: Identifying vulnerable code for vulnerability assessment through program metrics," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 60–71.

[15] B. Johnson, Y. Song, and E. Murphy-Hill, "Why don't software developers use static analysis tools to find bugs?" ISCE, 2013.

[16] "PhpMyAdmin 5.0.1," https://github.com/phpmyadmin/phpmyadmin. Last accessed Jan 7, 2020.

[17] "Moodle v3.8.1," https://github.com/moodle/moodle, Last accessed Jan 10, 2020.

[18] "Drupal 8.8.2," https://github.com/moodle/moodle, Last accessed: Dec 18, 2019.