Characterizing Packages for Vulnerability Prediction

Saviour Owolabi*, Francesco Rosati*, Ahmad Abdellatif, Lorenzo De Carli
University of Calgary, Alberta, Canada
{firstname.lastname@ucalgary.ca}

***Abstract*—Modern software development relies heavily on the use of external libraries and packages as software reuse provides benefits, such as reduced time to market and lower development cost. However, these libraries often come with their own set of direct and indirect dependencies which could introduce vulnerabilities, compromising the security of end users. Prior work shows that developers may remain unaware of these vulnerabilities until a security incident that exploits them occurs, leading to potential consequences for data privacy. Therefore, it is essential for developers to have the ability, before committing time to a project, to understand whether the external libraries and packages they intend to use may induce vulnerabilities, and how that might happen.**

In our work, we use the dataset made available by the Goblin framework to identify and evaluate salient features for predicting the vulnerability profile of software packages. We use these features to build classifiers for predicting whether or not a dependency-related vulnerability will occur within 3, 6, or 12 months. Our approach proves to be effective, achieving F1-scores of 0.74, 0.79 and 0.86 in the 3, 6, and 12 month contexts respectively. Providing timely vulnerability information could help developers identify potential security weaknesses before deploying a package to production, thereby minimizing the risk of security incidents.

***Index Terms*—Dependency Analysis, Vulnerability Prediction, Software Security, Machine Learning, CVE Analysis, Goblin Framework, Neo4J, Maven Central.**

I. Introduction

The reuse of external software packages is a common practice in modern development, enabling developers to reduce costs, accelerate timelines and build upon well-tested components [1]. Modern software development heavily relies on external libraries and frameworks, often forming the backbone of applications across various domains [2]. Dependencies enable developers to rapidly create robust systems by leveraging years of cumulative innovation encapsulated in reusable code modules. This practice not only promotes ecosystem-wide standardization and interoperability but also ensures scalability and efficiency in complex software projects [3]. However, despite their advantages, software dependencies could introduce significant challenges and security risks for end users [4]. Applications often rely on multi-layered dependency structures, where a single package may depend on dozens or even hundreds of other libraries [5]. This creates a network effect where vulnerabilities in one library can spread throughout the system [6]. The impact of software vulnerability can be significant and result in compromise and data loss [7], at a significant cost for companies [8], [9].

* Equal contribution

Traditionally, developers become aware of these flaws only after security incidents, undermining data privacy and system integrity [10]. It is therefore essential to predict such vulnerabilities in advance, providing useful tools for identifying risks during the early stages of development. Current methods for managing software vulnerabilities, such as Common Vulnerabilities and Exposures (CVE) scanning, are predominantly reactive, focusing on patching after an issue is detected. These approaches overlook the complex propagation of vulnerabilities through dependency structure [11]. Also, many tools lack intuitive interfaces and seamless integration into developers' workflows, limiting their effectiveness [12]. Additional problems include poor interpretability of prediction models and a lack of consideration for contextual factors specific to software packages, leaving significant gaps in risk mitigation [13].

Our work proposes a predictive model based on machine learning, leveraging dependency graphs and code metrics from the Maven ecosystem (Maven Central) to assess the likelihood of a given software package being associated with one or more CVEs over a time frame ranging from three to twelve months. Our work evaluates several classification techniques, including Naïve Bayes, Random Forest, XGBoost and Logistic Regression, to identify and analyze patterns in historical data. We utilized the Goblin framework [14] to extract data from the Maven ecosystem, including CVEs, update metrics and more, to create a rich dataset comprising information on over 15 million nodes and 134 million edges.

The development of our predictive model presented several challenges that required innovative solutions. First, **handling data complexity:** the large scale of the dataset required extensive and advanced preprocessing approaches, as well as an in-depth study of tools like Neo4j and Cypher to make queries more efficient. Another issue is that of **reducing false negatives:** we adopted robust models, such as Random Forest, to improve the system's sensitivity in detecting critical vulnerabilities. Finally, **interpretability** poses a challenge as model decisions should ideally be explainable. We perform feature analysis to make the model's results understandable and practical for decision-making.

The implemented models demonstrated strong predictive capabilities, operationalized as effectiveness in forecasting the likelihood that one or more dependencies in a Maven software package could be associated with one or more vulnerabilities in the future. In this task, the model achieves F1 score up to 86% and precision up to 83%. Additionally, the system was also able to identify and recommend risk-free dependencies,

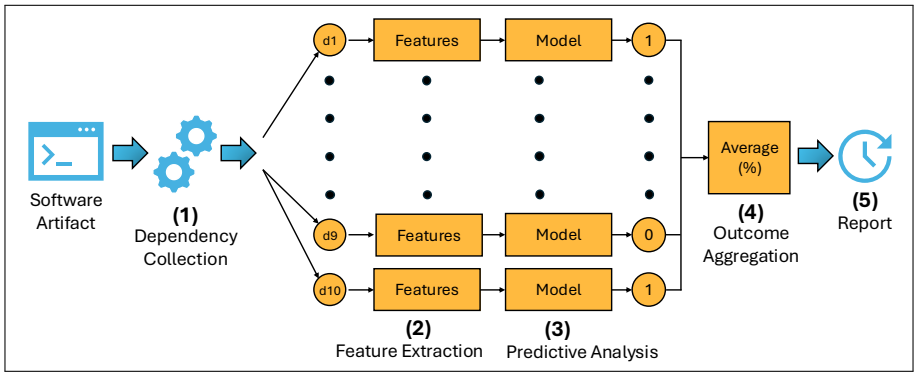

Fig. 1: Model Architecture

providing useful insights for developers. Our key contributions are as follows:

- We evaluate multiple classification models on a custom vulnerability prediction dataset built using the Goblin framework.
- We implement robust feature selection and preprocessing to enhance model accuracy and interpretability.
- To ensure ease of use for developers, we pair the model with an easy-to-use web-based UI to initiate analysis and visualize results and make our replication package publically available online [15].

Building upon these contributions, our study addresses the following research questions:

- **RQ #1:** Which algorithms and feature engineering best predict vulnerabilities? (Section IV-A)
- **RQ #2:** What are the key features in Maven graphs and metadata that significantly influence vulnerability risks? (Section IV-B)
- **RQ #3:** How can the model's results be presented to facilitate quick, data-driven security decisions? (Section IV-C)

II. Related Work

Vulnerability Prediction

Yasasin et al. [16] explore time-series forecasting models for post-release software vulnerability prediction. Leverett et al. [17] argue that predictive models can help strategic vulnerability management by estimating the volume and characteristics of upcoming CVEs, supporting proactive patching and resource allocation. Kalouptsoglou et al. [18] provide a systematic mapping of vulnerability prediction methods, highlighting trends like increased reliance on deep learning and code analysis. Li et al. [19] propose VulPecker, which uses code similarity to detect specific software vulnerabilities, demonstrating the utility of pattern- and similarity-based detection systems. Scandariato et al. [20] and Dam et al. [21] evaluate models to predict vulnerable mobile app components, based on analysis of small datasets of Android apps. Shin et al. [22] evaluate a variety of predictors, finding that several metrics can predict vulnerabilities with high accuracy. These studies emphasize the need for combining predictive, statistical and automated methods for effectively vulnerability management. However, gaps remain, particularly in the integration of this techniques with real-world operational challenges, such as highly dynamic software ecosystems. This research study builds on these insights, with the aim of enhancing predictive accuracy and applicability in practical scenarios.

Risk Mitigation Hasib et al. [13] propose a Morphological Analysis (MA) framework to systematize risk mitigation in software projects. The study highlights the value of models such as Bayesian statistics and genetic algorithms, for enhancing risk prediction and mitigation.

Defect Prediction Software defect analysis techniques seek to detect [23]–[28] or predict [29]–[32] bugs in software. Other work focuses on characterizing general assessment of software quality based on a variety of inputs [33]. While these techniques are closely related to the ones discussed above, the notion of "bug" is broader than that of "vulnerability" as not all correctness issues may have direct security implications.

Software Datasets Jaime et al. [14] introduced the Goblin framework, a tool for enriching and querying the Maven Central dependency graph. Goblin combines a temporal dependency graph meta-model with on-demand metrics such as CVEs and freshness, enabling ecosystem-level analysis and proactive library management.

III. Methodology

To model the vulnerability risk accrued by a release due to its dependency profile, we collected a dataset of relevant software packages, and we used it to generate historical vulnerability information for three **look-ahead** periods. We then evaluated four classification models in predicting whether vulnerability would arise within each period.

A. Datasets

We created our datasets with features collected from the Goblin Neo4J database [14] and the OSV (Open Source Vulnerabilities) database [34]. Each node in the Goblin Neo4J graph represents either an artifact (identified by its group and artifact ID) or a release (characterized by version and timestamp information). Dependency relations are modeled as edges between nodes, representing direct or transitive dependencies. These edges also encode additional metadata, such as version ranges, dependency scopes (e.g., compile, test, runtime) and timestamps which are crucial to our feature collection process.

TABLE I: Model Performance for Each Scenario

	Precision			Recall			Accuracy			F1-Score		
Model	**3M**	**6M**	**12M**	**3M**	**6M**	**12M**	**3M**	**6M**	**12M**	**3M**	**6M**	**12M**
Naive Bayes	0.54	0.59	0.64	0.44	0.46	0.47	0.78	0.74	0.70	0.48	0.51	0.54
Random Forests	**0.79**	**0.82**	**0.83**	**0.69**	**0.76**	**0.79**	**0.88**	**0.88**	0.81	**0.74**	**0.79**	**0.86**
XG Boost	0.78	**0.82**	**0.83**	0.67	0.74	0.77	**0.88**	0.87	**0.85**	0.72	0.78	0.80
Logistic Regression	0.66	0.68	0.70	0.36	0.41	0.48	0.80	0.77	0.72	0.47	0.51	0.57

The Goblin Neo4J provides vulnerability information for known CVEs associated with artifacts but does not include publication dates. As such, we collected this data from the OSV database. Our analysis focused on modeling transitive dependencies, so direct vulnerabilities were excluded from CVE-related feature computation.

The datasets we created (one each for the 3-month, 6-month, and 12-month prediction scenarios) contain 315,000 artifacts each, with each row corresponding to a randomly sampled **Release** from the Goblin Neo4J database. Sampling was stratified by the number of dependencies associated with each release. As our model is predicated on the effect of dependencies on security events, we excluded releases not associated with any dependencies from our sample.

To create our datasets, we queried the Neo4J database using Cypher queries to extract relevant data, including artifact, release nodes, dependency edges and associated metadata like timestamps and speed (releases per day). These values are computed and integrated dynamically using the **Goblin Weaver**.

B. Feature Collection

Our approach is depicted in Figure 1. In the Goblin framework, *CVE* data represent vulnerabilities that directly impact a release, while *CVE_AGGREGATED* includes both direct and transitive vulnerabilities. To predict vulnerabilities arising from dependencies, we designed a binary target, **CVEx**, which indicates whether the *CVE_AGGREGATED* data for the artifact includes vulnerabilities not present in *CVE* and were published within a specified prediction window.

To extract features, for each release under consideration, we first collect a list of direct dependencies (Step **1** in Figure 1) by querying the Goblin Neo4J database (when making predictions in our integrated solution, the dependencies are obtained by parsing Maven POM files). For each direct dependency, we extract the following features(Step **2**):

1) **Speed**: Average number of releases per day for an artifact [35].
2) **Time Difference**: Time difference (in days) between when a release was published and when the dependency under consideration was published.
3) **Month of Release**: The month in which the dependency under consideration was released.

We also aggregated features from prior versions of each dependency (up to 10 prior versions) to enrich the dataset with information on the dependency's past vulnerability behavior.

1) **Number of Prior Versions Considered**: Number of prior versions of a dependency over which data points have been accumulated, up to a limit of 10.

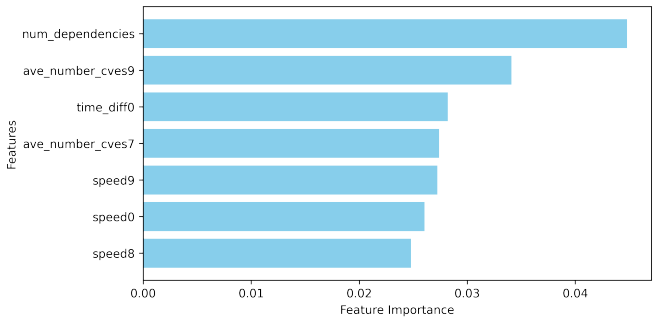

Fig. 2: Feature Importance

2) **Number of Prior Versions with dependency-related CVEs**: Number of prior versions with transitive CVEs.
3) **Average Number of dependency-related CVEs**: Average number of dependency-related CVEs across prior versions.
4) **Average Number of Days till First dependency-related CVE**: Average number of days from the publication of prior versions of a release till a dependency-related CVE was published against it.

For uniformity in the dataset creation process, ensuring we had a consistent number of columns, dependency-related features were collected only for the ten most outdated dependencies of each release. Prior research has identified outdated dependencies as a significant vulnerability risk factor [6], [36]. This choice is also supported by our analysis of the Maven dataset, which showed that 75% of the releases had fewer than 11 dependencies. Therefore, for most projects, our approach captures data points for all dependencies. For packages with fewer than 10 dependencies, available features were replicated.

Along with the dependency-related features, we collected the release month and the number of dependencies per release. Prior studies have identified a cyclical pattern in the occurrence of security events, linked to hacker conferences [37].

C. Model Engineering

In each scenario (3-months, 6-months, 12-months), we trained four classifiers (Naïve Bayes, Random Forests, XGBoost, and Logistic Regression) on our data discussed in Section III. These models were used to predict the likelihood that a vulnerability arises for each dependency within the target look-ahead period (Step **3**). The application we designed leverages these models to make per-dependency predictions for any software package entered for analysis. This allows us to alert users about dependencies that might pose a vulnerability risk within the prediction time frame. Following the predictions, the results are aggregated (Step **4**) to compute an overall assessment of the software package and, the aggregated outcome is returned to the user in a detailed assessment report (Step **5**).

TABLE II: Impact of no. of Dependencies on Performance

Num. Dependencies	Precision	Recall	Accuracy	F1-Score
1	0.79	0.71	0.82	0.75
3	0.83	0.76	0.85	0.79
5	0.83	0.77	0.85	0.80
8	0.83	0.79	0.86	0.81
10	0.83	0.79	0.86	0.81

D. Implementation

We implement our approach in Python using Scikit-learn for machine learning, Pandas for data processing and FastAPI for building the back-end interface. Our implementation interfaces with Neo4J to extract dependency data and CVE data from the OSV database. It also processes POM files from Maven Central or GitHub to execute the predictions. We implement a UI for the predictor using the React.js framework.

IV. Evaluation

A. Predictor evaluation (RQ #1)

Table I presents the results of four classification algorithms (Naïve Bayes, Random Forests, XGBoost, and Logistic Regression) on vulnerability prediction at 3, 6 and 12 months on our test set, following hyper-parameter tuning (with 5-fold cross-validation). From the table, Random Forests emerges as the best-performing algorithm, achieving an F1-score of 86% over a 12-month look-ahead period. This, along with the strong performance of XG Boost highlights that tree-based models are particularly effective in capturing the complex patterns in dependency graphs, answering RQ #1. We attribute the performance improvement with longer look-ahead periods to differences in dataset balance [38]. Vulnerability reports increase over time, making longer look-ahead datasets (12M>6M>3M) more balanced than shorter ones that contain fewer positive samples.

B. Feature Analysis (RQ #2)

a) Feature Importance: We extracted feature importance data from our best performing model (Random Forests w/ 12-month). The analysis (Fig. 2) highlights that the number of dependencies is the most critical factor in predicting vulnerabilities, confirming the hypothesis that larger dependency graphs increase risk. In addition, the average number of CVEs in more recently released dependencies emphasizes the significance of historical vulnerability trends, particularly in indirect dependencies. Temporal aspects, such as the time difference between publication of a package and its dependencies also play key roles, with actively maintained dependencies (captured by the speed feature) reducing risks. These findings directly answer RQ #2 by underscoring the importance of dependency management and proactive updates. Note that the indexes attached to the feature names (where applicable) correspond to the dependency they are associated with, from most outdated *0* to least outdated *9*.

b) Impact of Number of Dependencies: To evaluate the impact that the number of dependencies considered might have, we trained the Random Forests model (under the same conditions described above), but varying the number of dependencies considered (i.e., 1, 3, 5, 8 , and 10 dependencies), and evaluated the model's performance in those contexts. Table II presents the result of this analysis, showing that the number of dependencies considered has a positive impact on model performance, up to 8 dependencies. This indicates that the number of dependencies considered is an important model hyper-parameter of our model.

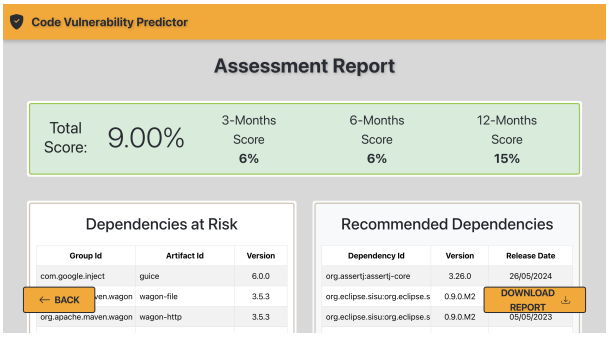

Fig. 3: Code Vulnerability Predictor Application UI

C. User interface (RQ #3)

To address RQ #3, we developed a ReactJS web interface (Fig. 3) that facilitates quick, data-driven security decisions. The interface allows users to specify a project to be analyzed, by uploading a corresponding POM file. The identified dependencies are sent to the machine learning model for analysis. Once the predictions are returned, the interface displays the three predicted risk scores (3-month, 6-month and 12-month scores) along with their average (total score). Additionally, it provides users with a list of identified at-risk dependencies and recommended alternatives. These alternatives are identified through an additional database query that retrieves dependencies released between the current date and the past two years, which do not have any associated *CVE_aggregated*.

V. Discussion

Our model achieves promising predictive performance. However, there are some potential limitations. The model primarily relies on dependency metadata, which may not fully capture the intricate interactions contributing to software vulnerabilities. For instance, specific attributes of source code such as code complexity, coding patterns, or direct references to known vulnerabilities, could provide additional insights. Another limitation concerns generalizability. Our model is trained on Maven, which could restrict its applicability to other ecosystems. Expanding the dataset to include platforms such as npm or PyPI, could help identify potential differences in vulnerability trends across ecosystems.

Furthermore, the focus of the dataset on dependency graphs does not consider variations in dependency risk across different contexts, such as popular versus unpopular packages, which tend to have different security profile and impact [39]. Exploring these differences in future work could allow targeted interventions to address vulnerabilities. In summary, this work highlights the considerable potential of machine learning for vulnerability prediction while also identifying key areas for advancement.

References

[1] A. Stefi, K. R. Lang, and T. Hess, "A contingency perspective on external component reuse and software project success," in *Americas Conference on Information Systems*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:38819262

[2] P. Boldi and G. Gousios, "Fine-grained network analysis for modern software ecosystems," *ACM Trans. Internet Technol.*, vol. 21, no. 1, Dec. 2020. [Online]. Available: https://doi.org/10.1145/3418209

[3] S. S. Ostadzadeh, F. Shams, and K. Badie, "An architectural model framework to improve digital ecosystems interoperability," in *New Trends in Networking, Computing, E-learning, Systems Sciences, and Engineering*, K. Elleithy and T. Sobh, Eds. Cham: Springer International Publishing, 2015, pp. 513–520.

[4] Q. Li, J. Song, D. Tan, H. Wang, and J. Liu, "Pdgraph: A large-scale empirical study on project dependency of security vulnerabilities," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 161–173.

[5] P. Boldi, "How network analysis can improve the reliability of modern software ecosystems," in *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*, 2019, pp. 168–172.

[6] J. Hejderup, "In Dependencies We Trust: How vulnerable are dependencies in software modules?" Master's thesis, Delft University of Technology, May 2015.

[7] R. K. Vaidya, L. De Carli, D. Davidson, and V. Rastogi, "Security Issues in Language-based Software Ecosystems," Nov. 2019, arXiv:1903.02613 [cs]. [Online]. Available: http://arxiv.org/abs/1903.02613

[8] C. O'Donnell, "The 'event-stream' vulnerability," https://medium.com/@codfish/the-event-stream-vulnerability-6acd4c515aae, Dec. 2018.

[9] L. Constantin, "SolarWinds attack explained: And why it was so hard to detect — CSO Online," https://www.csoonline.com/article/3601508/solarwinds-supply-chain-attack-explained-why-organizations-were-not-prepared.html, Dec. 2020.

[10] K.-U. Loser and M. Degeling, "Security and privacy as hygiene factors of developer behavior in small and agile teams," in *ICT and Society*, K. Kimppa, D. Whitehouse, T. Kuusela, and J. Phahlamohlaka, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 255–265.

[11] T. Bhuddtham and P. Watanapongse, "Time-related vulnerability lookahead extension to the cve," in *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, 2016, pp. 1–6.

[12] F. Nembhard and M. Carvalho, "The impact of interface design on the usability of code analyzers," in *2019 SoutheastCon*, 2019, pp. 1–6.

[13] N. Hasib, S. W. A. Rizvi, and V. Katiyar, "Risk mitigation and monitoring challenges in software organizations: A morphological analysis," *International Journal on Recent and Innovation Trends in Computing and Communication (IJRITCC)*, vol. 11, no. 8, pp. 172–185, sep 2023.

[14] D. Jaime, J. E. Haddad, and P. Poizat, "Goblin: A framework for enriching and querying the maven central dependency graph," in *IEEE/ACM MSR*, 2024.

[15] Anonymous, "Characterizing packages for vulnerability prediction," https://zenodo.org/records/14317549, 2024, (Accessed on 12/08/2024).

[16] E. Yasasin, J. Prester, G. Wagner, and G. Schryen, "Forecasting it security vulnerabilities – an empirical analysis," *Computers & Security*, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016740481830854X

[17] E. Leverett, M. Rhode, and A. Wedgbury, "Vulnerability forecasting: Theory and practice," *Digital Threats*, 2022. [Online]. Available: https://doi.org/10.1145/3492328

[18] I. Kalouptsoglou, M. Siavvas, A. Ampatzoglou, D. Kehagias, and A. Chatzigeorgiou, "Software vulnerability prediction: A systematic mapping study," *Information and Software Technology*, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058492300157X

[19] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2991079.2991102

[20] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, pp. 993–1006, 2014.

[21] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. C. Grundy, and A. K. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Transactions on Software Engineering*, vol. 47, pp. 67–85, 2021.

[22] Y. Shin, A. Meneely, L. A. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, pp. 772–787, 2011.

[23] T. Dey and A. Mockus, "Deriving a usage-independent software quality metric," *Empirical Software Engineering*, vol. 25, no. 2, p. 1596–1641, Mar. 2020.

[24] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE*, 2005.

[25] T. Zimmerman, N. Nagappan, K. Herzig, R. Premraj, and L. Williams, "An empirical study on the relation between dependency neighborhoods and failures," in *ICST*, 2011.

[26] E. Wyss, L. De Carli, and D. Davidson, "(nothing but) many eyes make all bugs shallow," in *SCORED*, 2023.

[27] I. Herraiz, E. Shihab, T. H. Nguyen, and A. E. Hassan, "Impact of installation counts on perceived quality: A case study on debian," in *2011 18th Working Conference on Reverse Engineering*, 2011, pp. 219–228.

[28] T. Khoshgoftaar, E. Allen, N. Goel, A. Nandi, and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system," in *ISSRE*, 1996.

[29] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653–661, 2000.

[30] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *ESEM*, 2007.

[31] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *ICSE*, 2008.

[32] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *ISESE*, 2006.

[33] D. Spinellis, G. Gousios, V. Karakoidas, P. Louridas, P. J. Adams, I. Samoladas, and I. Stamelos, "Evaluating the quality of open source software," *Electronic Notes in Theoretical Computer Science*, vol. 233, pp. 5–28, 2009, proceedings of the International Workshop on Software Quality and Maintainability (SQM 2008). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066109000632

[34] "OSV - Open Source Vulnerabilities," https://osv.dev/.

[35] D. Jaime, J. El Haddad, and P. Poizat, "A Preliminary Study of Rhythm and Speed in the Maven Ecosystem," in *21st Belgium-Netherlands Software Evolution Workshop*, Mons, Belgium, Sep. 2022. [Online]. Available: https://hal.science/hal-03725099

[36] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 109–118.

[37] E. Yasasin, J. Prester, G. Wagner, and G. Schryen, "Forecasting IT security vulnerabilities – An empirical analysis," *Computers & Security*, vol. 88, p. 101610, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016740481830854X

[38] J. Van Hulse, T. Khoshgoftaar, and A. Napolitano, "Experimental perspectives on learning from imbalanced data," vol. 227, 06 2007, pp. 935–942.

[39] M. Taylor, R. Vaidya, D. Davidson, L. De Carli, and V. Rastogi, "Defending Against Package Typosquatting," in *NSS*, 2020.