Towards Detection of Malicious Software Packages Through Code Reuse by Malevolent Actors

Marc Ohm,[1] Lukas Kempf,[2] Felix Boes,[3] Michael Meier[4]

Abstract: Trojanized software packages used in software supply chain attacks constitute an emerging threat. Unfortunately, there is still a lack of scalable approaches that allow automated and timely detection of malicious software packages and thus most detections are based on manual labor and expertise. However, it has been observed that most attack campaigns comprise multiple packages that share the same or similar malicious code. We leverage that fact to automatically reproduce manually identified clusters of known malicious packages that have been used in real world attacks, thus, reducing the need for expert knowledge and manual inspection. Our approach, AST Clustering using MCL to mimic Expertise (ACME), yields promising results with a F_1 score of 0.99. Signatures are automatically generated based on characteristic code fragments from clusters and are subsequently used to scan the whole npm registry for unreported malicious packages. We are able to identify and report six malicious packages that have been removed from npm consequentially. Therefore, our approach can support the detection by reducing manual labor and hence may be employed by maintainers of package repositories to detect possible software supply chain attacks through trojanized software packages.

Keywords: Software Supply Chain; Malware; Abstract Syntax Tree; Markov Cluster Algorithm

1 Introduction

Modern software is developed on top of an ever-growing pool of third party tools, libraries and packages. Hence, the integrity of software projects depend directly on the integrity of the underlying software supply chain. Over the past few years, software supply chain attacks that leverage trojanized software packages kept emerging [Oh20]. A central role in that ecosystem is held by maintainers of package repositories like npm or Python Package Index (PyPI). These platforms are repeatedly abused for the distribution of trojanized software packages that are part of a software supply chain attack.

From the point of view of a malware author, distributing trojanized software packages to open source package repositories is a both cost-efficient and effective for a number of reasons. This is because open source packages are frequently used in various software projects. However, the packages code base is unlikely to be audited by the user of the

[1] University of Bonn, Institute for Computer Science 4, Friedrich-Hirzebruch-Allee 8, 53115 Bonn, Germany ohm@cs.uni-bonn.de

[2] University of Bonn & Hochschule Bonn-Rhein-Sieg kempf@uni-bonn.de

[3] University of Bonn boes@cs.uni-bonn.de

[4] University of Bonn & Fraunhofer FKIE mm@cs.uni-bonn.de

doi:10.18420/sicherheit2022_02

package. Therefore, malicious packages tend to be available for roughly 200 days before being identified and removed [Oh20]. Clearly, an improvement of automated capabilities for timely detection of such attacks is mandatory. It must be noted that at least npm and PyPI are taking effort to implement the detection of malicious software packages [Ha21; Py22].

Based on a manual annotated dataset, we evaluate various automated approaches to mimic the manual clustering of malicious packages which is typically performed by an expert. Following the idiom “if you’ve seen one, you’ve seen them all” we automatize the identification of related malicious packages to keep the upper hand in the arms race of software supply chain attacks. Consequentially, we propose a timely detection of malicious packages based on signatures derived from identified clusters. To this end, we leverage Abstract Syntax Trees (AST) that are generated from known malicious packages that have previously been used in real world attacks. Eventually, clusters of packages that share source code are identified through Markov Cluster Algorithm (MCL).

Our results indicate excellent performance ($F_1 = 0.99$) on the annotated dataset and good scalability for practical application. This way, suspicious packages are detected as soon as they are published to a package repository. It supports the detection by notifying an analyst about suspicious packages and giving hints about possible connections to already known malicious packages. Thus, the contribution of this paper is the automated and signature-based detection of possibly malicious packages which may then be analyzed by an expert. Eventually, we were able to detect and report six incidents of malicious packages on npm that share code with previously distributed malicious packages.

The remainder of this paper is organized as follows. Sect. 2 provides related work to frame the academic context of our approach. The underlying methodology for our approach is depicted in Sect. 3. Our results are presented and discussed in Sect. 4. Sect. 5 concludes the paper and provides an outlook for future work.

2 Related Work

The detection of similar code fragments is being used for the detection of software plagiarism. To this end, a vast majority of approaches leverage ASTs [CDR09; CJ11; DG13; NJK19; RKC18].

The detection of source code similarity also found application in cyber security. Especially, the detection of software vulnerabilities leverages code similarity detection in order to identify known but currently undetected vulnerabilities in software. Known vulnerable source code is used to generate signatures which are used to scan other source codes for these known patterns. [Bi20; Ch20; Li16; YLR12]

In contrast to vulnerable software components, the research field of malicious software packages is comparably new. Most often, they try to identify possible typosquatting attacks [Ta20; Ts16; Vu20]. Nonetheless, there are approaches that try to detect malicious packages based on their source code. For instance, by looking for anomalies in a package's source codes compared to all other packages [Ča19; Ga19] or by leveraging heuristics [Du20; PO17] of presumably malicious characteristics. Moreover, dynamic analysis of suspicious packages might be considered in order to detect malicious behavior [Du20; OSM20]. Furthermore, Ohm et al. [Oh20] systematized software supply chain attacks by collecting and analyzing a large dataset of malicious open source packages that have been used in real world attacks.

Now, with an annotated dataset of known malicious packages at hand, it is possible to leverage insights and ground truth to develop suitable detection techniques. Our contribution differs from the approaches mentioned beforehand by leveraging evidence-based characteristics of known malicious packages. In this work, we focus on static code analysis in order to detect malicious packages based on automatically generated signatures.

3 Methodology

As observed by Ohm et al. [Oh20], malicious packages tend to have characteristic code fragments in common. This might be because they are employed in the same attack campaign or were simply copied by other malware authors. As discussed in the introduction, an automated approach that solves the time-consuming and tedious tasks to identify and search for said characteristic code fragments is highly anticipated. Our approach consists of three automated steps: (1) calculation of the source code similarity between all known malicious packages, (2) clustering based on these similarities, and (3) generation of signatures for each cluster.

After providing our metric that measures how well the automated clustering mimics the manual approach, we evaluate the clustering on an annotated dataset. In order to show that our tool is efficient and feasible for practical application on the large scale, it is evaluated on the full npm repository in Sect. 4.3. Hereby, we are able to identify and report six malicious packages that have been removed from npm consequentially.

3.1 Embedding of Packages

Each package consists of a number of source files and each source file consists of a number of functions. In our approach, each function is represented as Abstract Syntax Tree. AcornJs [Ma20], a lightweight parser for JavaScript, is used to transform source code into AST representation. Through abstraction of source code into a structural representation, naming of identifiers is of no matter. Hereby, we treat member functions as independent functions and group all statements in the global scope into a function. Similarity of two ASTs, i.e., two packages, is calculated according to the Tree Edit Distance introduced by Zhang-Shasha [ZS89] using the Python package zss [HJ13]. The similarity of two packages, A and B, is the smallest Tree Edit Distance when comparing all ASTs, i.e., all functions, of package A to all ASTs of package B. This way, we can quantify the similarity between all malicious packages at hand which is then used to identify clusters among these.

3.2 Clustering of Packages

The defined goal is to group known malicious packages. Under the assumption that the malicious packages mostly share the malicious code, we can now identify the relevant code fragment for each cluster and use it as signature. To this end, we evaluate several unsupervised clustering algorithms on the packages (using the distances between their representations) and compare the resulting clusters with the experts clustering of malicious packages. Hereby, unsupervised methods have been chosen to reduce the need for prior knowledge about the dataset. We evaluate connected component (*ccomp*) and maximal cliques (*clique*) by leveraging the Python package `NetworkX` [HSS08]. Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is implemented by using the Python package `scikit-Learn` [Pe11] and Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) by `hdbscan` [MHA17]. Last, we examine Markov Cluster Algorithm (MCL) [va00] for which we leverage the Python package `Markov-Clustering` [Al20]. We expect the automated approach to identify the same amount of clusters containing the same set of malicious packages as if performed manually.

All approaches are evaluated on the "Backstabber's Knife Collection" dataset [Oh20][5] as ground truth. This dataset contains malicious packages harvested from npm, PyPI, and RubyGems. Furthermore, a manual clustering with expert knowledge of these was performed that formed seven clusters comprising 109 malicious packages from npm. For sake of brevity and with respect to the amount of npm packages in the dataset, we focus on packages written for Node.js in JavaScript. However, our approach is transferable to all kind of programming languages.

In order to compare the manual clustering by Ohm et al. [Oh20] with a fixed automated clustering approach, the conceiving metrics Precision, Recall and F_1-score are employed as follows. First of all, we assume that the manual clustering is complete and accurate, i.e., every malicious code similarity is found and packages are clustered correctly. Then, a pair of packages is said to be a **true positive** if the two packages are in the same cluster in both approaches, a **true negative** if the two packages are in different clusters in both approaches, a **false positive** if the two packages are in different manually generated clusters but in the same automatically generated cluster, and a **false negative** if the two packages are in the same manually generated clusters but in the different automatically generated clusters.

With this definition, the metrics Precision, Recall and F_1-score are interpreted as follows. By definition, Precision is the ratio of true positives in all positives. Therefore, the Precision

[5] For reproducibility we use the version of the dataset as described in the publication.

is high if the number of false positives is relatively low. Observe that this is the case if the automated approach generates clusters that are overall finer or as fine as the manual approach. Observe analogously that the Recall is high if the automated approach generates clusters that are overall coarser or as coarse as the manual approach. Consequentially, the F_1-score (which is the harmonic mean of Precision and Recall) measures how well the automated clustering mimics the manual approach.

3.3 Derivation of Signatures

It turns out that AST Clustering using MCL to mimic Expertise (ACME) mimics the manual clustering of an expert nearly perfectly with an F_1-score of 0.99. Based on top of ACME, the signature of a cluster of malicious packages is derived by constructing a so called fingerprint from each characteristic code fragment.

Our approach leverages fingerprinting as proposed by Chilowicz et al. [CDR09]. From each function f represented in an AST G, we derive a so called fingerprint $\mathcal{H}_f$. To this end, we focus our attention to the subgraph $G_f \subset G$ associated to f after all (nested) functions that are defined inside f are discarded. For each node $v \in G_f$, we concentrate on its type $t(v)$. After fixing SHA-256 as hash function C, the fingerprint of f is defined recursively as follows. Given an arbitrary node $v \in G_f$ with children $w_1, w_2, \ldots$, we define $\mathcal{H}(v)$:

$$\mathcal{H}(v) = C\left(t(v) \parallel \mathcal{H}(w_1) \parallel \mathcal{H}(w_2) \parallel \ldots\right) \tag{1}$$

Denoting the root of G_f by r_f, the fingerprint of f is

$$\mathcal{H}_f = \mathcal{H}(r_f) \tag{2}$$

We remark that we leverage a different function t than Chilowicz et al. Our $t(v)$ solely takes the type of node into account. Thus, our subgraph G_f is very focused on the structure of the code fragment by discarding nonstructural information like operators. For instance the code fragments $a + b$ and $a * b$ result in the same fingerprint. However, $a + (a + a)$ and $(a + a) + a$ yield different fingerprints. Let us remark further that we group code into a dummy global function if it resides outside of functions, i.e., in global scope. Furthermore, we treat functions inside classes as independent functions.

For each cluster c, one or more characteristic code fragment are chosen to serve as signature S_c as follows. By construction, a cluster consists of a set of ASTs and to each AST, we associate a fingerprint $\mathcal{H}$. Roughly speaking, a fingerprint is simply a hash of a given AST. Now a fingerprint $\mathcal{H}$ is "characteristic" if the following conditions are met: (1) The code fragment $\mathcal{H}$ is unique to its cluster, i.e., $\mathcal{H}$ is not derived from any package in any other cluster, (2) the code fragment $\mathcal{H}$ is derived from at least two packages in its cluster, and (3) the code fragment $\mathcal{H}$ cannot be derived from one of the most depended upon packages[6] from npm.

Observe that (1) ensures that the signatures of the clusters are pairwise disjoint. This means that a newly classified package is assigned to one cluster only. Observe further that the condition (2) focuses signatures on common and hence descriptive code for the analyzed cluster. Condition (3) forbids code fragments that can be found in popular and hence supposedly benign packages.

The signature of a cluster c comprises all characteristic code fragments, i.e., fingerprints, of that cluster. The use of ASTs and the leveraged abstraction level are able to detect exact clones of known malicious code by definition. By discarding identifier names, the approach becomes resilient against obfuscation through unreadable names and renaming in general. Now, a package p matches the signature $\mathcal{S}_c$ of cluster c if at least one of p's fingerprints $h_1^p, \ldots, h_N^p$ matches a fingerprint $h \in \mathcal{S}_c$.

4 Results

This section summarizes our results from experiments as introduced in the previous sections. First, we evaluate which clustering approach is suited best in combination with AST to reproduce the results of the manual clustering in Sect. 4.1. The best approach is leveraged in Sect. 4.2 to automatically generate and optimize signatures based on identified clusters. These signatures are subsequently used in Sect. 4.3 to scan the whole npm registry for unreported malicious packages that have code fragments common to known malicious packages.

4.1 Reproduction of Clustering

Recall from Sect. 3 that we aim to automate the tedious and time-consuming task of manually finding (variations of) recognized malicious code fragments in a given package repository. Hereby, packages with similar malicious code fragments are clustered using various unsupervised cluster algorithms. In this subsection, we evaluate the quality of these approaches that attempt to reproduce the result of the manual clustering of Ohm et al. [Oh20].

Tab. 1 displays the performance of ASTs in conjunction with all clustering algorithms. It is noticeable that most clustering algorithms yield either high Precision or high Recall. Solely, MCL is capable of reaching both high Precision and high Recall thus recreating the manual cluster as similar as possible. With a Precision of 0.97 and a Recall of 1.00 the F_1 score is

[6] `https://www.npmjs.com/browse/depended`, we are limited to the 108 most depended upon packages due to technical issues of the website.

at 0.99. Through the use of ACME we are able to recreate the manual clustering performed by expert almost perfectly.

Having a suitable, automated, and unsupervised clustering at hand, signatures can now be derived to describe the syntactic characteristics of malicious packages of that cluster. Using these signatures, packages related to the same attack campaign, i.e., sharing some code fragments, are detected and identified automatically. The upcoming subsection evaluates the quality of the automatically generated signatures to quantify their practical suitability.

4.2 Quality of Signatures

The previous subsection shows that the experts task of manually clustering malicious code is automated almost perfectly by combining ASTs with MCL. Using the ACME approach described in Sect. 3.3, a signature S_c is derived for each cluster c. In this subsection, we discuss the sizes of the clusters and we demonstrate that the first two conditions yield signatures with a promising Recall.

In Tab. 2, the resulting clusters, their sizes, and corresponding number of signatures are shown. Our approach automatically identified seven clusters that cover 97 packages. As stated initially, 104 packages belong to a manual created cluster in the dataset. This is because the manual clustering by Ohm et al. also took dependency into account for clustering. Our approach solely relies on code syntax similarity and hence may not cluster all packages as in the dataset. However, only few clusters based on dependency exists and hence ACME was able to reproduce the manual results almost perfectly. Nonetheless, if the manual clustering is flawed, our approach also contains these flaws.

It is noticeable that the sizes of clusters varies heavily ($\sigma^2 = 230.40$). The smallest ones comprise two packages while the biggest clusters are of size 38 and 36 respectively. The size of a signature weakly correlates with the size of the corresponding cluster (Pearson $r = 0.65$, $p = 0.12$). However, there are outliers. For instance cluster 1 and 5 yield very large signature compared to their size and in contrast to that cluster 2 yields a very small signature.

Tab. 1: Results of AST and all clustering approaches with employed parameters, sorted by F_1 score.

Clustering	Parameter	Precision	Recall	F_1
MCL	exp = 2, inf = 2	0.9747	0.9958	0.9851
ccomp		0.6761	0.9958	0.8054
DBSCAN	$\varepsilon = 1$, minPts = 2	0.6761	0.9958	0.8054
HDBSCAN	minClst = 2	0.6580	0.9967	0.7927
clique		0.9878	0.6074	0.7522

In order to demonstrate that condition (3) is mandatory, we test the quality of the signatures associated to all fingerprints satisfying only conditions (1) and (2) as follows. In a 10-fold cross validation, we cluster the 114 packages containing malicious code with ACME and derive the signatures associated to all fingerprints satisfying conditions (1) and (2). These signatures are evaluated against the 10% of the split in the cross validation and against 108 benign packages. In this context, a package is positive if the automatically generated signature matches the package. On average, the Recall is 0.88 but the number of false positives is 46%. This is because the signatures contain too many fingerprints of benign functions, i.e., condition (3) is mandatory to reduce the number of false positives.

In total 3,875 fingerprints satisfying only conditions (1) and (2) are derived from the malicious packages of the seven clusters. Considering relevant fingerprints, i.e., after applying condition (3), the seven clusters yield 3,396 (-12.36%) fingerprints in total.

4.3 Large Scale Evaluation

For large scale evaluation of our signatures, we harvest the npm repository on 25th of September 2020. At this time, 1,396,447 packages were listed and respectively 1,396,413 are obtained in their “latest” version. In total, 20,017,543 files and 749,558,178 function are inspected.

Tab. 2 also lists the number of matches ($|\mathcal{M}_c|$) our signatures produced per cluster. After the automated removal of false positive fingerprints, the total amount of matches is reduced from 283,887 to 136,157 (-52.04%). For manual optimization, we inspect the 50 most matching fingerprints for each cluster. This takes roughly 10 minutes per cluster and resulted in 133 signatures (3.92%) being removed. This further reduces the amount of matches from 136,157 to 70,432 (-48.27%).

Tab. 2: Identified clusters based on ACME sorted by size. The size of corresponding signatures $\mathcal{S}_c$ and matches M_c are based on the level of optimization: Consider all fingerprints satisfying only conditions (1) and (2), all conditions, and all conditions plus manual optimization.

		cond. (1) + (2)		cond. (1), (2) + (3)		cond. (1), (2), (3) + manual	
No.	Size	$\lvert\mathcal{S}_c\rvert$	$\lvert M_c\rvert$	$\lvert\mathcal{S}_c\rvert$	$\lvert M_c\rvert$	$\lvert\mathcal{S}_c\rvert$	$\lvert M_c\rvert$
1	38	3,752	278,473	3,282	131,842	3,232	70,228
2	36	1	1	1	1	1	1
3	14	40	1,137	34	694	2	0
4	3	3	3	3	3	3	3
5	2	75	4,191	72	3,536	22	200
6	2	2	81	2	81	1	0
7	2	2	0	2	0	2	0
$\sum$	97	3,875	283,887	3,396	136,157	3,263	70,432

It must be noted that the signatures of cluster 1 are responsible for most of the matches (99.71%). This indicates that our approach fails to choose descriptive fingerprints from this cluster and hence produces many false positive matches. This indicates that a more sophisticated selection of relevant fingerprints might be needed. Solely 204 suspicious packages need manual inspection when leaving out matches from that cluster. However, for the remaining clusters the ACME is able to pre-select a reasonable number of suspicious packages that need further manual inspection. Furthermore, ACME gives a hint to the analyst by stating suspicious functions of matching packages.

In addition to automatically generated signatures, we manually create signatures for packages that did not belong to a cluster. To this end, we extract the fingerprint of malicious functions by hand. This results in eight new pseudo-cluster with corresponding signatures. However, this yields only one additional match.

4.3.1 Detected Packages

By the construction of the signatures, see Sect. 3.3, every match is treated as suspicious and hence needs manual inspection to verify actual maliciousness. Eventually, we were able to identify seven unreported but malicious packages that have code in common with known malicious packages. We identify the packages `nodetest199`[7], `nodetest1010`[8], and `plutov-slack-client`[9] based on the signature of cluster 4. All of them try to establish a reverse shell upon installation in order to give the attacker full control of the victims' systems.

Furthermore, ACME detects the packages `revshell` and `node-shells` that claim to be proof of concept packages. Thus, npm security did not publish a security advisory for `revshell` but nonetheless removed it from the registry. However, the package `node-shells` was published by Adam Baldwin, head of security at npm, and was thus not reported by us.

The package `hellhun_homelibrary` which was found by a manually generated signature was indeed not a malicious package itself. It is affected by `flatmap-stream`, the malicious package that was used in the `event-stream` incident. Hence, npm security decided to inform the developer about it instead of removing it.

The publication dates of the packages related to cluster 4, namely `nodetest1010` (2018-08-03), `nodetest199` (2018-08-02), and `plutov-slack-client` (2018-03-30), fit the overall time frame of known malicious packages from that cluster (2018-03-06, 2018-09-08, 2018-03-25). Thus, we conclude that we identified remnants of a previous attack.

However, the package `npmpubman`[10] matches the signature of cluster 2 and it aimed at the

[7] https://www.npmjs.com/package/nodetest199

[8] https://www.npmjs.com/package/nodetest1010

[9] https://www.npmjs.com/package/plutov-slack-client

[10] https://www.npmjs.com/package/npmpubman

exfiltration of data about the victims' systems. It was published on 2020-09-13 which is way later than the average package from cluster 2. We reported it on 2020-09-28, only 15 days after it was published.

Overall, we conclude that our automated generation of signatures reduces the workload for manual analysis drastically. However, manual optimization is still required to further boil down the amount of matches, thus further reducing the amount of suspicious packages for manual inspection. Nonetheless, our straight forward approach already yields feasible results. Hence, ACME can be leveraged to search packages for variations of known malicious code.

5 Conclusion

In this paper, we examine how source code similarities of known malicious packages can be leveraged to support the detection of software supply chain attacks. Our main goal is to automatically detect malicious software packages that are uploaded to large package repositories like npm. To this end, we automatize the clustering and signature generation of known malicious packages which is typically based on expert knowledge and manual inspection. On a dataset of 114 malicious npm packages that are used in real world attacks, we evaluate several approaches to find and group syntactical similarities in source codes. Based on that, clusters of packages with similar structure are identified automatically and unsupervised.

Compared to the manual clustering of these packages at hand, our best approach yields promising results ($F_1 = 0.99$). It leverages Abstract Syntax Trees (AST) to compare source code of multiple packages and Markov Cluster Algorithm (MCL) to identify clusters among these and is hence called AST Clustering using MCL to mimic Expertise (ACME). Our approach can be used to support the detection by pre-selecting suspicious packages based on signatures of known malicious code fragments for manual inspection. This reduces the need for expert knowledge and manual inspection drastically.

To demonstrate the effectiveness of ACME, a scan of the whole npm registry based on all generated signatures is performed. This revealed seven previously unreported packages in total. A manual inspection shows that four of them are indeed malicious packages and are therefore reported by us and subsequently removed from npm. Two of the remaining three are proof of concept packages. The last package itself is not malicious but it contains a full copy of the known malicious package `flatmap-stream` as dependency.

In conclusion, this means that our approach is feasible to automatically generate signatures for known malicious packages which then may be used to scan packages for remnants, variations, and imitators of known malicious packages. Through the use of ASTs the approach is resilient to modifications of the source code like renaming of variables and minor structural modifications. Furthermore, ACME is transferable to any other programming language. However, automatically generated signatures reduce the manual work needed but still cause many false positives which may be removed manually. Nonetheless, our naive approach already yields promising results and good scalability.

For future work we plan to optimize our signature generation and enhance support for structural modifications of the source code. Eventually, we would like to expand our approach to other software ecosystems like Python Package Index (PyPI) and RubyGems.

Acknowledgment

This work is funded under the SPARTA project, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830892.

References

[Al20] Allard, G.: Markov Clustering, 2020, URL: `https://github.com/GuyAllard/markov_clustering`, visited on: 10/29/2020.

[Bi20] Bilgin, Z.; Ersoy, M. A.; Soykan, E. U.; Tomur, E.; Çomak, P.; Karaçay, L.: Vulnerability Prediction From Source Code Using Machine Learning. IEEE Access 8/, pp. 150672–150684, 2020.

[Ča19] Čarnogursky, M.: Attacks on Package Managers, MA thesis, Masaryk University, Faculty of Informatics, 2019.

[CDR09] Chilowicz, M.; Duris, E.; Roussel, G.: Syntax tree fingerprinting for source code similarity detection. In: 2009 IEEE 17th International Conference on Program Comprehension. IEEE, pp. 243–247, 2009.

[Ch20] Chinthanet, B.; Ponta, S. E.; Plate, H.; Sabetta, A.; Kula, R. G.; Ishio, T.; Matsumoto, K.: Code-based Vulnerability Detection in Node. js Applications: How far are we? arXiv preprint arXiv:2008.04568/, 2020.

[CJ11] Cosma, G.; Joy, M.: An approach to source-code plagiarism detection and investigation using latent semantic analysis. IEEE transactions on computers 61/3, pp. 379–394, 2011.

[DG13] Djurić, Z.; Gašević, D.: A source code similarity system for plagiarism detection. The Computer Journal 56/1, pp. 70–86, 2013.

[Du20] Duan, R.; Alrawi, O.; Kasturi, R. P.; Elder, R.; Saltaformaggio, B.; Lee, W.: Measuring and preventing supply chain attacks on package managers. arXiv preprint arXiv:2002.01139/, 2020.

[Ga19] Garrett, K.; Ferreira, G.; Jia, L.; Sunshine, J.; Kästner, C.: Detecting suspicious package updates. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). IEEE, pp. 13–16, 2019.

[Ha21] Hanley, M.: GitHub's commitment to npm ecosystem security, 2021, URL: https://github.blog/2021-11-15-githubs-commitment-to-npm-ecosystem-security/, visited on: 01/06/2022.

[HJ13] Henderson, T.; Johnson, S.: Zhang-Shasha: Tree edit distance in Python, 2013, URL: https://pythonhosted.org/zss, visited on: 01/15/2021.

[HSS08] Hagberg, A.; Swart, P.; S Chult, D.: Exploring network structure, dynamics, and function using NetworkX, tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[Li16] Li, Z.; Zou, D.; Xu, S.; Jin, H.; Qi, H.; Hu, J.: VulPecker: an automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd Annual Conference on Computer Security Applications. Pp. 201–213, 2016.

[Ma20] Marijn Haverbeke, I. S. et al.: Acorn, 2020, URL: https://github.com/acornjs/acorn, visited on: 10/21/2020.

[MHA17] McInnes, L.; Healy, J.; Astels, S.: hdbscan: Hierarchical density based clustering. Journal of Open Source Software 2/11, p. 205, 2017.

[NJK19] Novak, M.; Joy, M.; Kermek, D.: Source-code similarity detection and detection tools Used in academia: a systematic review. ACM Transactions on Computing Education (TOCE) 19/3, pp. 1–37, 2019.

[Oh20] Ohm, M.; Plate, H.; Sykosch, A.; Meier, M.: Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, 2020.

[OSM20] Ohm, M.; Sykosch, A.; Meier, M.: Towards detection of software supply chain attacks by forensic artifacts. In: Proceedings of the 15th International Conference on Availability, Reliability and Security. ACM, pp. 1–6, 2020.

[Pe11] Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V., et al.: Scikit-learn: Machine learning in Python. the Journal of machine Learning research 12/, pp. 2825–2830, 2011.

[PO17] Pfretzschner, B.; ben Othmane, L.: Identification of Dependency-based Attacks on Node. js. In: Proceedings of the 12th International Conference on Availability, Reliability and Security. Pp. 1–6, 2017.

[Py22] PyPI Warehouse: Malware Checks, 2022, URL: https://warehouse.pypa.io/development/malware-checks.html, visited on: 01/06/2022.

[RKC18] Ragkhitwetsagul, C.; Krinke, J.; Clark, D.: A comparison of code similarity analysers. Empirical Software Engineering 23/4, pp. 2464–2519, 2018.

[Ta20] Taylor, M.; Vaidya, R. K.; Davidson, D.; De Carli, L.; Rastogi, V.: SpellBound: Defending Against Package Typosquatting. arXiv preprint arXiv:2003.03471/, 2020.

[Ts16] Tschacher, N. P.: Typosquatting in programming language package managers, BA thesis, Universität Hamburg, Fachbereich Informatik, 2016.

[va00] vanDongen, S.: A cluster algorithm for graphs. Information Systems [INS]/R 0010, 2000.

[Vu20] Vu, D.-L.; Pashchenko, I.; Massacci, F.; Plate, H.; Sabetta, A.: Typosquatting and Combosquatting Attacks on the Python Ecosystem. Proc. of EuroS&PW'20/ , 2020.

[YLR12] Yamaguchi, F.; Lottmann, M.; Rieck, K.: Generalized vulnerability extrapolation using abstract syntax trees. In: Proceedings of the 28th Annual Computer Security Applications Conference. Pp. 359–368, 2012.

[ZS89] Zhang, K.; Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. SIAM journal on computing 18/6, pp. 1245–1262, 1989.