On the Feasibility of Supervised Machine Learning for the Detection of Malicious Software Packages

Marc Ohm
ohm@cs.uni-bonn.de
University of Bonn & Fraunhofer FKIE
Bonn, NRW, Germany

Felix Boes
boes@cs.uni-bonn.de
University of Bonn
Bonn, NRW, Germany

Christian Bungartz
ch.bungartz@uni-bonn.de
University of Bonn
Bonn, NRW, Germany

Michael Meier
mm@cs.uni-bonn.de
University of Bonn & Fraunhofer FKIE
Bonn, NRW, Germany

ABSTRACT

Modern software development heavily relies on a multitude of externally – often also open source – developed components that constitute a so-called Software Supply Chain. Over the last few years a rise of trojanized (i.e., maliciously manipulated) software packages have been observed and addressed in multiple academic publications. A central issue of this is the timely detection of such malicious packages for which typically single heuristic- or machine learning based approaches have been chosen. Especially the general suitability of supervised machine learning is currently not fully covered. In order to gain insight, we analyze a diverse set of commonly employed supervised machine learning techniques, both quantitatively and qualitatively. More precisely, we leverage a labeled dataset of known malicious software packages on which we measure the performance of each technique. This is followed by an in-depth analysis of the three best performing classifiers on unlabeled data, i.e., the whole npm package repository. Our combination of multiple classifiers indicates a good viability of supervised machine learning for the detection of malicious packages by pre-selecting a feasible number of suspicious packages for further manual analysis. This research effort includes the evaluation of over 25,210 different models which led to True Positive Rates of over 70 % and the detection and reporting of 13 previously unknown malicious packages.

CCS CONCEPTS

• **Security and privacy** → **Intrusion/anomaly detection and malware mitigation**; • **Computing methodologies** → *Supervised learning by classification*; • **Software and its engineering** → *Software libraries and repositories*; • **Information systems** → Open source software.

ARES 2022, August 23–26, 2022, Vienna, Austria

ACM ISBN 978-1-4503-9670-7/22/08…$15.00
https://doi.org/10.1145/3538969.3544415

KEYWORDS

Software Supply Chain, Supervised Machine Learning, Malware Detection

ACM Reference Format:
Marc Ohm, Felix Boes, Christian Bungartz, and Michael Meier. 2022. On the Feasibility of Supervised Machine Learning for the Detection of Malicious Software Packages. In *The 17th International Conference on Availability, Reliability and Security (ARES 2022), August 23–26, 2022, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3538969.3544415

1 INTRODUCTION

Modular software largely benefits from opportunistic software reuse. This allows for the fast and thus cost-efficient creation of new software by leveraging already existing functionalities of ready to use software components. While that method eases and accelerates the process of software development it also conveys danger. Each added dependency (i.e., software component) again has several nested dependencies. This leads to a chain of intransparent dependencies implicitly related to trust in the included software and authors of it. This chain of software dependencies is commonly referred to as a Software Supply Chain.

Very recent incidents that highlight the problems of intransparent Software Supply Chains are `log4j` [4] as well as the software packages `faker` and `colors` [19]. The first was used in numerous Java projects but contained a very easy to abuse vulnerability. In the second incident, the author of these packages decided to corrupt his own packages in order to highlight the commercial exploitation of open source developers. As a direct consequence, this led to failure in all depending upon software projects.

In case of a Software Supply Chain Attack, an attacker maliciously manipulates a less secure dependency down the Software Supply Chain in order to attack the end product, its developers, and users. Unfortunately, there has been an increasing number of malicious packages released in the recent years. Most notable is the case of `event-stream`. Through social engineering, an attacker was able to get publishing rights of said package which they then weaponized with a malicious dependency. This dependency targeted the software of a bitcoin wallet further up the Software Supply Chain. Its intention was to steal cryptocurrency from that wallet's users. [12]

Attackers have a variety of entry points to infiltrate software development with a Software Supply Chain attack. As described

by Ohm et al. [12], prominent attack vectors are (1) releasing a new package that contains or includes malicious code and (2) the annexation and trojanization of existing packages. Typically, an attacker's goal is to further distribute the malicious code along software supply chains. Thus, a neuralgic point in the ecosystem of open source software development and software reuse is taken by package repositories like PyPI, npm, RubyGems, and many others. These function as main distributor of off the shelf software packages. For many years, these package repositories' sole purpose has been hosting and distributing software packages. But, due to the ever-growing number of misuses, the question of liability for the distribution of malicious packages comes up. Furthermore, the detection of malicious packages is a time-consuming job that is often performed manually. Thus, it is desirable to pre-select suspicious packages with a high confidence in order to reduce the workload of analysts.

While there are also academic approaches (cf. Section 2) for the detection of malicious packages, package repositories started to implement and stated commitment efforts to prevent further misuse of their infrastructure for Software Supply Chain Attacks [7, 15].

The goal of this work is the mitigation of Software Supply Chain attacks by assisting an analyst in choosing potentially harmful software packages of a package repository for further inspection. To this end, a set of features that are characteristic for known malicious packages is compiled. A diverse set of supervised machine learning approaches and preprocessing techniques is evaluated on a manually labeled dataset in order to determine the feasibility of them regarding the use case. Among the 25,210 different models, a combination of the three best performing models is evaluated on all packages available on npm, on which we perform an exploratory study. As a side product, we detect and report 13 previously unknown malicious packages.

The remainder of this paper is structured as follows. Section 2 gives an overview on related work and how these tried to detect malicious software packages. In Section 3, our methodology under which we performed our experiments is depicted. The leveraged dataset and features used for the machine learning techniques are detailed in Section 4. Results are presented in Section 5 and discussed in Section 6. Finally, a conclusion and directions for future work are given in Section 7.

2 RELATED WORK

Our research touches the area of malware detection using machine learning which has been adopted in general [17, 21]. We, however, focus on software packages that are distributed through open source package repositories like npm, PyPI, and RubyGems. This allows to optimize the approach for a single language and a predefined distribution format of the software. More specifically, we aim at the detection of malicious open source software components – a research area that gained attention in the recent years. Various approaches that try to detect malicious packages have been published.

Pfretzschner et al. [14] use a heuristic-based and static analysis of the source code to determine whether a Node.js package is malicious or not. However, they were not able to identify any real-world dependency-based attacks and hence leverage an artificial dataset. Their approach currently yields high False Positives and high False Negatives.

Duan et al. [3] combine dynamic and static analysis to classify packages in npm, PyPI and RubyGems based on a heuristic rule set. They were able to identify and report 339 malicious packages in total using (an unknown number of) iterative labeling rounds where in each round the rule set is updated to reduce identified False Positives. Moreover, their research contains a systematization of security issues in the established package repositories as well as a threat model.

Ohm et al. present two approaches. One leverages the dynamic analysis of software components in order to identify unusual and thus suspicious changes in the behavior of the software and its dependencies [13]. That approach heavily relies on manual inspection and vetting of the found artifacts. The other one utilized unsupervised signature generation that focuses on the detecting of known malicious code fragments [11]. That approach relies on nearly no user interaction but may cause a high number of False Positives if the signatures are not specific enough. This, however, may be fine-tuned by manual verification of generated signatures.

Besides inspecting software packages for malicious code or behavior there are also studies that take a broader look at the ecosystem around packages.

Vu et al. [23] evaluate if there is a discrepancy between the openly available source code and the published package, and if this may be employed to identify malicious packages. When tested on a dataset of known malicious packages they were able to correctly identify all of them.

A rule-based decision model for malicious commits on GitHub is investigated by Gonzalez et al. [6] with which they are able to correctly identify roughly 53 % of known malicious commits. Garrett et al. [5] employed an unsupervised learning technique to identify anomalies in package updates. Their approach reduced the number of manual inspections needed by 89 %, but they did not verify if the preselected updates where indeed malicious. When tested on `eslint-scope` (a known malicious package) their approach yields correct results. In fact, we leverage some of their proposed features for our approach (cf. Section 4).

A highly related paper by Sejfia et al. [18] was published as preprint just shortly before finalizing our work. They, too, investigate the suitability of supervised machine learning models for the detection of malicious packages with comparable results. Our work differs in some aspects, but is ultimately able to substantiate and complement their findings.

Moreover, there are approaches to automatically detect typosquatting and combosquatting attacks in various open source ecosystems [20, 24]. Furthermore, there are studies investigating the misuse of package repositories for attacks and the impact throughout the Software Supply Chain in general [1, 2, 8, 9, 22, 26].

3 METHODOLOGY

In this work, we investigate the feasibility of supervised machine learning for the detection of malicious software packages. More precisely, we aim to assist an analyst by highlighting software packages

that are supposed to be malicious. To this end, we investigate the npm repository and answer the following two research questions:

Question 1 *Are supervised machine learning techniques suitable to effectively (re)identify malicious software packages in a package repository?*

Here, the evaluated machine learning techniques provide promising results. However, each technique tends to misclassify certain clusters of software packages. Since npm provides more than 1.9 million packages [10], reducing the False Positive Rate is highly anticipated. To this end, we answer the second research question.

Question 2 *To what extent does the detection quality benefit from combining diverse supervised machine learning techniques?*

In order to answer both questions, several experiments as described in the following section are conducted.

3.1 Experiments

The quality of supervised machine learning techniques heavily relies on a sound dataset that consists of telling features. Recall that package repositories are insufficiently curated, i.e., software packages are not reviewed, and it is unknown which packages are benign (or malicious).

Focusing on npm, we compose a first sound dataset by combining the 150 malicious packages of [12] with the top 15,000 npm packages (which are assumed to be benign), see Section 4. This way, the ratio of malicious and benign packages is 1:100, i.e., 1 % malicious, which is our assumed ratio for npm packages (hopefully, too high). Here, our set of features comprises npm specific features (e.g., we analyze whether an installation script opens a network connection) as well as npm agnostic features (e.g., we search for suspicious strings like */etc/shadow* or *.bashrc*).

In this work, we train and evaluate the seven supervised machine learning methods described in Table 5. Each method requires a range of hyperparameters including four optional preprocessing methods, three optional feature weighting methods, three optional oversampling methods and machine learning specific hyperparameters, c.f. Table 5. This leads to a total of 25,210 different models. Each model is trained and evaluated using fourfold cross-validation. The quality of the model is measured in terms of Matthews correlation coefficient (MCC) which takes the imbalance of the dataset into account. The evaluation of these models in Section 5.2 provides a first answer to Question 1.

Observe that, from the point of view of a repository owner, it is mandatory to evaluate the (most promising) models on the complete npm package repository (which consists of more than 1.9 million packages [10]). Note that the latest version of the npm repository does not include any package that is known to be malicious. Therefore, we do not have any meaningful labels and it is impossible to train the models on the full npm repository.

In order to answer Question 1 under the constraints mentioned above, we proceed as follows. We begin with selecting the three most promising models, based on the evaluation on the first dataset. Using the confidence of each model, we pick the top N packages and declare them to be potentially malicious. The number N is chosen such that the package owners security analysts are capable to further analyze these packages. In this setting, we declare a package to be a True Positive if it is both potentially malicious and if the analyst finds malicious behavior. Analogously, we declare a package to be a False Positive if it is potentially malicious but the analyst is not able to find any malicious code. Note that a manual inspection of all npm packages is impossible. With this methodology at hand, the three models are evaluated on the first dataset as well as on the full npm repository, see Section 5.3. This effort led to the detection and reporting of 13 previously unknown malicious packages. In particular, we provide a second, more realistic answer to Question 1.

The answers to Question 1 provide efficient models to detect malicious software packages. Recall that, in a realistic scenario, the top N packages are selected for manual inspection. It is no surprise that this leads to a high number of False Positives (in the sense of the above paragraph). By comparing the set of potentially malicious packages for each of the three best performing models, the False Positives are reduced by 93 % while the True Positives are reduced by only 12 %, see Section 5.2. This provides an interesting and practically relevant answer to Question 2.

4 DATASET & FEATURES

As stated in the Section 3, it is mandatory to have a dataset of known malicious and known benign packages as well as a set of suitable features in order to successfully train our supervised classifiers.

Due to their malicious nature, datasets of confirmed malicious packages are not easily available. The only dataset available to us was the "Backstabber's Knife Collection" [12], therefore it was chosen to provide the malicious packages for training. The state from January 25th 2021 was used. At the time, it contained 150 different (i.e., not counting multiple versions) malicious npm packages in total.

A set of known benign packages does not exist. However, we believe that due to their popularity, the packages that most other packages depend on have a high probability to be benign. Therefore, the 15,000 most depended upon packages in the npm registry have been selected as the dataset of benign packages. The names of these packages have been downloaded from the npm registry on January 6th 2021, and the packages themselves have been downloaded on January 21st 2021. Some of these packages contained errors, such that we could only successfully extract the features for 12,390 benign packages. Note that the resulting labeled dataset is unbalanced (as is the unlabeled set of software packages in a package repository).

Features as listed in Table 1 are partially adopted from [5] (http, fs, child_process, and eval). The remaining features were chosen based on expert knowledge and statistical analysis of the before mentioned datasets extracting striking differences between benign and malicious. According to Section 3, some features are crafted specifically for Node.js/JavaScript packages from npm. However, these features are easily transferable to other programming languages as well.

In order to substantiate our choice of feature, we investigate the statistical distribution of values in the classes of benign and malicious packages. Even though the interquartile range of malicious

Table 1: The features utilized for the classification. Continuous values (Cont.) can be any floating point number, while boolean values (Bool) are either true or false.

Value	Short description
Cont.	Levenshtein distance to popular packages' names
Cont.	Average number of square brackets per file
Cont.	Size of the package in bytes
Cont.	Number of dependencies
Cont.	Number of development dependencies
Cont.	Number of JavaScript files
Cont.	Number of square brackets
Bool	Has install scripts
Bool	Contains an IP
Bool	Uses Base64 conversions
Bool	Contains Base64 strings
Bool	Creates buffers from an ASCII value array
Bool	Contains Bytestrings
Bool	Contains Domains
Bool	Uses `Buffer.from`
Bool	Uses `eval`
Bool	Uses `require('child_process')` in a JavaScript file
Bool	Uses `require('child_process')` in an install script
Bool	Uses `require('fs')` in a JavaScript file
Bool	Uses `require('fs')` in an install script
Bool	Uses `require('network')` in a JavaScript file
Bool	Uses `require('network')` in an install script
Bool	Accesses `process.env` in a JavaScript file
Bool	Accesses `process.env` in an install script
Bool	Contains suspicious strings

packages overlap with these of benign features, the median values for continuous features yield a noticeable difference as shown in Figure 1.

The average values per class for the boolean features is shown in Figure 2. Again, a noticeable difference in the average values is visible. Overall, this indicates the general suitability of our selected features to distinguish the classes of benign and malicious packages.

5 RESULTS

This section displays and discusses the results of our conducted experiments. They follow the methodology described in Section 3 as well as the dataset and features listed in Section 4.

Firstly, on the labeled dataset, we evaluate a diverse set of supervised machine learning classifiers quantitatively to determine the three best performing ones, see Section 5.1. Recall that we aim to assist an expert by highlighting software packages for manual inspection. To this end, we evaluate (a combination) of the three models to predict a fixed amount of $N = 50$ and $N = 1000$ packages as malicious. The respective experiments on the labeled dataset are found in Section 5.2. For the evaluation on all available packages in the npm package repository, see Section 5.3.

5.1 Comparison of Diverse Classifiers

In this paper, we aim to assist an analyst by highlighting software packages that are supposed to be malicious for manual inspection. In the experiment described in this section, we evaluate the detection quality of a diverse set of machine learning techniques (including a combination of preprocessing steps) on the labeled dataset. This leads to a total of 25,210 different models. For the subsequent experiments in Section 5.2 and Section 5.3, we choose the three best performing models.

In order to measure the prediction quality of the models, we ask which packages are malicious and study the Precision, the Recall and the Matthews Correlation Coefficient (MCC). In this light, a

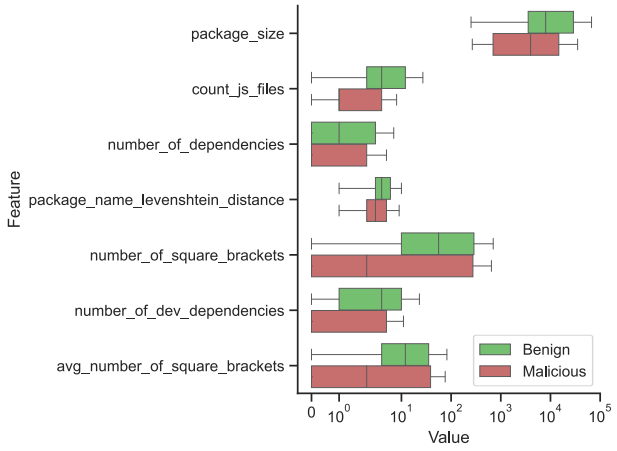

Figure 1: Boxplot for continuous features using log scale on the x-axis. All feature show a noticeable difference in the median value when comparing benign and malicious packages.

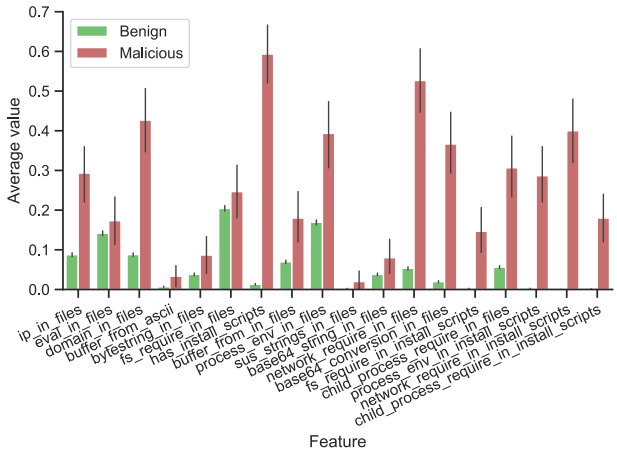

Figure 2: Barplot of average values for boolean features. Again a noticeable difference in the average value per feature for benign and malicious is visible.

package that is predicted to be malicious is likely to be malicious if the Precision is high (i.e., the ratio of True Positives among all Positives is high). Analogously, a malicious package is likely to be recognized if the Recall is high (i.e., the ratio of True Positives among all malicious packages is high). In order to account for the imbalance of the labeled dataset, the MCC is the preferred quality measure.

We train and evaluate multiple supervised machine learning classifiers on our labeled dataset. The collection of classifiers comprises AutoEncoder (AE), density-based classification (DB), Naive Bayes (NB), Linear Support Vector Machine (SVM), Kernel SVM, Multi Layer Perceptron (MLP), and Random Forest (RF). For more details about the leveraged classifiers and their hyperparameters, see Table 5 in the Appendix. This selection is founded on both the goals and characteristics of the data at hand. In particular, we choose AE and DB based on the presumably anomalous nature of the malicious instances, SVM, RF, and NB, due to their interpretability – especially with respect to feature importance – and lastly, MLP and Kernel SVM based on their greater expressive prowess.

As each classifier imposes distinct requirements regarding data embedding, we employ a range of different preprocessing methods, including oversampling. Latter is motivated by the – by nature of the scenario – high class imbalance. The preprocessing methods can be separated into data scaling and feature extraction/weighting. For the former, we evaluated None, Min-Max Scaling, Standardization, Normalization by Mean, and Robust Scaling. The latter is represented by Bone, t-SNE, PCA, and Spearman Correlation. With regard to oversampling, we test the usage of SMOTE, ADASYN, Borderline SMOTE, as well as the application of no oversampling.

The preprocessing methods are able to generate a broad spectrum of different embeddings, particularly with regard to the handling of anomalies. Due to the anomalous nature of the malicious instances, it is important to choose an approach that preserves this characteristic most optimal for the respective classifier. In return, this enables the establishment of clear conclusions concerning the classifiers' performances in comparison to each other.

Table 2 lists several metrics for each classifier based on fourfold cross-validation. As highlighted, the Kernel SVM yields a perfect Precision (1.0), meaning that it if it classifies a package as malicious it indeed is malicious. NB and RF both share the best Recall (0.63), which means that if a package is malicious it will most probably be classified as malicious by them. Our use-case scenario is the assistance of an analyst by pre-selection of packages for manual inspection; as such, solely high Recall is not sufficient. Instead, we also require a low number of False Positives. Hence, RF is a better fit for us than NB. In accordance with our observations in Figure 2 and Figure 1, the absence of a clear outlier status of the malicious instances results in subpar performances of both DB and AE.

In order to select the best performing classifiers, we leverage the Matthews correlation coefficient (MCC) which better factors in the imbalance of the dataset compared to the frequently used F_1-score. For both metrics RF, MLP, and Kernel SVM perform best. Thus, following experiments solely concern these three classifiers.

Seeing classifiers with either high Recall or high Precision already indicates the need for combination of multiple classifiers. This observation is investigated further in the following sections.

Table 2: Performance of all evaluated supervised machine learning classifiers on the labeled dataset using fourfold cross-validation.

Classifier	*TP*	*FP*	*TN*	*FN*	*Prec.*	*Rec.*	*MCC*
AE	14	18	3080	24	0.44	0.37	0.40
DB	10	2	3096	28	0.83	0.26	0.47
SVM (L)	15	1	3097	23	0.94	0.39	0.61
SVM (K)	20	0	3098	18	1.00	0.53	0.72
MLP	22	2	3096	16	0.92	0.58	0.73
NB	24	68	3030	14	0.26	0.63	0.40
RF	24	1	3097	14	0.96	0.63	0.78

5.2 Results on Dataset

Using RF, MLP, and Kernel SVM*, we train them again and evaluate their practicality. We repeat this experiment on the dataset of known malicious and benign packages in order to obtain a holistic impression of the classifies' performance on real world (unlabeled) data. Having ground truth at hand, we are able to make statements about the classifiers' Precision and Recall, i.e., how well they are suited to correctly identify malicious packages when mixed with benign ones. To this end, we split the dataset into training and evaluation with a ratio of 75:25 four times such that we can classify all 150 known malicious packages but never have them in the training and test set. In other words, the union over the four test sets is empty. Instead of comparing the classification of the three classifiers to the ground truth in total, we solely look at the N packages per classifier that are assigned the highest confidence by the classifier for the malicious class as well as the intersections of classifications. This is done to generate high confidence preselections of probably malicious packages for further manual analysis.

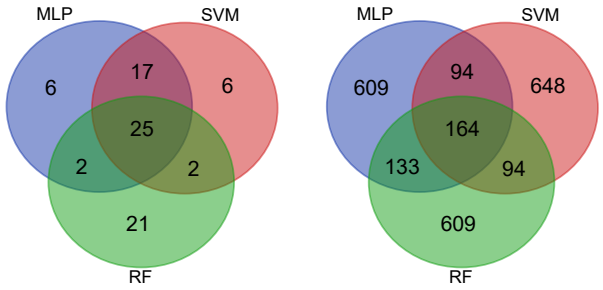

Figure 3: Intersections of the sets of packages labeled as malicious by MLP, SVM, and RF on the ground truth dataset. Left for $N = 50$ and right for $N = 1000$.

These intersections are visualized in Figure 3. On the left side, results for $N = 50$ and on the right side for $N = 1000$ are shown. It is visible that MLP and SVM have a great common share when $N = 50$. For $N = 1000$, this shifts to RF and MLP having more classifications in common. Overall, there is a considerable agreement by these three models.

*From now on just referred to as SVM.

Table 3: Performance of each classifier alone as well as for the intersections with other classifiers on the ground truth dataset. Measured for $N = 50$ and $N = 1000$.

	N = 50							N = 1000						
Classifier	*TP*	*FP*	*TN*	*FN*	*Prec.*	*Rec.*	*MCC*	*TP*	*FP*	*TN*	*FN*	*Prec.*	*Rec.*	*MCC*
MLP	50	0	12,390	100	1.00	0.33	0.58	109	891	11,499	41	0.11	0.73	0.26
SVM	50	0	12,390	100	1.00	0.33	0.58	108	892	11,498	42	0.11	0.72	0.26
RF	50	0	12,390	100	1.00	0.33	0.58	117	883	11,507	33	0.12	0.78	0.28
MLP ∩ SVM	42	0	12,390	108	1.00	0.28	0.53	103	155	12,235	47	0.40	0.69	0.52
MLP ∩ RF	27	0	12,390	123	1.00	0.18	0.42	104	193	12,197	46	0.35	0.69	0.48
SVM ∩ RF	27	0	12,390	123	1.00	0.18	0.42	106	152	12,238	44	0.41	0.71	0.53
SVM ∩ RF ∩ MLP	25	0	12,390	125	1.00	0.17	0.41	102	62	12,328	48	0.62	0.68	0.65

Furthermore, Table 3 lists before mentioned metrics in order to assess the quality of the preselections. It is noticeable that for $N = 50$ (left side of Table 3) we achieve a Precision of 100 %, i.e., the 50 packages with the highest confidence are indeed malicious. However, as we know that there are 150 malicious packages in the dataset, we miss a high number of relevant selections. Thus, the classifiers limited to $N = 50$ yield a comparable low Recall. Moreover, the combination of multiple classifiers leads to worse results as the intersection further limits the amount of preselected packages.

When looking at $N = 1000$ (right side of Table 3), we see an improved amount of True Positives but also a drastic increase in the numbers of False Positives. Each classifier on its one yields a bad Precision with roughly 90 % False Positives and only around 10 % of the packages preselected being True Positives. Just like in the evaluation of all classifiers in Section 5.1, RF yields the best Recall (0.78), i.e., the lowest number of False Negatives.

If we solely look at the intersection between multiple classifiers, we see an improved ratio of True Positives and False Positives. The best combination of classifiers is the intersection of all three (SVM ∩ RF ∩ MLP). By doing so, we get a very reduced number of preselected packages (-83.6 %) with a high number of them being indeed malicious (62.2 %) and only few being False Positives (37.8 %). Overall, this combination yields the highest Precision and MCC.

Based on these results, we arrive at two conclusions. Firstly, an appropriate threshold N needs to be determined. If chosen small, the preselected packages are malicious with high confidence but many non-selected packages may be missed. If chosen high(er), False Positives start to mix in and one needs two improve the preselection. That brings us to the second observation, that it seems useful to solely consider the combination of multiple classifiers as preselection for further manual analysis. This way, packages are preselected for which each classifier supports the classification with varying confidence. Let us remark that a suitable choice of N is not readily derived from the experiment but should be selected such that the analyst(s) are able to inspect the suggested stream of potentially harmful packages.

In the upcoming subsection, we investigate how the presented classifiers perform on real world data taken from npm.

5.3 Results on Real World Data

In this section, we perform our next experiment. It is conducted similarly to the previous experiment, however, we do not have a ground truth at hand. More precisely, we need to verify classifications manually. Since classifying 1.9 million packages is out of scope, we are not able to make statements considering False Negatives and True Negatives. Let us remark again that N has to be decided such that the analyst(s) are able to inspect the suggested stream of potentially harmful packages (which is typically far less than N). Here, we present our evaluation for $N = 50$ and $N = 1000$.

Again, the intersection of our classifiers for $N = 50^*$ and for $N = 1000$ is visualized in Figure 4. This time, there is no common agreement between all three classifiers for $N = 50$. Similarly, the pairwise intersection of two classifiers is also lower than when tested on the labeled dataset. This indicates that the classifiers work independently.

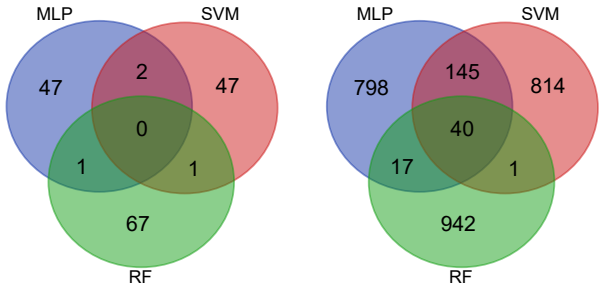

Figure 4: Intersections of the sets of packages labeled as malicious by MLP, SVM, and RF for all packages from npm. Left for $N = 50$ and right for $N = 1000$.

When setting $N = 1000$, the situation changes and a common agreement of all three classifiers exists. Again, MLP and SVM have a great common share similar to $N = 50$ on the labeled dataset. Table 4 lists all metrics considering True Positives and False Positives identified through manual inspection of the package.

While analyzing the packages in question by hand, we encountered packages that syntactically seem malicious, i.e., similar code

*Note that we set $N = 69$ for RF because they all shared a confidence value of 1.0 and hence any limit to $N = 50$ would be arbitrary.

Table 4: Performance of each classifier alone as well as for the intersections with other classifiers for all packages from npm. Measured for $N = 50$ and $N = 1000$. Here, X_{wor} states the metric X when removing research packages from the statistics.

	N = 50						N = 1000					
Classifier	TP	TPR	FP	TP_{wor}	TPR_{wor}	FP_{wor}	TP	TPR	FP	TP_{wor}	TPR_{wor}	FP_{wor}
MLP	28	56 %	22	3	6 %	47	-	-	-	-	-	-
SVM	8	16 %	42	3	6 %	47	-	-	-	-	-	-
RF	10	14.5 %	59	7	14 %	62	-	-	-	-	-	-
MLP ∩ SVM	0	0 %	2	0	0 %	2	37	20 %	148	9	4.9 %	176
MLP ∩ RF	1	100 %	0	1	100 %	0	40	70.2 %	17	6	10.5 %	51
SVM ∩ RF	1	100 %	0	1	100 %	0	26	65 %	15	4	9.8 %	37
SVM ∩ RF ∩ MLP	-	-	-	-	-	-	26	63.4 %	14	4	10 %	36

and procedure as known malicious packages, but without the intention to cause real harm to the user. Often, these packages contained a note in the package description stating their research nature. Looking strictly at the syntax of the packages, they have correctly been labeled as malicious by our classifiers. However, when taking the semantic into consideration, we would need to handle them separately. We named these packages *research packages* and adopted our evaluation to show the performance with and without them. Table 4 lists the metrics without research packages with a minor *wor* as index, e.g., TP_{wor}, and in text we add these metrics in brackets.

Looking at $N = 50$ in Table 4, we see a promising True Positive Rate (TPR) for MLP of 56 % (6 %). RF achieves a TPR of 14.5 % but even after factoring out research packages it is at 14 %. This indicates that RF is able to separate truly malicious packages from research packages. Considering combination of multiple classifiers yields bad results for MLP ∩ SVM and an empty set for SVM ∩ RF ∩ MLP. For both MLP ∩ RF and SVM ∩ RF the only package in the intersection is correctly classified as malicious.

When inspecting packages with $N = 1000$, we solely look at the intersections as we found out in Section 5.2 that these yield a good preselection when dealing with a higher number of potential malicious packages.

It is noticeable that MLP ∩ SVM yields the highest number of packages to inspect (185) while also achieving the worst TPR of 20 % (4.9 %). The combinations MLP ∩ RF and SVM ∩ RF yield considerable better results by preselecting only 57 and respectively 41 packages. The former combination achieves a TPR of 70.2 % (10.5 %) while the latter combination achieves a TPR of 65 % (9.8 %). The agreement of all three classifiers yields a good average with 40 preselected packages and a TPR of 63.4 % (10 %).

In general, related work often suffers from a high number of False Positives and/or the need for a lot of manual optimization. We designed our solution such that it yields a feasible amount of preselected packages that have a high chance of being actually malicious. Overall, we were able to identify and report 13 actual malicious packages to the security team of npm. All of them have been replaced by a so-called "security holding package" to stop further misuse.

Our results may be compared best to Sejfia et al. [18]. In their work, they analyzed three different classifiers namely: Decision Tree, Naive Bayesian, and linear SVM. We obtained the best results in our experiments with RF, MLP, and Kernel SVM. However, the performances of classifiers are comparable. Our RF reaches a Precision of 0.96 and Recall of 0.63, which yields a slightly better Recall than their Decision Tree (Precision 0.98 and Recall 0.43). The NB trained by us yields worse results with a Precision of 0.26 and a Recall of 0.63 where theirs reaches a Precision of 0.90 and a Recall of 0.19. Moreover, our linear SVM yields a Precision of 0.94 and a Recall of 0.39 which again is a slightly better Recall than their counterpart with a Precision of 0.98 and a Recall of 0.27. However, our Kernel SVM outperforms both with a Precision of 1.00 and a Recall of 0.53. It must be noted that this comparison is done using two different datasets and any divergence might stem from different training and test data.

Considering the good suitability of our approach, it may be deployed by operators of package repositories in order to check all packages (possibly periodically after each training of the classifiers) or any new package and updated version of existing packages before making them publicly available.

6 DISCUSSION

This section discusses our assumptions and observations.

In the compilation of the labeled dataset, we assume that the 15,000 most depended upon packages on npm are benign. Let us remark two observations. Firstly, there might be undetected malicious code contained in one of these packages. Since machine learning classifier are designed to find the sweet spot to identify the majority of a class, single outliers, i.e., malicious code, may not affect the final model. Secondly, these 15,000 packages might not represent the average npm package. Seeing a high TPR (low FP, c.f. Table 4) for most of our models indicates that this is not of a problem. However, one could randomly sample packages from npm instead which again complicates the assumption of these packages being benign.

Our dataset is unbalanced with a ratio of malicious to benign of 1:100. Assuming that roughly 1 % of npm packages is malicious is — hopefully — too high. In general, we leveraged a low number of samples for training because there are currently no more at hand. Thus, we tested several oversampling methods. Except for the Random Forest classifier, all oversampling methods lead to significantly worse results. Random Forest is by design largely invariant to feature scale and sample distribution. Hence, the indifference to the application of oversampling is to be expected. Additionally,

we can not observe any correlation between oversampling and preprocessing methods.

As oversampling is exclusively applied to the respective training data, the performance decrease can not be attributed to an impact on the performance metric. Accordingly, oversampling actively hinders the training process. This can be explained by the feature value distribution. As visible in Figure 1 and Figure 2, the feature values for malicious instances are significantly broader spread than their benign counterpart. In combination with the low number of genuine malicious instances, this leads to overfitting when oversampling is applied, thus resulting in a sharp drop in the classification performance on the validation data. As such, to improve our classifiers' results, training on a bigger dataset can not be substituted by oversampling.

We consider a variety of different preprocessing methods to provide adequate data transformation for all classifiers. Except for the Random Forest, which is — as mentioned above — largely invariant to feature scale, we can observe a clear pattern concerning data scaling with respect to all classifiers. Each profit from preprocessing in some form, with either standardization or min-max scaling resulting in the best performance. This is in accordance with our observations visualized in Figure 1, in particular, the noticeable difference in the median but lack thereof with regard to the minimum and maximum values. We fail to detect any substantial dependence on feature weighting or extraction, except SVM, where we can observe a slight preference toward t-SNE. The embedding provides a class separation not obtainable by the RBF kernel of the SVM, consequently outweighing the loss of information caused by the embedding into a lower-dimensional space.

When choosing an appropriate N, to preselect packages for manual analysis, we solely looked at $N = 50$ and $N = 1000$. For real world use, these values might be adopted to the available work force and a manageable number of False Positives. There may be suitable combinations other than intersections of high confidence predictions. However, we found these to work well.

Furthermore, we investigated why benign packages were falsely labeled as malicious. It was noticeable that RF prefers small packages with a low number of files. Most often, the preselected packages by RF comprised just one or two files and an archive size of below 500 Bytes. This peculiarity is also present in packages that are uploaded as typosquatting prevention placeholders, deprecation warnings, and contact information of developers. All of these typically contain a function that prints a corresponding message to the terminal on installation.

Another more general observation is that our classifiers will label packages that are written in TypeScript or CoffeeScript as malicious. Both languages are related to JavaScript but need to be compiled into JavaScript syntax before use and sometimes during installation. For our classifiers, it seems like an install script executes an external process which is frequently seen in malicious packages (cf. Figure 2). Overall, our features do not cover TypeScript or CoffeeScript.

Moreover, some packages may be considered as wrappers. These packages are used to distribute and install other programs through npm. On installation, these packages download and install external programs which again resembles malicious behavior.

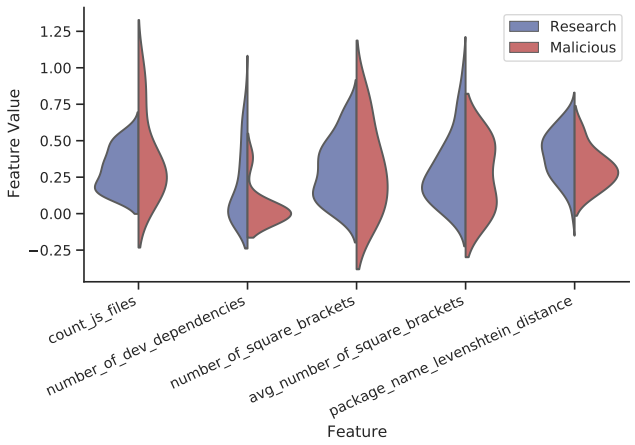

Figure 5: Violinplot showing difference in feature value distribution between benign and research for the five most notable features. All feature values are log-normalized and min-max scaled.

Figure 5 visualizes the differences in feature value distribution depending on the packages we established as research and malicious, respectively (employing the intersection of all classifiers' predictions). While most feature a near-identical distribution, the depicted five features showcase notable dissimilarities. In particular, we can observe a greater level of uniformity for the research packages. Especially, the differences in the number of square brackets and JavaScript files indicate a significantly more extensive spread of project scope for the malicious packages. This enables a Random Forest classifier to differentiate between the two classes.

7 CONCLUSION & FUTURE WORK

The more software dependencies a software comprises, the larger the Software Supply Chain is and the higher the chance are to fall victim to a Software Supply Chain attack through a malicious software package. Package repositories like npm, PyPI, or RubyGems have grown over the recent years and became a neuralgic point of the open source ecosystem. Malevolent actors leveraged the fact that these repositories are typically not curated and may serve as large-scale malware distribution platforms.

In this paper, we investigated the feasibility of supervised machine learning classifiers for the detection of malicious software packages. To this end, we compiled a dataset of 150 known malicious and 15,000 benign packages from npm. A total of 25,210 supervised machine learning models were trained on this dataset and according to our proposed features.

A comparison of the diverse set of classifiers trained, revealed the suitability of Kernel Support Vector Machines (SVM), Multi Layer Perceptrons (MLP), and Random Forests (RF). These were further evaluated for their practicability to preselect potential malicious packages for further manual analysis.

In our experiments, we showed that the combination of multiple classifiers can yield a suitable preselection with a True Positive Rate

of up to 70 %. Consequentially, we manually verified and reported 13 previously unknown malicious packages.

Moreover, our experiments revealed a new class of research packages that are syntactically almost indistinguishable from malicious packages as they solely differ in their semantic, i.e., no intention to cause harm. Currently, this extra class can only be separated manually. However, we see the handling of such research packages as an organizational task on the side of package repositories.

In conclusion, we were able to show that supervised machine learning classifiers can be utilized to detect malicious packages at large scale by drastically reducing the manual effort. Combinations of diverse classifiers yield good True Positives with a reasonable amount of False Positives.

For future work we would like to transfer our findings onto other programming languages like Python and Ruby. Moreover, we want to answer whether ensemble learning is able to improve the detection quality further. Furthermore, a more sophisticated preselection method other than based on confidence and intersections might be possible.

ACKNOWLEDGMENTS

We would like to thank Timo Pohl for his expertise and support in feature engineering as well as dataset acquisition and preparation.

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

[1] Justin Cappos, Justin Samuel, Scott Baker, and John H Hartman. 2008. A look in the mirror: Attacks on package managers. In *Proceedings of the 15th ACM conference on Computer and communications security*. 565–574.

[2] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*. 181–191.

[3] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2020. Measuring and preventing supply chain attacks on package managers. *arXiv e-prints* (2020), arXiv–2002.

[4] The Apache Software Foundation. 2022. *Apache Log4j Security Vulnerabilities*. https://logging.apache.org/log4j/2.x/security.html

[5] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2019. Detecting suspicious package updates. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 13–16.

[6] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schäfer. 2021. Anomalicious: Automated Detection of Anomalous and Potentially Malicious Commits on GitHub. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 258–267.

[7] Mike Hanley. 2021. *GitHub's commitment to npm ecosystem security.* https://github.blog/2021-11-15-githubs-commitment-to-npm-ecosystem-security/

[8] Berkay Kaplan and Jingyu Qian. 2021. A Survey on Common Threats in npm and PyPi Registries. In *International Workshop on Deployable Machine Learning for Security Defense*. Springer, 132–156.

[9] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem. *arXiv preprint arXiv:2201.03981* (2022).

[10] npm. 2022. *The complete list of all npm software packages.* https://replicate.npmjs.com/_all_docs

[11] Marc Ohm, Lukas Kempf, Felix Boes, and Michael Meier. 2022. Towards Detection of Malicious Software Packages Through Code Reuse by Malevolent Actors. In *Sicherheit 2022*.

[12] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. 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, 23–43.

[13] Marc Ohm, Arnold Sykosch, and Michael Meier. 2020. Towards detection of software supply chain attacks by forensic artifacts. In *Proceedings of the 15th international conference on availability, reliability and security*. 1–6.

[14] Brian Pfretzschner and Lotfi ben Othmane. 2017. Identification of dependency-based attacks on node. js. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*. 1–6.

[15] PyPI Warehouse. 2022. *Malware Checks.* https://warehouse.pypa.io/development/malware-checks.html

[16] Jason D. M. Rennie, Lawrence Shih, Jaime Teevan, and David R. Karger. 2003. Tackling the Poor Assumptions of Naive Bayes Text Classifiers. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning* (Washington, DC, USA) *(ICML'03)*. AAAI Press, 616–623. https://doi.org/10.5555/3041838.3041916

[17] Iqbal H Sarker, Md Hasan Furhad, and Raza Nowrozy. 2021. Ai-driven cybersecurity: an overview, security intelligence modeling and research directions. *SN Computer Science* 2, 3 (2021), 1–18.

[18] Adriana Sejfia and Max Schäfer. 2022. Practical Automated Detection of Malicious npm Packages. *arXiv preprint arXiv:2202.13953* (2022).

[19] Ax Sharma. 2022. *Dev corrupts NPM libs 'colors' and 'faker' breaking thousands of apps.* https://www.bleepingcomputer.com/news/security/dev-corrupts-npm-libs-colors-and-faker-breaking-thousands-of-apps/

[20] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending against package typosquatting. In *International Conference on Network and System Security*. Springer, 112–131.

[21] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. 2019. Survey of machine learning techniques for malware analysis. *Computers & Security* 81 (2019), 123–147.

[22] Ruturaj K Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. 2019. Security issues in language-based sofware ecosystems. *arXiv preprint arXiv:1903.02613* (2019).

[23] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. Lastpymile: identifying the discrepancy between sources and packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 780–792.

[24] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Typosquatting and combosquatting attacks on the python ecosystem. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 509–514.

[25] Miao Xie and Jiankun Hu. 2013. Evaluating host-based anomaly detection systems: A preliminary analysis of ADFA-LD. *Proceedings of the 2013 6th International Congress on Image and Signal Processing, CISP 2013* 3, 1711–1716. https://doi.org/10.1109/CISP.2013.6743952

[26] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. 995–1010.

A MACHINE LEARNING METHODS AND HYPERPARAMETERS

Table 5: Here shown are all machine learning methods employed, including a short description denoting significant changes/adaptions of the default approach if present. The hyperparameter column lists all evaluated parameter values for the respective classifiers with the best performing underlined.

Method	Description	Hyperparameter
Autoencoder	Classic autoencoder, featuring 5 layers for encoder and decoder each (activation function: leaky ReLU). The labeling is performed employing a reconstruction error threshold based on a predefined percentage of expected anomalies.	Evaluated parameters: • *Learning Rate:* 3 values sampled uniformly from interval $[0.01, 0.2]$ (best: $\underline{0.0399}$) • *Number of units (first layer):* $(128, \underline{256})$ • *Number of units (fecond layer):* $(64, \underline{100})$ • *Number of units (fhird layer):* $(25, 32, \underline{50})$ • *Number of units (fourth layer):* $(10, \underline{16}, 20)$ • *Number of units (fifth layer):* $(2, 4, \underline{8})$ • *Expected anomalies percentage:* $(\underline{1.0\%}, 2.5\%, 5.0\%)$
Complement Naive Bayes	Following the implementation given by Rennie et al. [16].	Evaluated parameters: • α*:* $\underline{1.0}$
Density-based	Instance is labeled as anomaly if it contains less then k training instances in a predefined *radius* [25].	Evaluated parameters: • k*:* $(5, 10, 15, 20, 25, 30, \underline{50})$ • *radius:* $(0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.5, \underline{2.0})$
Kernel SVM	Support vector machine employing an RBF kernel.	Evaluated parameters: • *Gamma:* 3 values sampled employing a normal distribuation with mean 0.2 and standard deviation 0.075 (best: $\underline{0.3218}$) • C*:* 3 values sampled uniformly from interval $[0.5, 2.0]$ (best: $\underline{1.2928}$)
Linear SVM	—	Evaluated parameters: • C*:* 6 values sampled uniformly from interval $[0.5, 2.0]$ (best: $\underline{1.5913}$)
MLP	Simple fully connected neural network with one hidden layer (activation function: sigmoid).	Evaluated parameters: • *Learning Rate:* 5 values sampled uniformly from interval $[0.01, 0.2]$ (best: $\underline{0.1485}$) • *Number of hidden units:* $(16, \underline{32}, 64)$
Random Forest	—	Evaluated parameters: • *Number of trees:* $(\underline{16}, 32, 64, 128, 256, 512)$