On the Feasibility of Cross-Language Detection of Malicious Packages in npm and PyPI

Piergiorgio Ladisa
SAP Security Research
Mougins, France
University of Rennes 1/INRIA/IRISA
Rennes, France
piergiorgio.ladisa@sap.com
piergiorgio.ladisa@irisa.fr

Serena Elisa Ponta
SAP Security Research
Mougins, France
serena.ponta@sap.com

Nicola Ronzoni
SAP Security Research
Mougins, France
nicola.ronzoni@sap.com

Matias Martinez
Universitat Politècnica de Catalunya - Barcelona Tech
Barcelona, Spain
matias.martinez@upc.edu

Olivier Barais
University of Rennes 1/INRIA/IRISA
Rennes, France
olivier.barais@irisa.fr

ABSTRACT

Current software supply chains heavily rely on open-source packages hosted in public repositories. Given the popularity of ecosystems like npm and PyPI, malicious users started to spread malware by publishing open-source packages containing malicious code.

Recent works apply machine learning techniques to detect malicious packages in the npm ecosystem. However, the scarcity of samples poses a challenge to the application of machine learning techniques in other ecosystems. Despite the differences between JavaScript and Python, the open-source software supply chain attacks targeting such languages show noticeable similarities (e.g., use of installation scripts, obfuscated strings, URLs).

In this paper, we present a novel approach that involves a set of language-independent features and the training of models capable of detecting malicious packages in npm and PyPI by capturing their commonalities. This methodology allows us to train models on a diverse dataset encompassing multiple languages, thereby overcoming the challenge of limited sample availability. We evaluate the models both in a controlled experiment (where labels of data are known) and in the wild by scanning newly uploaded packages for both npm and PyPI for 10 days.

We find that our approach successfully detects malicious packages for both npm and PyPI. Over an analysis of 31,292 packages, we reported 58 previously unknown malicious packages (38 for npm and 20 for PyPI), which were consequently removed from the respective repositories.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

Open-Source Security, Supply Chain Attacks, Malware Detection

ACM Reference Format:
Piergiorgio Ladisa, Serena Elisa Ponta, Nicola Ronzoni, Matias Martinez, and Olivier Barais. 2023. On the Feasibility of Cross-Language Detection of Malicious Packages in npm and PyPI. In *Annual Computer Security Applications Conference (ACSAC '23), December 04–08, 2023, Austin, TX, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3627106.3627138

1 INTRODUCTION

From large companies to independent developers, the current way of producing software is characterized by a large consumption of open-source packages. Given the popularity of some programming languages, package repositories and package managers have been created for specific ecosystems to ease the consumption of Open-Source Software (OSS) for downstream users. Package repositories (e.g., PyPI, npm) are public databases that can be queried to retrieve packages implementing certain tasks. On the client side, package managers automatically resolve and install the required packages as well as their dependencies.

While these mechanisms facilitate software modularization and increase the speed of implementation, malicious users started to exploit these mechanisms as an easy way to spread malwares on a large scale. The attack surface of the OSS supply chain is large and attackers have multiple attack vectors at their disposal to conduct *open-source software supply chain attacks* [20]. Given that most companies (both private and public) use open-source [4], enhancing the security of the software supply chain has become a priority in the interest of both the community and nations security [11, 18, 32].

One way to counter OSS supply chain attacks is to detect the presence of malicious behavior in packages consumed from package repositories. Recent works propose Machine Learning (ML) techniques for vetting packages in npm [24, 30]. The primary challenge in applying ML approaches to detect malicious packages lies in the availability of labeled data. Currently, the widely used dataset for malicious packages is Backstabber's Knife Collection (BKC) [2], which comprises 102 distinct malicious packages for JavaScript and 92 for Python (as of October 2022).

Though every language has its own characteristics (e.g., a particular list of sensitive APIs), we observe that malicious packages from the Python and JavaScript ecosystems present common patterns. As examples, the code snippets responsible for the malicious behavior are usually characterized by the presence of obfuscated strings and the usage of installation scripts.

In this preliminary work, we evaluate the feasibility of using a machine-learning approach to capture similarities of malicious behaviors across different languages, specifically Python and JavaScript. This entails constructing a *cross-language* classifier using a dataset featuring samples from both languages, as well as developing two separate *mono-language* classifiers trained on datasets comprising samples from either JavaScript or Python.

We set out to answer the following research questions:

RQ1: Which cross-language and mono-language models demonstrate the best trade-off between precision and recall when detecting malicious packages in the case of JavaScript and Python languages?

RQ2: How do the models identified in RQ1 perform in the detection of potentially malicious packages in the wild?

Our work's technical contributions in addressing these questions are as follows.

Firstly, we propose the use of 141 language-independent features, derived from a combination of previous works [24, 30] as well as expert knowledge.

Secondly, we evaluate tree-based learning algorithms, specifically Decision Tree (DT), Random Forest (RF), and eXtreme Gradient Boosting (XGBoost), to train three classifiers for distinct datasets: one comprising solely JavaScript packages, another containing only Python packages, and a third containing packages from both languages. Out of the 9 models generated, we discovered that both the cross-language and mono-language models based on XGBoost performed best in a controlled experiment.

Thirdly, we conduct a real-world assessment of these models by analyzing packages uploaded daily to npm and PyPI over a 10-day period. This experiment revealed that the cross-language model outperformed the respective mono-language models.

Lastly, throughout our analysis involving 31,292 packages, we successfully identified 58 previously unidentified malicious packages across npm and PyPI. We promptly reported these malicious discoveries to the respective repositories, which confirmed our findings. Moreover, we enhanced BKC[1] by uploading these newly identified malicious packages.

The paper is organized as follows. Section 2 provides an overview of the problem of Open Source Software (OSS) supply chain attacks and highlights the key distinctions between Python and JavaScript. Section 3 presents the datasets, the learning algorithms employed for training the models to classify malicious packages, and the set of features used. Section 4 answers the research questions. Section 5 examines the findings and noteworthy aspects of the npm and PyPI ecosystems observed during the real-world experiment. Section 6 delves into the processes involved in responsibly reporting the identified malicious packages. Section 7 outlines potential threats to the validity of the study. Section 8 presents related works within the domain of machine-learning approaches for detecting malicious packages. Section 9 summarizes the key conclusions drawn from the study and provides insights into future prospects.

[1]Commit `6d7d140168f80c048fc1622a4788b1f5c17af2d0`

2 OPEN-SOURCE SOFTWARE SUPPLY CHAIN ATTACKS

OSS supply chain attacks involve the introduction of malicious code into open-source components, which serves as a method for propagating malware among downstream consumers. Various attack vectors can be employed to carry out such attacks [20]. Broadly speaking, attackers may choose to: *develop and advertise a malicious package from scratch*, *create name confusion with legitimate package*, or *subvert a legitimate package* (i.e., injecting the malicious code in the source code, during the build, or within the package repository).

Regardless of the attack vector used, attackers aim to have their malicious package(s) hosted on popular distribution platforms such as npm and PyPI registries. Downstream users typically acquire OSS by downloading it from these platforms using dependency management systems like pip or the npm CLI. After victims download a malicious package, the malicious code contained within it can be triggered at various stages of the package lifecycle, including installation, runtime, or during testing [26].

In 2020, Ohm et al. [26] observed that in most cases (i.e., 56%) attackers inject malicious code in a manner that is triggered during the installation phase. One way to achieve this is by leveraging the capability of package managers to automatically execute custom *installation scripts* included within the downloaded packages. These scripts enable the specification of actions to be executed during the installation process. The designated entry point for these scripts is referred to as the *installation hook*.

In JavaScript, the *package.json* file provides installation hooks through the *scripts* property. This property is a dictionary where the keys represent lifecycle phases (e.g., *pre-install*), indicating when the scripts in the corresponding values should be executed [5].

In Python, the *setup.py* script is executed during the installation of source distributions (*sdist*). Attackers have the opportunity to inject malicious code anywhere within this file, allowing them to execute commands at installation time [1].

Attackers may also inject malicious code into other executable scripts within the package, aiming for its execution during runtime or tests whenever the manipulated functionality is triggered.

Depending on the specific attack vector employed by attackers, different safeguards can be implemented to partially or fully mitigate such attacks [20]. One effective protection measure involves detecting malicious packages prior to their consumption. The academic community has already begun proposing methods for identifying malicious packages [10, 15, 21, 24, 25, 27, 30]. A prominent challenge [13] for all detection approaches is the wide variety of evasion techniques that can be employed by malware. Therefore, it is crucial to enrich datasets of malicious packages (e.g., BKC) such that new strategies can be devised to detect malware categories akin to those already encountered or anticipate emerging types of malware.

3 MODELS TRAINING

In this section, we outline the training process for both the mono-language and cross-language models, which are designed to classify malicious packages. First, we describe the datasets we constructed, comprising benign and malicious packages in JavaScript and Python. Then, we delve into the algorithms and optimizations employed to obtain the models. Finally, we present the collection of language-independent features extracted from the packages.

3.1 Datasets

We build a labeled dataset of benign and malicious packages. Given that benign packages significantly outnumber malicious ones in real package repositories, we create an imbalanced dataset to reflect this reality. As the exact ratio of malicious to benign packages remains uncertain, we adopt the 90-10 ratio between benign and malicious samples, as recommended in previous studies [23, 31].

Language	Total # of samples	Filter by version	Filter by campaign	Filter by duplicates
JavaScript	2071	1505	1408	102
Python	273	225	133	92

Table 1: Number of malicious samples remaining after applying each filtering step.

Malicious samples. We use BKC [2], which is a collection of malicious OSS packages found in popular package repositories and voluntarily contributed by the community. At the time of writing (October 2022, commit `c2d9691`) BKC contains a total of 3,161 packages for different ecosystems: 2,071 JavaScript, 273 Python, 813 Ruby, and 4 Java packages. Since our focus is on JavaScript and Python, we restrict to packages for such ecosystems. It is noteworthy that a significant portion of the packages is associated with malware campaigns (i.e., packages having different names but containing the same malicious behavior [26]). Consequently, it is possible to encounter multiple duplicates within the BKC dataset. To avoid bias we remove duplicates from the dataset. We consider as duplicates packages that *(i)* have multiple versions, *(ii)* are marked as part of a campaign in BKC, and *(iii)* have the same values for the considered features (cf. Section 3.3). In the first case, we consider only the latest version. For packages belonging to the same campaign or having the same values for the features, we take only one sample. Table 1 shows the number of packages remaining after each filtering step. Finally, we get 102 malicious packages for npm and 92 for PyPI.

Regarding malicious behaviors, it is worth mentioning that our approach is limited to those present in BKC, i.e., reverse shell, dropper, data exfiltration, Denial of Service (DoS), and financial gain [26]. Therefore, additional behaviors (e.g., phishing campaigns [17]) are out of our scope.

Benign samples. Since no ground truth for benign packages exists, we build our dataset as follows.

Assuming that popular projects in npm and PyPI are free from malicious code [24], we aim at collecting these projects and use the popularity information provided by `libraries.io`[2] through the SourceRank score[3]. Since `libraries.io` offers APIs to search for packages by category (e.g., machine learning, math, UI) but not by popularity, we adopted the following approach to gather popular packages. First, we compiled a list of 150 categories from the ones suggested by the `libraries.io` UI for the most popular packages. Second, we searched for these keywords and sorted the results in descending order of popularity. Third, we selected the top-n projects for each ecosystem to maintain the desired 90-10 ratio (i.e., 828 for Python and 918 for JavaScript). Finally, we download the packages from the respective package repositories (i.e., npm for JavaScript and PyPI for Python).

Mono-language and Cross-Language Datasets. Once we have collected both malicious and benign packages, we proceed to construct various types of datasets. The two mono-language datasets contain packages in a single programming language: one for JavaScript and one for Python. The cross-language dataset consists of the union of the two mono-language datasets. Table 2 shows the number of packages contained in each dataset and related stratification in benign and malicious.

Type	Programming Language(s)	Malicious samples	Benign samples
Mono-language	JavaScript	102	918
Mono-language	Python	92	828
Cross-language	JavaScript+Python	194	1640

Table 2: Composition of both mono-language and cross-language datasets.

3.2 Algorithms Selection and Tuning

The main objective of our work is to explore the feasibility of cross-language detection of malicious packages. As mentioned earlier, malicious packages are assumed to be significantly fewer than benign ones. Therefore, we seek supervised classification algorithms that are well-suited for imbalanced datasets, requiring no preprocessing of the dataset itself. Furthermore, to gain more insights into the performances of the models, we look for algorithms that provide explainable predictions. Finally, due to the number of features considered, we only consider learning algorithms that handle high-dimensional data.

Algorithms that meet these criteria and that showed the best performances in recent works are Decision Tree (DT) [24, 30] and Random Forest (RF) [24]. In the context of tree-based algorithms, we also consider the XGBoost (XGB) algorithm [9], knowing that boosting algorithms are well-suited for imbalanced datasets [14]. We train the aforementioned classifiers using *scikit-learn*[4].

To fine-tune the selected algorithms for the classification problem we use the Bayesian Optimizer (BO) [33] in combination with 5-fold cross-validation to find the best combination of hyperparameters leading to the highest precision. We choose precision as the objective function to reduce the number of false positives, i.e., the number of samples that the security analyst has to manually review to confirm the classification. The value of the precision considered

[2]https://libraries.io/

[3]https://docs.libraries.io/overview.html#sourcerank

[4]https://scikit-learn.org

for the optimization problem is the average value obtained after the 5-fold cross-validation.

For the DT classifier, the hyperparameters optimized through the BO are the maximum depth of the tree, maximum number of features, split criterion (i.e., information gain, Gini index, log-loss), the minimum number of observations within the leaves, and minimum number of samples required to split an internal node. For the RF classifier, in addition to the ones for DT, we also optimize the number of estimators (i.e., decision trees) and the maximum number of samples to train each tree. Finally, for the XGBoost classifier, the hyperparameters to be optimized are the maximum depth of the tree, number of estimators, subsample ratio of features to construct each tree, learning rate, minimum split loss, and minimum sum of hessian needed in a child node of the tree.

3.3 Features

To identify the features of interest we inspect the malicious samples available in BKC [2]. Since our goal is to detect the presence of malicious code in more than one programming language, we do not consider features specifically crafted for a certain programming language (e.g., usage of certain APIs), but we focus on lexical and structural aspects. Thus, we adopt the features proposed in [24, 30] that have general applicability and are not specific to JavaScript (cf. Table 6). Then, we propose additional features based on expert knowledge.

The final set of features considered in our work is described in Table 3. Such features capture characteristics of strings, identifiers, file extensions, and the usage of installation hooks. Their independence from a specific language has the advantage that they do not need constant maintenance as it would be required for language-specific features, like a list of security-related APIs.

We utilize the Pygments lexer[5] to parse and process the source code files (*.js, .py*) as well as the installation scripts (*package.json, setup.py*) of the package being analyzed. From such files, we extract the following types of lexical tokens: strings, identifiers, operators, and punctuation.

In the following, we describe and motivate the choice of the different features.

Usage of installation hooks. As described in Section 2, the installation script is commonly used by attackers to trigger execution via installation hooks. Thus, we detect the presence of the *setup.py* file (for Python) and of the tokens *install, post-install, pre-install* in the *package.json* file (for JavaScript) using a boolean feature.

Code obfuscation. A property that often characterizes malicious strings is obfuscation. To detect obfuscation and encodings (e.g., base64) in both strings and identifiers, we leverage the Shannon entropy [22] combined with the concept of Generalization Language (GL) [19]. Generalization Languages are used in the context of error detection and consist of encoding data using a pre-defined alphabet to abstract specific values into a pattern. The observation made by Huang et al. [19] is that abstracting specific values into patterns overcomes data sparsity and makes co-occurrence of values more reliable to quantify compatibility (in our case the Shannon entropy) between patterns.

[5]https://pygments.org/

In our case, given a string or identifier (i.e., variable, function, and class names) as input, we transform every character x using the following mapping:

$$GL_4(x) = \begin{cases} \text{L if } x \text{ is a lowercase character} \\ \text{U if } x \text{ is an uppercase character} \\ \text{D if } x \text{ is a digit} \\ \text{S if } x \text{ is a symbol} \end{cases} \tag{1}$$

where GL_4 indicates a GL with an alphabet of four symbols. For example, the string *YmFzaA==* (base64 encoding of *bash*) becomes *ULULLUSS*, whereas a non-encoded string like *while* becomes *LLLLL*. Once the strings or the identifiers are transformed using GL_4, we compute the mean, standard deviation, 3rd quartile, and maximum value of Shannon entropy in both the cases of source code files and installation scripts. Also, for both strings and identifiers, we count those that are homogeneous (i.e., having all characters equal after GL_4 encoding) and heterogeneous (i.e., having at least one different character after GL_4 encoding).

Sensitive Strings. Malicious code generally introduces strings with certain properties (e.g., URLs, shell commands). To detect the presence of security-related strings, we adopt the approach presented in [22]. Thus, we build a dictionary of keywords from offensive security cheat sheets (e.g., reverse shells, the path to sensitive files). The keywords are included both in plain text and in different encodings (e.g., base64, base32, rot-13, URL-encoding). The corresponding feature consists of the count of hits to this dictionary.

Structural features of source code files and string manipulation. We capture the code size by counting the number of words and the number of lines of code.

For symbols commonly used to manipulate strings (i.e., plus sign, equal sign, and square brackets) we compute their ratio over the size of the file containing them and we report mean, standard deviation, 3rd quartile, and maximum value for all the files contained in the analyzed package.

Structural features of the package. Since malicious packages may execute malware from external resources to the running script (e.g., binaries, shell scripts), we build a custom list of 91 popular file extensions based on the ones appearing in the malicious packages contained in BKC. We report the count of files contained in the analyzed package for these extensions.

To support the choice of the features described above, we inspect their statistical distribution. In particular, we compare the distribution of the features when they are extracted from benign packages and malicious packages. This allows us to initially assess whether the proposed features are able to discriminate between the two classes. Figure 1 depicts the violin plots for some representative cases. Comparing the distributions in benign and malicious cases, we observe significant differences for most of the features. In the cases where the distributions are similar (e.g., No. of IP addresses in source code files), we still detect that at least one of the statistical measures (e.g., mean, median, minimum value) differs between the malicious and benign cases.

Type	Description	Captured Behavior
Boolean	Usage of installation hook(s)	Arbitrary code execution
Continuous	Number of words in installation scripts	Structural feature of source code
Continuous	Number of lines in installation scripts	Structural feature of source code
Continuous	Number of words in source code files	Structural feature of source code
Continuous	Number of lines in source code files	Structural feature of source code
Continuous	Number of URLs	Security-sensitive string(s)
Continuous	Number of IP addresses	Security-sensitive string(s)
Continuous	Number of suspicious tokens in strings	Security-sensitive string(s)
Continuous	Number of base64 strings	Presence of obfuscation
Continuous	Mean, std. deviation, 3rd quartile, and max value of Shannon entropy of strings in all source code files	Presence of obfuscation
Continuous	Number of homogeneous and heterogenous strings in all source code files	Presence of obfuscation
Continuous	Mean, std. deviation, 3rd quartile, and max value of Shannon entropy of identifiers in all source code files	Presence of obfuscation
Continuous	Number of homogeneous and heterogenous identifiers in all source code files	Presence of obfuscation
Continuous	Mean, std. deviation, 3rd quartile, and max value of Shannon entropy of strings in installation script	Presence of obfuscation
Continuous	Mean, std. deviation, 3rd quartile, and max value of Shannon entropy of identifiers in installation script	Presence of obfuscation
Continuous	Mean, std. deviation, 3rd quartile, and max value of ratio of square brackets per source code file size	String manipulation
Continuous	Mean, std. deviation, 3rd quartile, and max value of ratio of equal signs per source code file size	String manipulation
Continuous	Mean, std. deviation, 3rd quartile, and max value of ratio of plus signs per source code file size	String manipulation
Continuous	No. of files per selected extensions (91 in total)	Structural feature of the package

Table 3: Set of features considered for the classification of malicious packages.

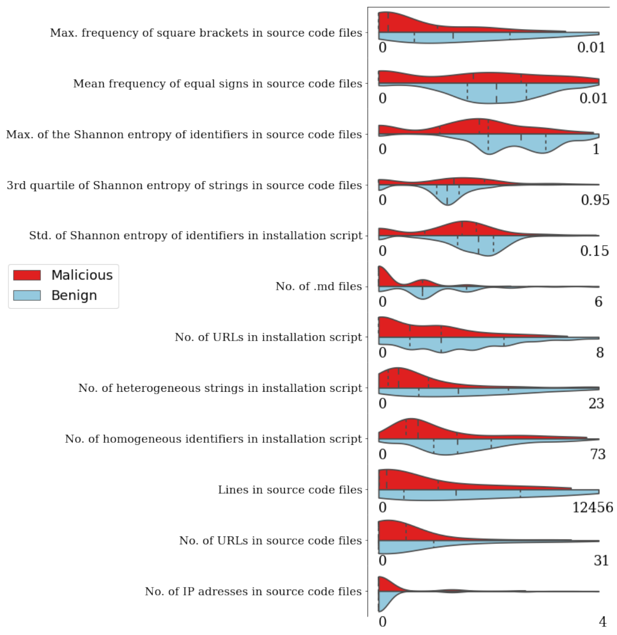


Figure 1: Distribution of a subset of continuous features extracted from benign and malicious samples (both Python and JavaScript).

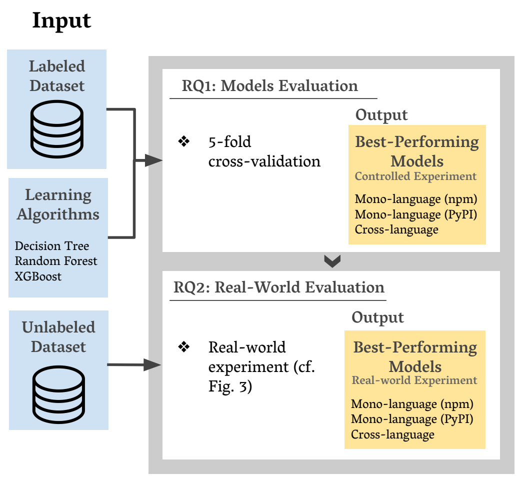


Figure 2: Workflow followed for the evaluation of our approach for the detection of potentially malicious packages.

4 EVALUATION

This section answers to the research questions. Figure 2 describes the approach followed. Similarly to [24, 30], we conduct both a controlled and a real-world experiment.

We answer RQ1 by conducting a controlled experiment, in which we use the labeled datasets constructed as described in Section 3.1 to evaluate the performances of the trained classifiers in both the cases of npm and PyPI. Once identified the best-performing models, we evaluated the mono-language and cross-language classifiers by analyzing newly published packages in npm and PyPI, thereby answering RQ2.

The labeled dataset used in the controlled experiment, the best-performing models from RQ1, and the list of malicious packages found in the wild are available online[6].

4.1 RQ1: Models

We evaluate the performances of the classification algorithms described in Section 3.2 to solve the task of detecting potentially malicious packages with a controlled experiment, where labels of data are known.

To address the limitation of small dataset sizes, we follow an approach similar to [30] and refrain from keeping a separate hold-out set. Instead, we conduct a 5-fold cross-validation experiment, repeating it ten times. This approach allows us to assess the performance of the models and report the corresponding score metrics. During each split of the cross-validation, we employ stratified sampling to ensure that the 90-10 ratio between benign and malicious samples is maintained.

Each learning algorithm (i.e., DT, RF, XGBoost) is trained respectively on the two mono-language datasets and on the cross-language dataset (cf. Section 3.1). We obtain a total of 9 classifiers. Table 4 reports the percentage values - computed on the positive class (i.e., malicious packages) - of precision, recall, F1-score, and accuracy after the 5-fold cross-validation for each ecosystem.

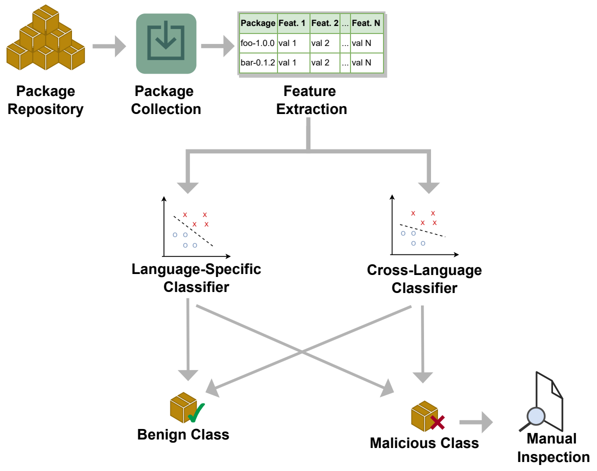
Detection in JavaScript. Considering the mono-language case (i.e., both train and test sets composed of only JavaScript samples), the model providing the highest precision is the one obtained with the DT algorithm (i.e., 100%). However, such a model is subject to a high number of false negatives (recall of 68.0%) when compared to XGBoost (recall of 75.5%). In fact, the latter model achieves the best trade-off between accuracy and recall (i.e., F1 score of 84.4%).

For the cross-language case (i.e., train set composed of JavaScript and Python samples, test set composed of JavaScript samples only), the model achieving the highest precision is RF. However, the best trade-off is offered also in this case by XGBoost.

Detection in Python. In both the case of mono-language models for Python (i.e., train and test sets composed of only Python samples) and cross-language models (i.e., train set composed of JavaScript and Python samples, test set composed of Python samples only), the model with highest precision is DT. However, the recall is too low to have practical utility. Thus, also in this case XGBoost outperforms the other models, having a precision of 74.4%, recall of 63.9%, and F1-score of 68.0%.

Response to RQ1: As shown in Table 4, the XGBoost models exhibit the most favorable balance between precision and recall in both the mono-language and cross-language scenarios for both JavaScript and Python packages.

[6]https://github.com/SAP-samples/cross-language-detection-artifacts



Package	Feat. 1	Feat. 2	...	Feat. N
foo-1.0.0	val 1	val 2	...	val N
bar-0.1.2	val 1	val 2	...	val N

Figure 3: Experiment consisting of the classification of daily-uploaded packages in PyPI and npm for the detection of malicious packages in the wild. The depicted procedure applies to both npm and PyPI.

By evaluating the performances of the XGBoost models both in the mono-language and cross-language case, it appears that the mono-language model performs slightly better than the cross-language one in the classification of JavaScript packages. Vice versa, in the case of Python, the cross-language model based on XGBoost performs slightly better than the mono-language one. Since a hold-out set was not used, the comparison between the cross-language and mono-language models, which exhibit the best performance, is addressed in the real-world experiment in Section 4.2.

4.2 RQ2: Real-World Evaluation

In this section, we evaluate how the mono-language models and the cross-language model identified in RQ1 (cf. Section 4.1) perform in the classification of benign and malicious packages in the wild. To do so we conduct the experiment depicted in Figure 3.

We collect newly uploaded packages to npm and PyPI within a period of 10 days (i.e., from Oct. 24 to Nov. 2, 2022). To get the list of packages uploaded to PyPI on a given day we rely on the XML-RPC APIs of the official warehouse[7]. Since a comparable feature is not available for npm, we use the RSS feed about new npm packages from *libraries.io*[8]. For both npm and PyPI, we directly download the packages from the official package repositories.

We extract the features from the downloaded packages as described in Section 3.3 and classify such packages using both the mono-language model (of the related ecosystem) and the cross-language model identified in Section 4.1. For the packages classified as malicious by at least one model, we conduct a manual review to verify the true positives and false positives. In the case of npm packages, we manually inspect *.js*, *.sh*, and *package.json* files while for PyPI packages we inspect *.py* and *.sh* files. In the manual analysis, we look for signs of malicious behavior (e.g., implementation of reverse shells, data exfiltration), and for the obfuscated scripts

[7]https://warehouse.pypa.io/api-reference/xml-rpc.html
[8]https://libraries.io

JavaScript								
	Mono-language (Train set: JS; Test Set: JS)				**Cross-language** (Train set: JS+PY; Test Set: JS)			
	Pr.	Rec.	F1	Acc.	Pr.	Rec.	F1	Acc.
DT	**100.0±0.0**	68.0±8.89	80.6±6.5	96.9±0.9	95.9±6.9	49.5±17.6	63.0±20.1	94.8±1.7
RF	98.5±3.1	53.5±14.7	68.1±12.8	95.3±1.4	**98.5±3.1**	50.0±16.8	64.55±16.1	95.0±1.6
XGB	96.5±4.0	**75.5±6.9**	**84.4±4.2**	**97.3±0.6**	97.1±3.8	**63.5±10.3**	**76.3±7.9**	**96.2±1.0**

Python								
	Mono-language (Train set: PY; Test Set: PY)				**Cross-language** (Train set: JS+PY; Test Set: PY)			
	Pr.	Rec.	F1	Acc.	Pr.	Rec.	F1	Acc.
DT	81.6±18.3	28.9±14.8	39.4±11.2	92.0±0.6	97.2±8.3	16.7±9.6	27.4±13.8	91.6±1.0
RF	79.2±9.4	51.7±14.4	61.0±9.9	93.8±1.0	92.5±16.9	15.6±8.6	25.9±12.9	91.6±0.9
XGB	**74.4±13.0**	**63.9±13.7**	**68.0±11.6**	**94.2±2.0**	**87.1±11.1**	**55.6±13.4**	**66.9±11.1**	**94.8±1.5**

Table 4: Results of 5-fold cross validation experiment for both the cases of JavaScript (JS) and Python (PY). For both mono-language and cross-language models, we report precision (Pr.), recall (Rec.), F1-score (F1), and accuracy (Acc.) in percentages.

	npm (JavaScript)				**PyPI (Python)**			
	Benign	Malicious (FP)	Malicious (TP)	Pr.	Benign	Malicious (FP)	Malicious (TP)	Pr.
Mono-language	10428	1229	38	3.1%	19097	485	15	3.1%
Cross-language	10519	1083	37	**3.4%**	19196	385	17	**4.4%**

Table 5: Comparison of mono-language and cross-language models in classifying daily-uploaded packages from the wild for 10 days. The best results are highlighted in bold.

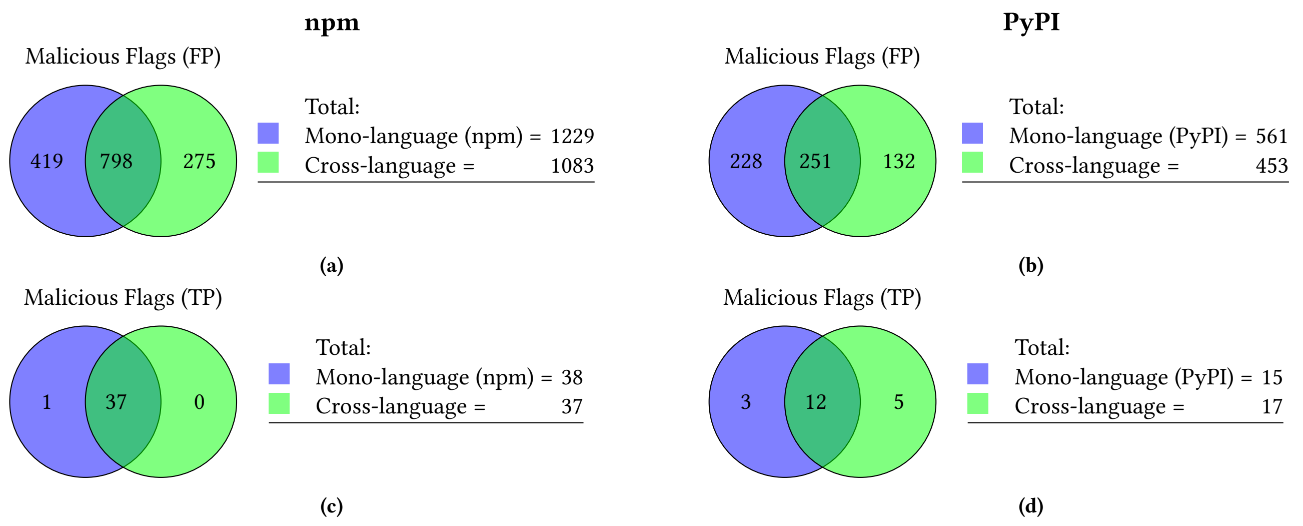


Figure 4: Comparison of results for the malicious samples classified by the mono-language and cross-language models in the case of npm. In (a) and (b) the false positives for npm and PyPI respectively. In (c) and (d) the true positives for npm and PyPI respectively.

we attempt to de-obfuscate them. Table 5 reports the results of the classification performed by both the mono-language and cross-language models.

In the case of npm, the mono-language model has 419 unique false positives w.r.t. the cross-language model (cf. Figure 4a). The number of unique false positives found by the cross-language model is 275. In terms of true positives, both models correctly classify the same 37 packages as malicious (cf. Figure 4c). However, the mono-language model identifies an additional malicious package. In terms of precision, the cross-language model shows slightly better performance compared to the mono-language model. Overall, we manually reviewed 1,530 packages over 10 days (i.e., 153 packages/day).

For PyPI, the mono-language model classifies as malicious 231 more packages than the cross-language model. Of these packages, 228 are false positives (cf. Figure 4b), and 3 are true positives that the cross-language model missed (cf. Figure 4d). Also in the case of PyPI, the cross-language model has better precision than the mono-language model (4.4% over 3.1%, cf. Table 5 column "Pr."). Overall, we manually reviewed 631 packages over 10 days (i.e., 63 packages/day).

Though the cross-language model does not drastically improve the precision compared to the mono-language models, the difference in terms of manual effort is non-negligible as the number of false positives considerably decreases without a major loss in terms of true positives.

All the malicious packages (true positives) found in npm (38 in total) and PyPI (20 in total) have been reported to the respective security teams following the official channels (cf. Section 6). As a result of our reporting, after triaging by the respective npm and PyPI security teams, the 58 reported malicious packages have been removed from the repositories and are no longer available for download. As we believe in the importance of maintaining (for research purposes) a database of malicious packages related to OSS supply chain attacks, we contributed to BKC by uploading the detected malicious packages.

Response to RQ2: By classifying packages daily uploaded to npm and PyPI within a period of 10 days we found that the cross-language model achieves better precision than the mono-language models. In total, we find 58 previously unknown malicious packages (38 for npm and 20 for PyPI).

5 DISCUSSION

In this section, we describe the characteristics of the malicious packages found in npm and PyPI. We report some remarks on the two ecosystems observed while inspecting the false positives. Finally, we highlight notable differences in terms of features between malicious true positives, false positives, and packages flagged as benign.

Malware types. Figure 5 depicts the types of malicious behaviors found in both npm and PyPI during our real-world experiment (cf. Section 4.2).

The majority of malicious packages both in npm and PyPI aim at achieving data exfiltration (27 and 6 packages respectively) and the most common exfiltrated information consists of public IP address and environment variables. Other common malicious behaviors that we observed involve the opening of a reverse shell and the download and execution of malicious code (i.e., dropper). The latter is commonly achieved by making an HTTP(S) request to download and execute a second-stage payload. In a malicious discovery within npm, we identified a novel tactic (in comparison to those in the BKC). This tactic involves placing the second-stage payload within the TXT field of a DNS response, under the control of the attacker. Subsequently, a request to this DNS is initiated through a third-party service (i.e., Google's DNS resolver). In this way, the attacker is less likely to have the request from the victim blocked by any firewall.

In npm, we found also research proofs-of-concept, i.e., the attacker aims at achieving execution to demonstrate the potential risks when installing their package. One notable example is a package containing code obfuscated with AES-256 encryption, which is decrypted at runtime. Upon execution, the code creates a new file on the victim's machine containing a warning message about the potential harm. Furthermore, we identified 4 packages (3 in npm and 1 in PyPI) conducting a Rickrolling attack at installation time (i.e., starting the reproduction of the song *Never Gonna Give You Up* from Rick Astley). When we reported these, the respective security teams decided not to remove the said packages as they did not consider Rickrolling as violating their term of service.

Although this behavior was not part of the ones available in the training dataset (cf. Section 3.1), our approach identified two keyloggers in PyPI.

Malware campaigns. During the analysis of daily uploaded packages to npm and PyPI, we observed the presence of several malware campaigns. Additionally, we consider variations of the malicious code that only differ in the subdomain of the URLs.

In the case of npm, we detected 7 campaigns of which the largest includes 12 packages. Only in one campaign the attacker experiment with different ways to achieve code execution, while in all the other cases both the malicious code and the way of triggering its execution are the same. In the case of PyPI, we detected 5 campaigns of which the largest includes 4 packages.

It is worth mentioning that we have detected a case of a cross-language campaign, meaning that the same malicious behavior was present in both npm and PyPI. This finding suggests that attackers may spread their malicious activities across multiple ecosystems in order to increase their chances of success.

Malicious code obfuscation. The majority of malicious packages found both in npm and PyPI during our experiment do not obfuscate the malicious code. Among the 38 malicious packages from npm, only 2 obfuscate the code using AES encryption, and 2 packages use custom obfuscation (e.g., renaming sensitive functions like exec in human-unreadable ways). Among the 20 malicious packages found in PyPI, 3 use simple obfuscation techniques (e.g., encoding the source code in bytes, base64, rot13) and 3 other packages employ custom obfuscation.

Analysis of false positives. The majority of false positives in both npm and PyPI consist of small packages (often containing only *package.json* or *setup.py*) with almost no functionality (e.g. print of "hello world" or sum of numbers). Possible use cases of such packages are to reserve names (e.g., future projects, prevent typosquatting) in the package repository or to test the upload of packages.

We also observe the concept of campaign (i.e., packages with the same behavior) for some non-malicious packages. A curious case observed in npm is about a set of packages whose sole purpose is to use the *give-me-a-joke* project as a dependency, probably with the aim of increasing its popularity. Another interesting finding is about a package containing nothing but the CV of its creator.

In 4 packages (3 in npm and 1 in PyPI) we detect the presence of obfuscated code. Since we could not observe obvious malicious behaviors we flagged these as false positives.

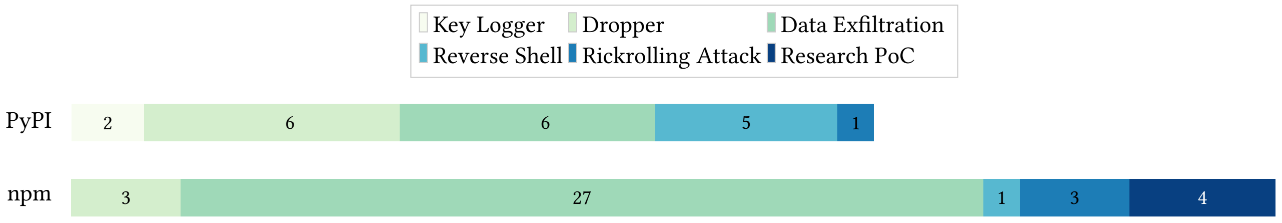


Figure 5: Types of malicious behavior among the 58 findings between npm and PyPI.

Finally, we acknowledge that the number of false positives (cf. Table 5) clearly needs further improvement, although they are still manageable for a manual review, especially considering that the majority of them were small in size.

Comparison on features. While 81% of the malicious packages found in npm (true positives) make use of installation hooks, only 8% and 2% make use of them among the false positives and the packages flagged as benign, respectively. Also in the case of PyPI, the majority of true positives (i.e., 58%) make use of installation hooks, while it is less common in the case of false positives and packages flagged as benign (i.e., 5% and 1%, respectively). Another aspect concerns the presence of Markdown files: only a minority of true positives (i.e., 25% for npm and 42% for PyPI) contain at least one *.md* file, whereas most of the packages flagged as benign contain such files (82% for npm and 79% for PyPI).

Cross-language vs. Mono-language models. As observed in the real-world evaluation (cf. Section 4.2), the cross-language model exhibits greater precision compared to the mono-language models. To delve into the rationale behind this, we explored the relationship between feature importance across models and the traits of the malicious packages identified by each model.

The usage of installation hooks and the count of Markdown files rank within the most influential features for the mono-language model of npm, whereas they hold less significance for the mono-language model of PyPI. When merging npm and PyPI samples to train the cross-language model, the importance of these features reemerges. Considering that a substantial portion of true positives in PyPI use installation hooks or lack Markdown files, this characteristic likely contributes to the improvement of the cross-language model's performance compared to the mono-language model specifically designed for PyPI.

Features associated with the size of source code files (e.g., *number of lines*) and the count of homogeneous strings in source code files (cf. Section 3.3) rank within the most influential features for the mono-language model of PyPI, whereas they hold less significance for the mono-language model of npm. Merging npm and PyPI samples to train the cross-language model, the importance of these features reemerges. Given that several false positives in npm lack script files and consist solely of the *package.json*, this characteristic is likely a contributing factor in augmenting the cross-language model's precision compared to the mono-language model for npm.

Porting to other languages. In our work, we explored the feasibility of detecting malicious packages across multiple ecosystems. As a preliminary analysis, we focused on the case of npm and PyPI. To achieve this, we defined a set of features that are not specific to npm or PyPI, making it feasible to port our approach to other ecosystems. The only feature required during porting is that the target ecosystem supports the use of install hooks, which is also one of the most important features for the trained models. Examples of ecosystems that support this functionality include Composer for PHP [7] and Gem for Ruby (by extending the definition of 'usage of installation hook(s)' to include the usage of extensions by gems [3]).

6 RESPONSIBLE DISCLOSURE

npm and PyPI have two different ways of officially reporting malicious packages.

In npm, reporting is done via the UI of the official website: the user has to search for the package to be reported, navigate to the project page, click on the 'Report Malware' button, and finally fill in the report form. This procedure can make it difficult to report multiple malwares at the same time (especially since malware campaigns are common). Moreover, multiple reports at the same time enable CAPTCHA tests. All 38 malicious packages found during our analysis have been responsibly reported to the security team according to the abovementioned procedure. Except for the packages conducting a Rickrolling attack (whose behavior was not considered a violation of npm's terms of use), they have been removed from the package repository.

In PyPI, the official procedure to report malicious packages consists of sending an e-mail to the security team[9] with the names of the packages and (preferably) the link to the lines of code containing the malware highlighted using *Inspector*[10]. Also in this case, we responsibly reported the 20 malicious packages found during our analysis. Following the triaging procedure, the PyPI security team confirmed our findings and removed them from the package repository.

7 THREATS TO VALIDITY

Though the results obtained in evaluating our cross-language approach are promising, there are some threats to validity worth highlighting.

Both the mono-language and cross-language models have been trained on previously known malicious samples from BKC [2]. This has introduced a bias on the type of malware identifiable through our classification (cf. Section 3.1), and a limitation of the approach

[9]https://pypi.org/security/
[10]https://inspector.pypi.io

is the difficulty of finding new (unseen) types of malware. Furthermore, attackers could potentially evade detection by carefully constructing malicious packages so that the values of their features (cf. Section 3.3) mimic the distribution seen in benign packages.

In the real-world experiment, we do not manually review (due to their large number) the packages flagged as benign by our classifiers to check the presence of false negatives so we cannot draw conclusions in terms of recall. Nevertheless, during our analysis, we learned from the news [6] about the discovery of 12 malicious packages uploaded to PyPI in the same period of our experiment. By inspecting these cases we observed that none of them were flagged as malicious by our classifiers (both mono-language and cross-language). The peculiarity of these missed packages - compared to those known from BKC - is that they are clones of benign packages (thus containing largely benign code) into which a line of malicious code has been injected.

As described in Section 5, in the manual analysis of packages collected in the wild and flagged as malicious, we encountered some suspicious packages heavily obfuscated, that we marked as false positives as we were not able to identify malicious behavior. However, a deeper analysis could reveal malicious intent and thus we could have misattributed some false positives.

As mentioned in Section 3.1, since there is no ground truth for benign packages in npm and PyPI, we made some assumptions and did our best effort to check that the benign samples in our dataset do not include malicious code. However, we cannot exclude that some packages considered benign may be hiding malicious code and thus the corresponding label in our dataset would be wrong.

Related to the imbalance problem, there is no certainty about the actual ratio of benevolent to malevolent packages in package repositories. In our work, we assume a ratio of 90% benign samples and 10% malicious ones as suggested in [23, 31], but other options have been proposed, e.g., 1% of malicious packages in [24]. As we did not conduct experiments on changing such ratios, we cannot evaluate the impact of different ratios on our classification.

In our work, we evaluate mono-language and cross-language models. However, both are trained using language-independent features, thus the results may differ with mono-language models trained on language-specific features. We consider our approach complementary to those using language-specific features: we cope with the scarcity of samples to increase the set of known malicious samples, required for language-specific approaches.

8 RELATED WORKS

Our work proposes a supervised learning approach for the detection of potentially malicious packages in software repositories (i.e., npm and PyPI) to counter OSS supply chain attacks. A comparison to the closest works is shown in Table 6. We further discuss works that focus on the problem of malicious code in the context of OSS supply chain security.

Sejfia et al. [30] propose a supervised learning approach combined with a code reproducer and a simple clone detector for the automated detection of malicious packages in npm. Although they also consider some language-generic features (e.g., presence of minified code, binary files), their main focus is on language-dependent aspects (e.g., use of specific APIs) for JavaScript.

Ohm et al. [24] conducted an extensive evaluation of 25,210 models to assess the feasibility of utilizing supervised learning techniques for the detection of malicious packages. They specifically focus on the npm ecosystem and consider mainly language-dependent features next to more generic ones.

Compared to [30] and [24], we only consider language-independent features that can be easily applied both to JavaScript and Python and we consider DT and RF as they are the best-performing models in [30] and [24], respectively. Since the models and datasets are not available, we cannot compare our models with theirs.

	Our work	**Sejfia et al. [30]**	**Ohm et al. [24]**
Target Language:	Python, JavaScript	JavaScript	JavaScript
Analysis type:	Static	Static	Static
Features:			
Use of installation hooks	✓	✓	✓
No. of words/lines in installation scripts	✓		
No. of words/lines in source code	✓		
Presence of URLs	✓		✓
Presence of IP addresses	✓		✓
Presence of Base64 strings	✓		✓
Presence of sensitive strings	✓		✓
Presence of obfuscation	✓	✓	
Presence of executables	✓	✓	
Count of script files	✓		✓
Count of files per extension	✓		
Stats on square brackets	✓		✓
Stats on equal sign	✓		
Stats on plus sign	✓		
Use of sensitive APIs		✓	✓

Table 6: Comparison with related works [24, 30] in the context of ML approaches applied to the detection of malicious packages in package repositories.

In another work, Ohm et al. [25] describe an approach to detect malicious package campaigns by unsupervised signature generation relative to code reuse. In particular, they generate the Abstract Syntax Tree (AST) from npm packages and cluster them so that they can identify packages sharing commonalities. In our work, we only focus on lexical features, and our approach is based on supervised learning. In addition, since our focus is cross-language, our target language is not limited to JavaScript.

Still, Ohm et al. [27] describe a dynamic analysis approach for the detection of malicious JavaScript and Python packages through the analysis of forensic artifacts. Compared to their work, our approach is only static and more lightweight.

Garret et al. [15] apply an anomaly detection approach to identify malicious updates in the npm ecosystem. Their use case is to compare a version of a JavaScript package with previous ones to detect malicious updates. They only consider JavaScript-specific features related to whether or not code is added. Instead, our purpose is to determine whether a package is malicious as is, regardless of its relationship to previous versions.

Duan et al. [10] propose a pipeline based on static and dynamic analysis for the detection of malicious packages in interpreted languages (i.e., JavaScript, Python, Ruby). In particular, they derive heuristic rules from a previous study on supply chain attacks. Though they also target malware detection on multiple languages, compared to them we propose a lightweight approach based on machine learning, that does not require updating and maintaining a list of APIs for each supported language, and that does not require code execution.

In the context of Java, Ladisa et al. [21] propose several indicators of malicious behavior that can be observed from Java bytecode. Apart from the different scope, we take the idea of applying Shannon entropy to strings to detect the presence of obfuscation and extend it to identifiers. In addition, we construct the list of suspicious tokens taking cues from their approach.

Fass et al. [12] aim at the detection of obfuscated code in JavaScript through the extraction of features from the AST and the training of a RF classifier. Compared to them, our approach to detect obfuscation (e.g., encoded strings) focuses only on lexical features.

Pfretzschner et al. [28] describe JavaScript language features that can be exploited by malicious dependencies to conduct attacks. Their approach to detection is static and only focuses on JavaScript.

Vu et al. [35] approach the detection of malicious injections in Python packages by analyzing discrepancies between source code and built version. Scalco et al. [29] transfer this approach in the context of JavaScript. Since obtaining the source code corresponding to a package is not straightforward [34], our goal is to determine whether a package is malicious without relying on additional, possibly unavailable, inputs.

Still, Vu et al. [36] interview PyPI's administrators to explore the security goals of such package repository and create a benchmark dataset to review the current malware checker tools available for PyPI (e.g., Bandit4Mal[11], OSSGadget[12]). They report that package repositories need malware detectors with very low false-positive rates and thus the need for improvements in detection capabilities.

Although our focus is on pre-built packages, it is worth mentioning that different works for the detection of OSS software supply chain attacks have been proposed also in the context of source code repositories [8, 16, 37].

9 CONCLUSION AND FUTURE WORKS

In this work, we investigate the feasibility of a cross-language approach for the detection of malicious packages in npm and PyPI. Our approach is lightweight and focuses on lexical aspects of code (e.g., characteristics of strings) and structural features of packages (e.g., number of files per extension). First, we define a set of simple yet effective features to discriminate between malicious and benign packages in JavaScript and Python. Then, we analyze the performances of learning algorithms based on DT for the classification of malicious and benign packages through a controlled experiment (i.e., where labels are known) and found that XGBoost performs better in all the contexts (i.e., using mono-language datasets for JavaScript or Python, and a cross-language dataset). Finally, we evaluated the mono-language and cross-language models by classifying packages in the wild. We identified 58 previously unknown malicious packages and reported them to the respective package repositories.

Considering the encouraging outcomes of our assessment of cross-language detection feasibility, our future focus aims to enhance detection accuracy. This involves delving into supplementary machine learning algorithms to find the best-performing model and expanding feature sets to further refine our approach. Furthermore, we envisage combining the ML-based classifier with the automatic analysis of packages associated with the same author of confirmed malicious packages. In fact, with a first manual inspection of the packages uploaded by the same author as those detected in our experiment, we were already able to find additional malicious packages (uploaded in a time window prior to our experiment), either related to earlier versions of the same package or other packages from the same campaign. In addition, we intend to extend our evaluation to include other languages (e.g., Ruby, PHP).

10 DATA AVAILABILITY

The labeled dataset used for the training (cf. Section 3.1), the best-performing models obtained in RQ1, and the list of malicious packages found in the wild are available online[13].

Acknowledgements. We thank the reviewers for their constructive feedback, which has greatly contributed to the improvement of our work. This work is partly funded by EU grants No. 952647 (AssureMOSS) and No. 101120393 (Sec4AI4Sec).

REFERENCES

[1] [n. d.]. 1. An Introduction to Distutils. https://docs.python.org/3/distutils/introduction.html. [Accessed 02-Dec-2022].
[2] [n. d.]. Backstabber's Knife Collection. https://dasfreak.github.io/Backstabbers-Knife-Collection/. [Accessed 07-Sep-2022].
[3] [n. d.]. Gems with Extensions. https://guides.rubygems.org/gems-with-extensions. [Accessed 30-Jun-2023].
[4] [n. d.]. Octoverse 2022: The state of open source. https://octoverse.github.com/. [Accessed 22-Nov-2022].
[5] [n. d.]. package.json - npm Docs. https://docs.npmjs.com/cli/v8/configuring-npm/package-json#scripts. [Accessed 02-Dec-2022].
[6] [n. d.]. Phylum Discovers Dozens More PyPI Packages Attempting to Deliver W4SP Stealer in Ongoing Supply-Chain Attack. https://blog.phylum.io/phylum-discovers-dozens-more-pypi-packages-attempting-to-deliver-w4sp-stealer-in-ongoing-supply-chain-attack. [Accessed 11-Jan-2023].
[7] [n. d.]. Scripts - Composer -- getcomposer.org. https://getcomposer.org/doc/articles/scripts.md#scripts. [Accessed 30-Jun-2023].
[8] Alan Cao and Brendan Dolan-Gavitt. 2022. What the Fork? Finding and Analyzing Malware in GitHub Forks. In *Proc. of NDSS*, Vol. 22.
[9] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.
[10] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *28th Annual Network and Distributed System Security Symposium, NDSS*. https://www.ndss-symposium.org/wp-content/uploads/ndss2021_1B-1_23055_paper.pdf

[11] https://github.com/lyvd/bandit4mal
[12] https://github.com/microsoft/OSSGadget
[13] https://github.com/SAP-samples/cross-language-detection-artifacts

[11] ENISA. [n. d.]. ENISA Threat Landscape 2022. https://www.enisa.europa.eu/publications/enisa-threat-landscape-2022. [Accessed 22-Nov-2022].

[12] Aurore Fass, Robert P Krawczyk, Michael Backes, and Ben Stock. 2018. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 303–325.

[13] Eric Filiol, Marko Helenius, and Stefano Zanero. 2006. Open problems in computer virology. *Journal in Computer Virology* 1 (2006), 55–66.

[14] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[15] 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.

[16] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schaefer. 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)*. 258–267. https://doi.org/10.1109/ICSE-SEIP52600.2021.00035

[17] Jossef Harush. [n. d.]. How 140k NuGet, NPM, and PyPi Packages Were Used to Spread Phishing Links. https://checkmarx.com/blog/how-140k-nuget-npm-and-pypi-packages-were-used-to-spread-phishing-links/. [Accessed 17-08-2023].

[18] The White House. [n. d.]. Executive Order on Improving the Nation's Cybersecurity. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/. [Accessed 22-Nov-2022].

[19] Zhipeng Huang and Yeye He. 2018. Auto-Detect: Data-Driven Error Detection in Tables. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1377–1392. https://doi.org/10.1145/3183713.3196889

[20] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. forthcoming 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. *IEEE Symposium on Security and Privacy (SP)* (forthcoming 2023).

[21] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, and Serena Elisa Ponta. 2022. Towards the Detection of Malicious Java Packages. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses* (Los Angeles, CA, USA) *(SCORED'22)*. Association for Computing Machinery, New York, NY, USA, 63–72. https://doi.org/10.1145/3560835.3564548

[22] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, and Serena Elisa Ponta. 2022. Towards the Detection of Malicious Java Packages. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses* (Los Angeles, CA, USA) *(SCORED'22)*. Association for Computing Machinery, New York, NY, USA, 63–72. https://doi.org/10.1145/3560835.3564548

[23] Robert Moskovitch, Dima Stopel, Clint Feher, Nir Nissim, and Yuval Elovici. 2008. Unknown malcode detection via text categorization and the imbalance problem. In *2008 IEEE international conference on intelligence and security informatics*. IEEE, 156–161.

[24] 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 *Proceedings of the 17th International Conference on Availability, Reliability and Security*. 1–10.

[25] Marc Ohm, Lukas Kempf, Felix Boes, and Michael Meier. 2020. If You've Seen One, You've Seen Them All: Leveraging AST Clustering Using MCL to Mimic Expertise to Detect Software Supply Chain Attacks. *CoRR* abs/2011.02235 (2020). arXiv:2011.02235 https://arxiv.org/abs/2011.02235

[26] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. arXiv:2005.09535 [cs.CR]

[27] 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* (Virtual Event, Ireland) *(ARES '20)*. Association for Computing Machinery, New York, NY, USA, Article 65, 6 pages. https://doi.org/10.1145/3407023.3409183

[28] 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* (Reggio Calabria, Italy) *(ARES '17)*. Association for Computing Machinery, New York, NY, USA, Article 68, 6 pages. https://doi.org/10.1145/3098954.3120928

[29] Simone Scalco, Duc-Ly Vu, Ranindya Paramitha, and Fabio Massacci. 2022. On the feasibility of detecting injections in malicious npm packages. https://doi.org/10.1145/3538969.3543815

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

[31] Asaf Shabtai, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. 2009. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *information security technical report* 14, 1 (2009), 16–29.

[32] Sonatype. [n. d.]. 8th Annual State of the Software Supply Chain Report. https://www.sonatype.com/state-of-the-software-supply-chain/introduction. [Accessed 22-Nov-2022].

[33] Ryan Turner, David Eriksson, Michael McCourt, Juha Kiili, Eero Laaksonen, Zhen Xu, and Isabelle Guyon. 2021. Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. In *NeurIPS 2020 Competition and Demonstration Track*. PMLR, 3–26.

[34] Duc-Ly Vu. 2021. py2src: Towards the Automatic (and Reliable) Identification of Sources for PyPI Package. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1394–1396. https://doi.org/10.1109/ASE51524.2021.9678526

[35] 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 780–792. https://doi.org/10.1145/3468264.3468592

[36] Duc-Ly Vu, Zachary Newman, and John Speed Meyers. 2022. A Benchmark Comparison of Python Malware Detection Approaches. *arXiv preprint arXiv:2209.07957* (2022).

[37] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Towards using source code repositories to identify software supply chain attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2093–2095.