Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Bachelor Thesis

in Computer Science

Typosquatting Attacks and Mitigations

by

Minh Tien Truong

1st Supervisor:	Prof. Dr. Luigi Lo Iacono
2nd Supervisor:	Prof. Dr. Sascha Alda
Advisor:	Dr. Marcus Rickert
	Accso - Accelerated Solutions GmbH
Submitted on:	6th March 2023

Acknowledgment

I would like to express my heartfelt gratitude to my friends and family for their unwavering support throughout the completion of this bachelor thesis. Their encouragement and motivation helped me to stay focused and motivated throughout the writing process. In particular, I would like to thank those who took the time to proofread my work and provide valuable feedback, including David, Jonathan, Marcus and Philip. Your insightful comments and constructive criticism helped me to improve the quality of my thesis significantly. I am truly blessed to have you all in my life, and I will always be grateful for your kindness and generosity. Additionally, I would like to thank Dr. Marc-Philipp Ohm for granting me access to his dataset.

Erklärung

Minh Tien Truong
Adresse: *Zensiert*

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbst angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

.....	.....
Ort, Datum	Unterschrift

Abstract

Typosquatting has been among the most common attacks on software supply chain systems in recent years. These attacks involve malicious packages that mimic legitimate packages. The attackers register these malicious packages under a name resembling the legitimate ones while containing a typo or other types of misspelling. As a result, developers making one such mistake during the installation process download the malicious library instead of the legitimate one.

This work compares and contrasts different package managers regarding their vulnerability and the derivation of possible approaches to reduce and prevent future attacks. For this purpose, the package managers npm, Maven, PyPI, Go, Packagist, NuGet, RubyGems, and Crates were compared for specific characteristics, and their attack frequency was brought into relation. Subsequently, the effect of the different package manager properties on the attack frequency has been investigated. Another core investigation of this work is the effectiveness in the detection of positive typosquatting candidates using various string-matching algorithms. These were evaluated with respect to the identification of typosquatting candidates and then compared with each other in terms of accuracy and runtime. Additionally, a new string-matching algorithm, a modified variant of the Damerau-Levenshtein distance, has been proposed and examined. This algorithm exhibited a significantly greater true positive rate than the other string-matching approaches. In order to verify whether the typosquatting candidates are genuine typosquatting packages, machine learning techniques were incorporated to classify packages either into malicious or benign classes. The result is a 98% accuracy on the datasets collected for this thesis and an 88% accuracy on the evalutation of an external dataset.

Keywords— package manager, typosquatting, npm, security, approximate pattern matching, levenshtein, damerau-levenshtein, jaro-winkler, gestalt-pattern, machine learning, malicious code, decision tree, random forest

Contents

List of Figures

List of Tables

List of Abbreviations

AST	Abstract Syntax Tree
CART	Classification and Regression Trees
CLI	Command Line Interface
OSS	Open Source Software
SSC	Software Supply Chain
VCS	Version Control System

1. Introduction

The importance of security in IT has become particularly clear in recent years. One of the most prominent offenders is the ransomware *WannaCry*, which was able to infiltrate over 300,000 computers in over 150 countries in May 2017 [1], [2]. Shortly after, in July 2017, the Equifax group experienced a data breach due to an unpatched vulnerability. This breach resulted in the data of around 50% of US citizens being stolen. The stolen data ranged from general personal data to credit card information and social security numbers [3]. Remarkably during the height of the COVID-19 pandemic, the number of reported crimes to the Internet Crime Complaint Center (IC3) increased by approximately 400% [4]. In December 2021, the vulnerability *Log4Shell* was discovered. The vulnerability allowed malicious actors to execute arbitrary code on the host system. Wiz and EY (Ernest & Young) found that 93% of the analyzed cloud environments were vulnerable to Log4Shell [5].

In particular, attacks on software supply chains (SSC) rapidly increased during this period [6]. SSC refers to components and processes involved in creating, modifying, and distributing software products [7]. According to Sonatype, an average annual increase of 742% was recorded in the last three years [8]. A highlighting of this statistic can be found in Figure 1. While the number of malicious packages detected by Sonatype was approximately 12,000 in 2021, this quickly rose to 88,000 in 2022. The most commonly used attack types on SSC were *Dependency Confusion* and *Typosquatting Packages* [6]. The difference between this type of attack and the Equifax data breach, as well as the Log4Shell incident is that perpetrators are now actively trying to introduce vulnerabilities into a system instead of passively waiting for one to present itself.

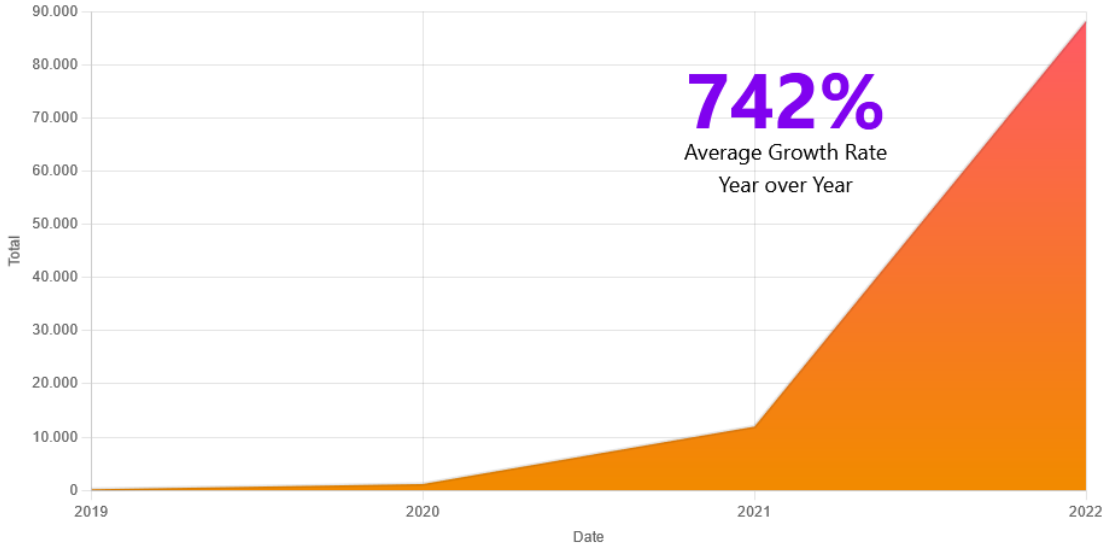

Figure 1: Detection of Software Supply Chain Attacks from 2019 to 2022 [8]

Open source experienced a rapid rise in popularity. For instance, GitHub is home to over 360 million repositories created by over 100 million users [9]. A study conducted by Synopsis also estimates that 97% of the code bases contain open source software (OSS) [10]. While OSS offers many advantages, such as speed and cost efficiency during development, it also offers new attack vectors to be exploited. In order to distribute and manage these OSS, package managers are used. Package managers are critical SSC components in the software development process, helping developers and users find and manage the packages they need to build and run their applications. Some popular package managers include npm for JavaScript, PyPI/pip for Python, and Maven for Java. However, due to their popularity and widespread use, package managers are also attractive targets for attacks. Malicious actors may exploit the users' trust in these systems for their own personal gain.

As mentioned, these attacks often take the form of dependency confusion or typosquatting packages.

The main focus of this thesis will therefore be typosquatting attacks. In the context of package managers, typosquatting attacks involve creating malicious packages designed to be confused with legitimate packages by using similar or misspelled names. These attacks trick users into downloading and installing malicious packages, for instance, `npm install react` vs. `npm install raect`. The targets of these attacks are often developers, unlike usual cyber attacks, which most frequently target the end-user.

1.1. Contribution and Goals

This thesis aims to derive and develop mitigations for typosquatting attacks on application package managers by comparing different package managers, evaluating the number of attacks on those package managers, and investigating the effect of various package manager properties on their appeal as a primary target for typosquatting attacks. Aside from analyzing the package managers and detecting mitigating strategies through package manager properties, alternative methodologies for detecting and identifying such typosquatting packages will be implemented and examined based on current literature. A quantitative analysis of real-world typosquatting packages targeting the npm ecosystem was conducted for this purpose in order to identify common patterns and properties that will aid in the identification of such campaigns. The data and typosquatting packages used, analyzed, and collected for this thesis are stored in a private GitHub repository[1]. Access will be granted upon request.

The objective of this work thus consists of finding an answer to the following questions:

- Are there differences in the frequency of attacks depending on package managers?
- If so, why are there differences in the frequency of attacks depending on package managers?
- Can general practices and procedures be derived from less vulnerable package managers to mitigate typosquatting attacks?
- Can typosquatting attacks or packages be detected or mitigated at the package manager level?

1.2. Outline

This thesis is structured into six chapters, with the chapter Introduction being the first one, introducing the reader into the topic of typosquatting attacks, the contributions of this thesis and the outline. The second chapter, Fundamentals introduces some of this thesis's necessary definitions and fundamentals. In particular, definitions for components or terms from the area of the SSC are covered here, as well as some principles in machine learning and approximate string matching. The third chapter, Typosquatting Attacks follows, in which the procedures of typosquatting attacks are explained and the term *typosquatting* itself is further differentiated. Subsequently, historical examples are presented. The fourth chapter, Current State of the Art and Related Literatures, deals with the research literature in the area of typosquatting attacks on package managers and literatures in the area of approximate string matching. Chapter five, Methodology deals with data acquisition and cleansing required for the chapter Mitigation and Detection of Typosquatting Packages.

The central part of this thesis is chapter six, Mitigation and Detection of Typosquatting Packages, covering, in general, the methodology, analysis, and implementation of the mitigation and detection strategies of typosquatting packages. This chapter is divided into three subchapters. The first subchapter, Comparison of Package Managers, deals with the comparison of different package managers and examines the correlation of the attack frequency with package manager specific properties. Based on these, possible mitigation

[1] https://github.com/xMinhx/Typosquatting-Attacks-and-Mitigations

measures are derived. The subchapter Effectivity of String Metrics for Typosquatting Packages statistically examines the effectivity of typosquatting packages based on their package names and identifies characteristics of particularly effective attacks. In addition, this subchapter compares various string-matching algorithms with one another for their accuracy and runtime performance in identifying typosquatting candidates. Subchapter Detection of Malicious Packages presents a method for detecting malicious intent among typosquatting candidates and thus distinguishes typosquatting candidates from typosquatting packages. For this purpose, the presented method uses procedures from the field of machine learning. The last chapter, Conclusion and Future Work summarizes the knowledge acquired from this thesis and suggests approaches for further research.

2. Fundamentals

For the understanding of this work, terms from the area of software development are required. These are explained in the following sections in more detail.

2.1. Package / Library

Packages often referred to as libraries, are reusable specific types of software components that are pre-written mainly by other programmers but outwardly offer an interface [11]. Developers can then use this interface to integrate the functionality of a specific software component into their own software component.

2.2. Dependency

A package is a dependency if it is required by another software component to function properly [11]. A distinction is made between two types of dependencies, direct and transitive dependencies [12]. A direct dependency between a software component and a package exists if the latter is integrated directly and explicitly by the developer of the primary software component. A transitive dependency between a software component and a library or package exists if the latter is indirectly included in the software component through other packages. The direct dependencies of a package integrated into a software component are thus transitive dependencies of the software component. The distinction between direct and transitive dependencies is further illustrated in the following figure:

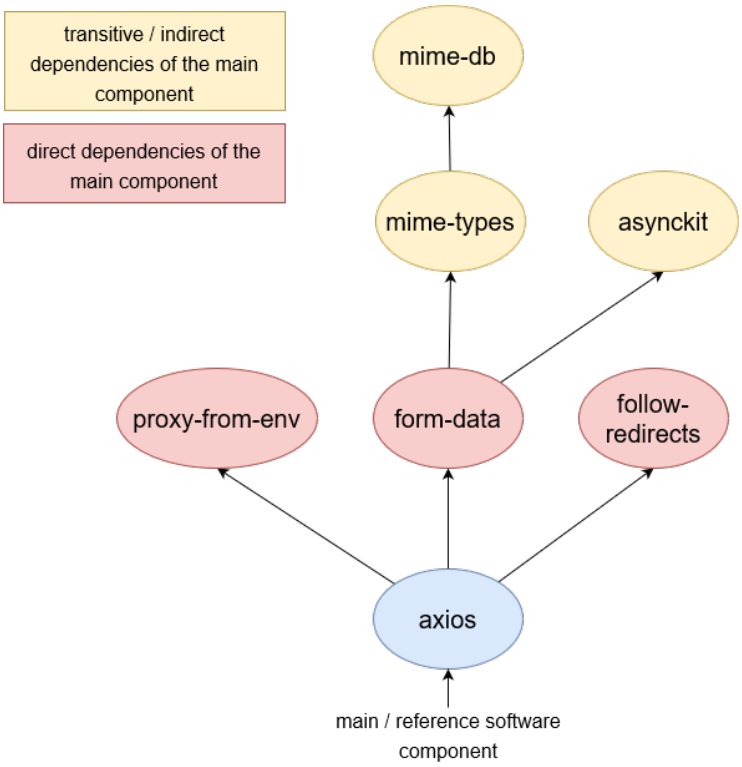

Figure 2: Simplified dependency tree of the axios package

2.3. Package Repository

Package repositories are components of SSC. A package repository is a collection of packages on a remote server from which the respective packages can be downloaded. Often such repositories offer a search function as well as handle the management of metadata. Examples of metadata include the number of downloads, the author of the package, versioning, release date, or dependencies. Usually, such package repositories are tailored to a specific programming language or operating system [13].

2.4. Package Manager

Package managers are components of SSC as well. These software tools have the purpose of assisting developers in downloading and managing their packages [13]. That is especially necessary to manage all the dependencies automatically so developers do not have to download and organize them manually. For example, to download a specific package with all its dependencies with the package manager npm, one can use the command `npm install <package>`. Package manager and package repository are often used interchangeably due to their close relationship and interaction. In the following, the term package manager in this thesis refers to both components, the package manager and the package repository. If only the package repository component is meant, this term is also used as such. If only the component package manager is meant, then this is additionally emphasized.

2.5. Approximate String Matching

Approximate string matching is a method that aims to match identical or similar strings [14]. The similarity of two or more strings can be defined by certain algorithms or metrics. Depending on the algorithm or metric, the similarity of two strings can be specified either in operation steps (i.e., how many operations are required to convert one string into the other) or by assigning a value between 0 and 1, where 0 often means no similarities and 1 indicates that both strings are identical [15], [16]. Four of the five approximate string-matching algorithms considered in this thesis are introduced in the following subsections. Even though, strictly speaking, not all of the string-matching methods presented here are metrics, the term metric is still used for all of them for simplicity.

2.5.1. Levenshtein

Vladimir Levenshtein first introduced the Levenshtein distance in 1966 [16]. The Levenshtein distance specifies the similarity of two strings as the number of operations necessary to transform one string into the other [14]. Allowed operations are insertions, deletions, and substitutions. For instance, given a string "care" and "car" the Levenshtein distance would be one. By removing or inserting the letter "e", one can transform "care" into "car" in one operation step and vice versa. The algorithm for the Levenshtein distance can be described by the following recursion [17]:

$$d_{a,b}(i,j) = \min \begin{cases} 0 & \text{if } i = j = 0, \\ d_{a,b}(i-1,j) + 1 & \text{if } i > 0, \\ d_{a,b}(i,j-1) + 1 & \text{if } j > 0, \\ d_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} & \text{if } i,j > 0, \end{cases} \tag{1}$$

The Levenshtein distance is usually implemented using a $n \times m$-matrix where n and m denote the length of the string a and b. Therefore, i and j are the rows and columns of the matrix. The values are calculated by iterating over every row and column while applying the above equation. Figure 3 shows an example of such a matrix.

		c	a	r	e
	0	1	2	3	4
c	1	0	1	2	3
a	2	1	0	1	2
r	3	2	1	0	1

Figure 3: Levenshtein - Matrix example [18]

2.5.2. Damerau-Levenshtein

Similarly to the Levenshtein distance, the Damerau-Levenshtein introduced by Fred Damerau in 1964 is also an edit distance metric with the difference to the Levenshtein that transpositions are also considered as a permitted operation step [19]. The equivalence of transposition in the Levenshtein distance would consist of two operation steps, a deletion and an insertion. In contrast, the transposition in the Damerau-Levenshstein distance is considered one step. A visual demonstration is shown in Figure 4 and 5

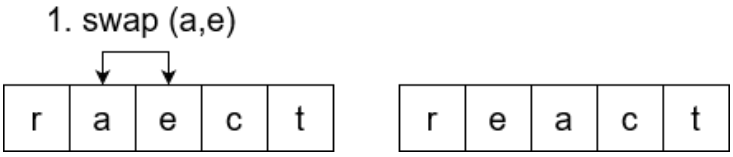

Figure 4: Damerau-Levenshtein example of a transposition

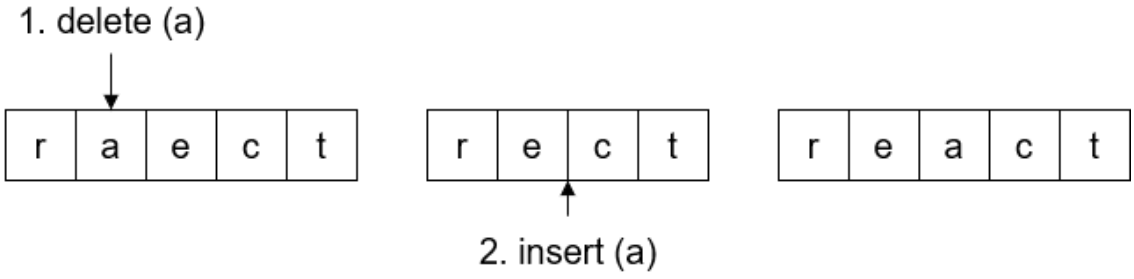

Figure 5: Equivalence of transposition in Levenshtein distance

Thus the following equation resembles the Levenshtein recursion with the addition of included transposition [17]:

$$d_{a,b}(i,j) = \min \begin{cases} 0 & \text{if } i = j = 0, \\ d_{a,b}(i-1,j) + 1 & \text{if } i > 0, \\ d_{a,b}(i,j-1) + 1 & \text{if } j > 0, \\ d_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} & \text{if } i,j > 0, \\ d_{a,b}(i-2,j-2) + 1 & \text{if } i,j > 1 \text{ and } a_i = b_{j-1} \text{ and } a_{i-1} = b_j, \end{cases} \tag{2}$$

2.5.3. Jaro-Winkler

Matthew Jaro introduced another string similarity metric called the Jaro similarity in 1989 [15]. Unlike the previously mentioned string similarity metrics (Levenshtein and Damerau-Levenshtein), the values of this metric are restricted to values between 0 and 1. If two strings are considered identical, then their similarity is indicated by the value 1. If no similarity between two strings exists, then this is indicated by 0. As the two strings become more similar, the function value approaches 1. The Jaro distance can be described using the following equation [20]:

$$sim_j = \begin{cases} 0 & \text{if } m = 0 \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) & \text{else,} \end{cases} \tag{3}$$

Where $|s_i|$ is the length of the string s_i. t denotes the number of characters not in the correct order divided by 2. Furthermore, m is the number of matching characters in both strings. A character matches if at least one of the following conditions is true:

1. The same character has the same index in both strings
2. The same character is no further away than k positions from each other

With $k = \lfloor \frac{\max(|s_1|,|s_2|)}{2} \rfloor - 1$.

In 1990, William Winkler presented a modified version of the Jaro similarity, the Jaro-Winkler similarity [21]. This version has the additional property that identical prefixes up to a length of l provide a positive weighting of p to the similarity score. The background of this is that the assumption exists that people would make fewer typing errors at the beginning of a word than at the end or in the middle of a word. Thus the modified variant has the following form [20]:

$$sim_w = sim_j + lp(1 - sim_j) \text{ where } 0 <= p <= 0.25 \tag{4}$$

2.5.4. Gestalt Pattern Matching

The Gestalt Pattern Matching algorithm, an algorithm developed in 1983 by John Ratcliff and John Obershelp is best explained with the following quote: "[...] people can recognize a pattern as a functional unit [...] by summation of its parts. [...] a person can recognize a picture in a connect-the-dots puzzle before finishing or even beginning it. This process [...] is called gestalt." [22]. This is precisely the principle on which this algorithm works. It tries to imitate the human intuition of word recognition. This is done via the following formula [20]:

$$D_{ro} = \frac{2K_m}{|S_1| + |S_2|} \tag{5}$$

K_m is the number of matching characters, while $|S_i|$ is the length of the string S_i. The number of matching characters is defined by finding the largest common substring. This process is then repeated recursively for the substrings to the left and right of the largest common substring. Each character of a common substring that is detected during this process is defined as a matching character.

2.6. Fundamentals of Machine Learning

This section introduces the machine learning fundamentals needed for the approaches and concepts used in this thesis.

2.6.1. Classification and Regression Trees

Classification and Regression Trees (CART), also known as Decision Trees, first discussed and suggested by Leo Breiman, are supervised learning methods used to perform classifications and regressions [23]. CARTs often use a binary tree as a data structure, displayed in Figure 6; this means that every node n except the root r has exactly one incoming and two outgoing edges. A decision is made in each node n of the tree, which is determined by a split criterion. It is examined here whether the incoming value conforms or disagrees with the split criterion in the node. The edges are then chosen by the decision made in the respective node. The first decision is made in the root r. The classifications or regressions are then found in the leaves ℓ of the tree. Figure 7 shows an example of a Decision Tree by Segaran [24].

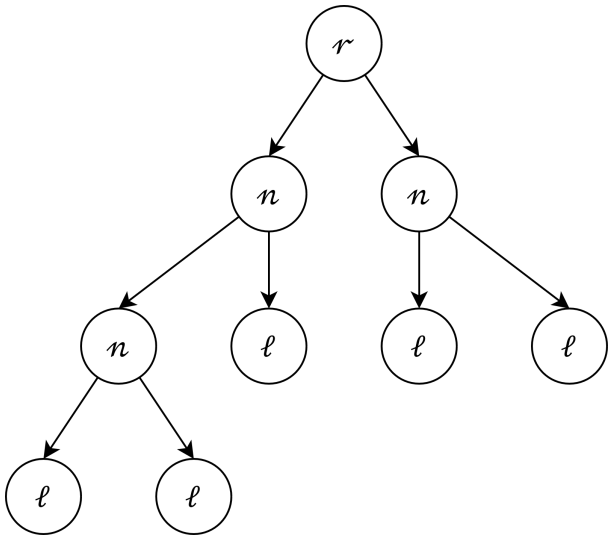

Figure 6: Binary tree example

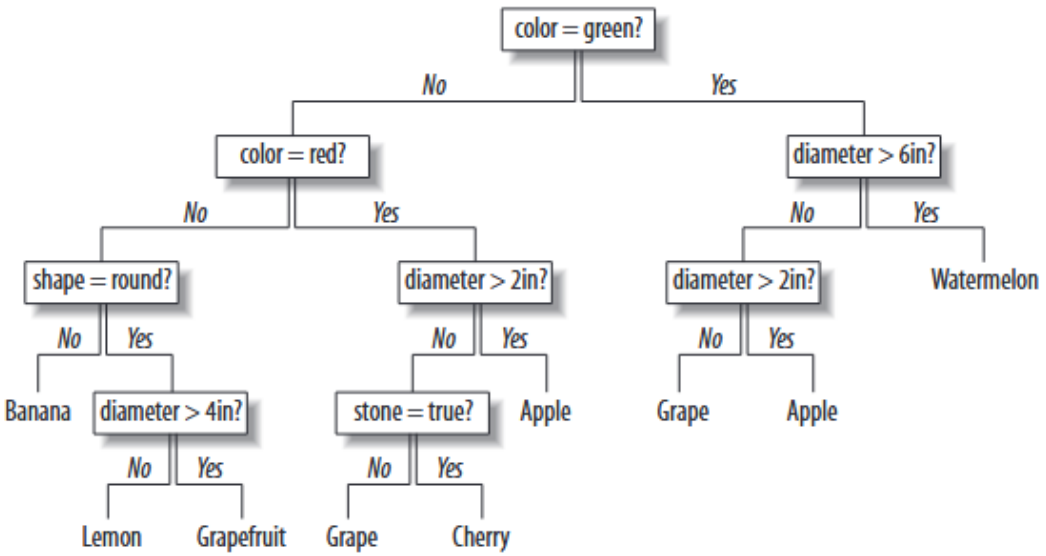

Figure 7: Decision tree example [24]

In order to determine the corresponding split criterion for each node, the datasets are systematically separated into two subsets according to their different attributes and for each domain of that respective attribute. One dataset contains all instances of that specific domain, while the other does not. Subsequently, the information gain is calculated [24]. The information gain is a measure for amount of information gained following a split. The Decision Tree works with the greedy approach and tries to perform splits based on the highest information gain. A metric for impurities is used to calculate the information gain, which is explained in more detail in the following section.

2.6.2. Measures of Impurity and Information Gain

As mentioned, it must be determined according to which criteria the nodes are dividing the respective datasets. In order to determine the splitting criteria, the information gain is used. The information gain describes how much the uncertainty changes after a split, with the size of the sets also factored into the weighting [25]. This can be best explained with the help of a visual example presented in Figure 8, given a set $\boldsymbol{M}$. In this set $\boldsymbol{M}$, the elements have one class *color* and two attributes, *shape* and *kernel color*. There are two class domains *blue* and *red*. The domains of the shape attribute are *square* and *circle*, and for the attribute, kernel color, the domains are *yellow* and *purple*. The goal is to determine the class of an element by the attribute shape and kernel color with high certainty. 50% of the elements are squares, and the remaining 50% are circles. If one randomly picks an element from this set $\boldsymbol{M}$, it is unknown whether it will be blue or red. The uncertainty, in this case, is maximized. Nevertheless, if the set is split by, for instance, shape, such that all squares are placed into one set $\boldsymbol{A}$ and all circles into the other set $\boldsymbol{B}$, one would pick up an element from the first set by chance. The certainty of the element being red would be maximized. Thus the information gain is also maximized, and the given split criterion minimizes uncertainty. A split by kernel color would also increase the information gain and reduce the uncertainty, but not to the same extent as a split by shape would do.

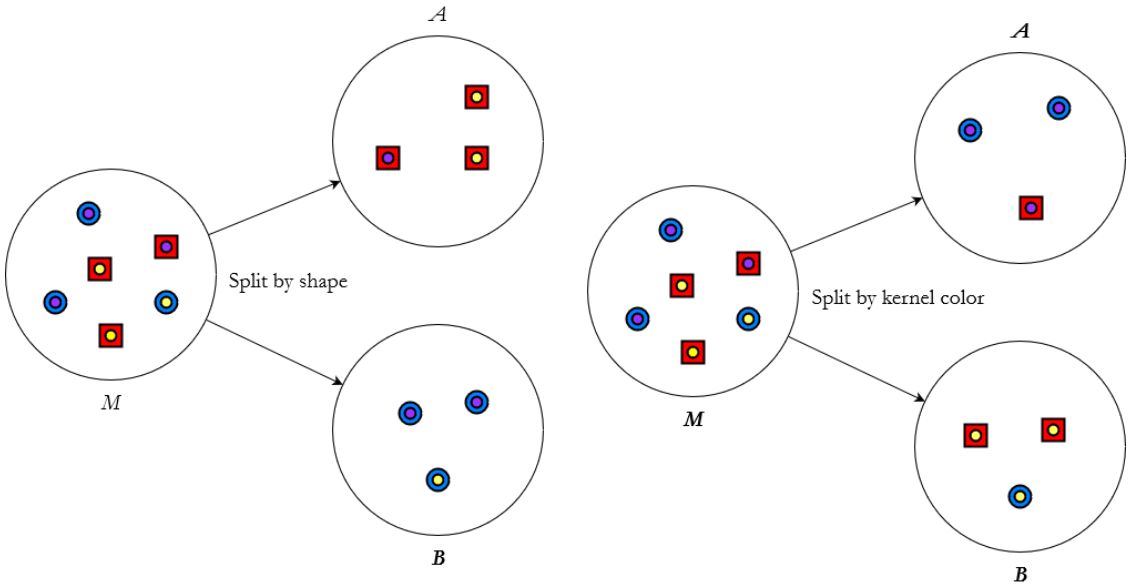

Figure 8: Split by shape vs. split by kernel color

In the literature, the two impurity metrics, Entropy and Gini coefficient, are frequently encountered to calculate the information gain. Entropy describes the disorder in a dataset and can be calculated by the following equation [26]:

$$E(D) = -\sum_{i=1}^{k} p_i \cdot log_2(p_i) \tag{6}$$

The Entropy of a set D is therefore defined by the negative sum of i over k where k is the number of classes (in our example Figure 8 this would be 2, red and blue). p_i denotes the ratio between the class i and the number of data in the whole dataset.
The value range of this metric is between 0 and 1. The higher the value, the higher the impurity [24]; the maximum impurity is reached with a value of 1. The Gini coefficient, on the other hand, describes the probability of a miss classification of an entity after being randomly chosen and is described with the following equation [27]:

$$G(D) = 1 - \sum_{i=1}^{k} p_i^2 \tag{7}$$

The notion for p_i and k is the same as the one mentioned in the Entropy metric.

To determine the information gain, one has to calculate the difference between the impurity of the dataset before the split and the average of the impurity of all child sets after the split. The equation for such a calculation is demonstrated below [24]:

$$I(X,Y) = Im(X) - \sum_{i=1}^{d} \frac{c(Y_i)}{c(Y)} \cdot Im(Y_i) \tag{8}$$

Thus the information gain is calculated by the parent set X, and the child sets Y where d denotes the number of children, $c(Y_i)$ the cardinality of a specific child set Y_i. And $c(Y)$ the cardinality of all child sets. $Im(X)$ is the impurity of the parent set X and $Im(Y_i)$ is the impurity of the child set Y_i. The function Im can either be the Entropy or the Gini coefficient.

2.6.3. Random Forest

Random Forest is an ensemble machine-learning method that uses Decision Trees for classification or regression problems. In a Random Forest, several randomized Decision Trees are created during the training phase, whereby a random set is formed from the data population for each tree. These sets are then used as input for the respective trees. Bootstrapping is one of the more popular methods to generate samples to train these Decision Trees [28]. The classifications of the various trees are collected and evaluated to a common conclusion further visualized in Figure 9 [29]. The advantage of Random Forests over Decision Trees is that they are often more robust and less susceptible to overfitting [28].

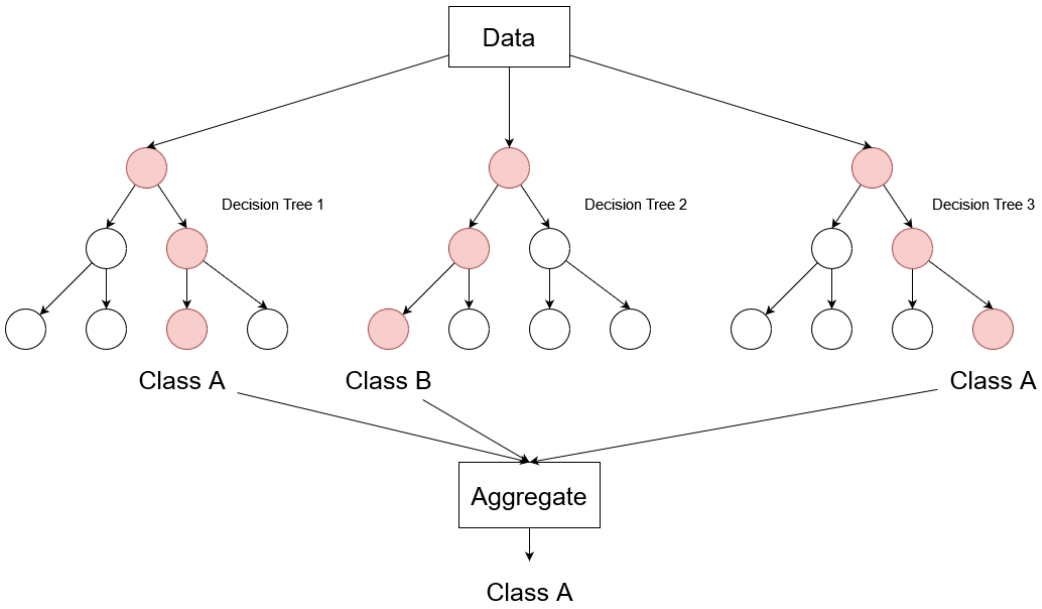

Figure 9: Random forest simplified example

2.6.4. Bootstrapping and Out-of-Bag

Bootstrapping is a method used to create multiple training datasets for Random Forest models. This can be achieved by taking randomized samples from the total dataset and selecting data randomly, which may result in the same data being chosen multiple times [30]. On statistical average, 63.2% of the data from the total population is used for a training dataset [29]. The remaining 36.8% of the data for each tree is referred to as the out-of-bag set, which is used as a validation set to estimate the model's performance during the training phase. This process is also known as out-of-bag estimation. Therefore, in bootstrapping, the out-of-bag samples are used during the training phase to estimate the model's performance. In contrast, the testing phase involves evaluating the model's performance on a separate, previously unseen test set [28].

2.6.5. Classification Report

The prediction of a binary classification model can be described as either a true positive, true negative, false positive, or false negative prediction [31]. A prediction is true positive if the model correctly predicted the existence of a property in an object and true negative if the model correctly predicted the absence of a property in an object. In contrast, a false positive in binary classification happens when the model wrongly predicts the presence of a property in an object when it is not there. A false negative arises when the model wrongly predicts the absence of a property in an object while it is present. A visual demonstration is shown in Figure 10.

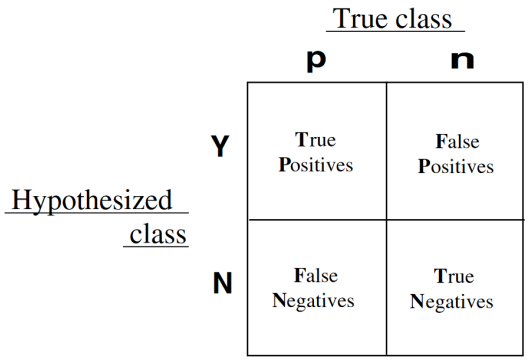

Figure 10: Confusion Matrix [31]

A classification report summarizes a machine learning model's performance in a classification problem. A classification problem involves the prediction of classes based on incoming data. Metrics such as Precision, Recall, F1-score, and Support are often included in the report and typically provide information on the model's ability to properly identify positive and negative samples for each class, as well as the model's overall accuracy [32]. Table 1 gives an example of a classification report. In this table, the model had to predict whether an object is of class A or of class B.

	Precision	**Recall**	**F1-Score**	**Support**
A	0.7	0.5	0.58	13
B	0.8	0.6	0.67	21
Accuracy			0.75	34
Weighted Avg.	0.76	0.59	0.63	34

Table 1: Classification report for Random Forest

Precision describes the ratio between true positive predictions for a given class and a class's total number of positive predictions [31]. Using the example provided above in class A, this means that 70% of all objects classified as A were true instances of class A. The recall is the ratio between true positive predictions and the total number of true instances of the class in the dataset [31]. Therefore, the recall value of 0.6 in class B describes that of the true instances of Bs in the dataset; 60% could be correctly identified as types of class B. The remaining 40% were not classified as instances of class B. The F1-score describes the harmonic mean between the precision and the recall. It is used to evaluate the overall performance of a model by giving both recall and precision the same weight [31]. However, a high F1 score can only be achieved if both performance metrics are high. If one metric is significantly lower than the other, then the weighting is stronger for the lower value [32]. For instance, given a precision of 0.2 and a recall value of 1.0, the resulting F1-score is, therefore, 0.33. The accuracy is the ratio between the number of correctly classified objects and the number of all objects [31]. For instance, an accuracy of 75% means that 75% of all objects were correctly classified into A and B. The Support denotes the number of true instances for each class. The Weighted Avg. describes the average for each category (Precision, Recall, F1-Score), weighted by the number of instances (Support). The equation for precision, recall, F1-score, and accuracy is given below [31].

$$\begin{aligned} \text{Precision} &= \frac{\boldsymbol{TP}}{\boldsymbol{TP} + \boldsymbol{FP}} \\ \text{Recall} &= \frac{\boldsymbol{TP}}{\boldsymbol{P}} \\ \text{F1-score} &= \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \\ \text{Accuracy} &= \frac{\boldsymbol{TP} + \boldsymbol{TN}}{\boldsymbol{P} + \boldsymbol{N}} \end{aligned} \tag{9}$$

TP denotes true positive predictions, FP false positive predictions, and TN true negative. P is the total number of positives, and N is the total number of negatives. Therefore P + N is the total population in a dataset.

3. Typosquatting Attacks

3.1. Introduction

Typosquatting is assigned to the area of social engineering [33]. Moreover, it originally stems from DNS (Domain Name System), where end-users are tricked into visiting websites they did not intend to visit in the first place. For this purpose, web addresses are registered that imitate real web addresses, except that these imitations also contain typing and spelling errors since the original web address is already occupied. An example is *google.com* and *googl.com*. Most Typosquatting domains in the DNS space are used for financial gain [34]; in contrast, attacks on package managers are mostly used to inject malicious code [35].

This thesis explores the notion of typosquatting in the context of SSCs. For software development in today's time, external libraries are often used for software products. These external libraries may already use external libraries themselves so that a whole tree of dependencies can develop, which is demonstrated in Figure 11. To use the library *vue*, for example, altogether, 20 other libraries must be installed, whereby only 5 of these 20 libraries have a direct dependence on *vue*. The remaining 15 are introduced into the library through transitive relations. Package managers are used to simplify and automate the administration of such packages and their dependencies.

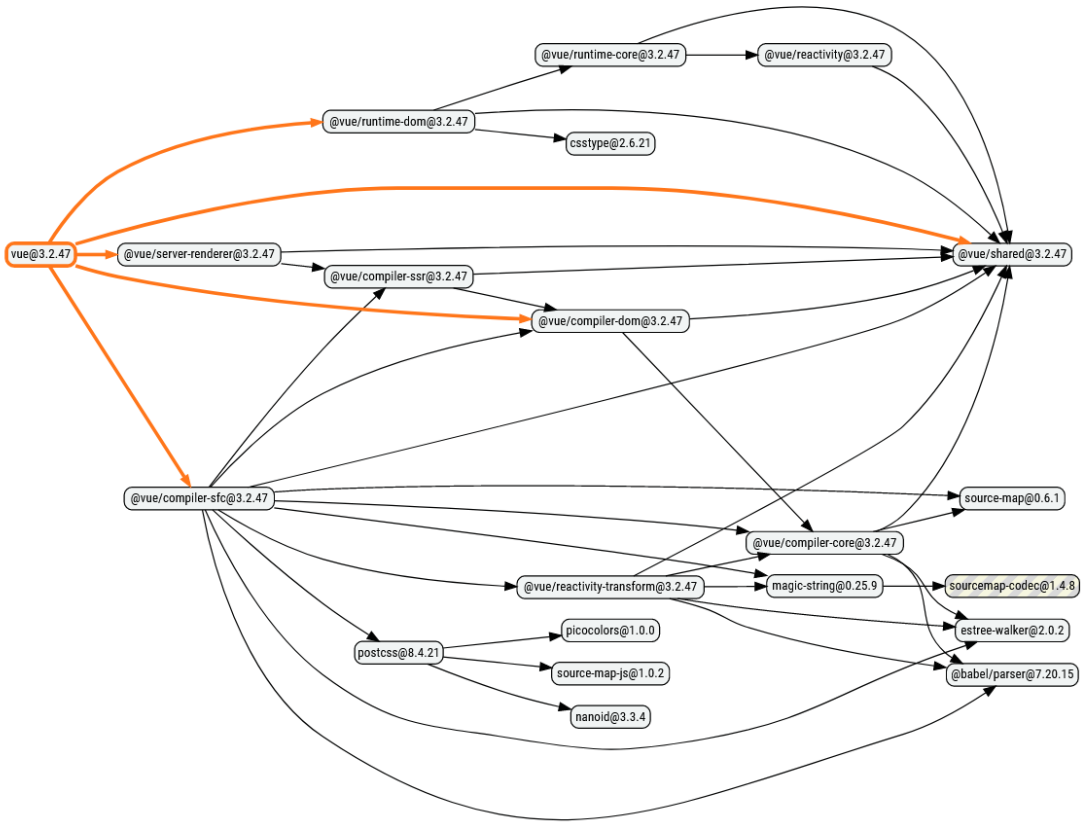

Figure 11: A dependency tree of the library *vue* [2]

As mentioned in the introduction, typosquatting attacks on package manager use software packages that imitate legitimate packages. This starts by selecting prevalent packages, then creating new packages that use a modified form of the popular package's name. Malicious code can then be inserted into this imitation. The malicious actor then may register the imitation via the package manager or directly on the package repository through the web. After registering the package, all developers who made the appropriate typo during installation now become victims. For instance, if an attacker wanted to imitate

[2]https://npmgraph.js.org/?q=vue (last visited: 12.01.2023)

the *vue* library of the npm ecosystem, all they would have to do is register new packages with the name *vu* or *veu*. Anyone typing *npm install vu* or *npm install veu* would then fall victim to this attack. Depending on the implementation type, the functionalities of the original library can also be imitated. This is achieved by either copying the source code, since this is accessible to everyone, or by specifying the original library as a dependency in the metadata (e.g. in *the package.json* for npm). This leads to a situation in which the victims only realize late or not at all that they have been targeted. A simplified interaction between victim and perpetrator is displayed in Figure 12.

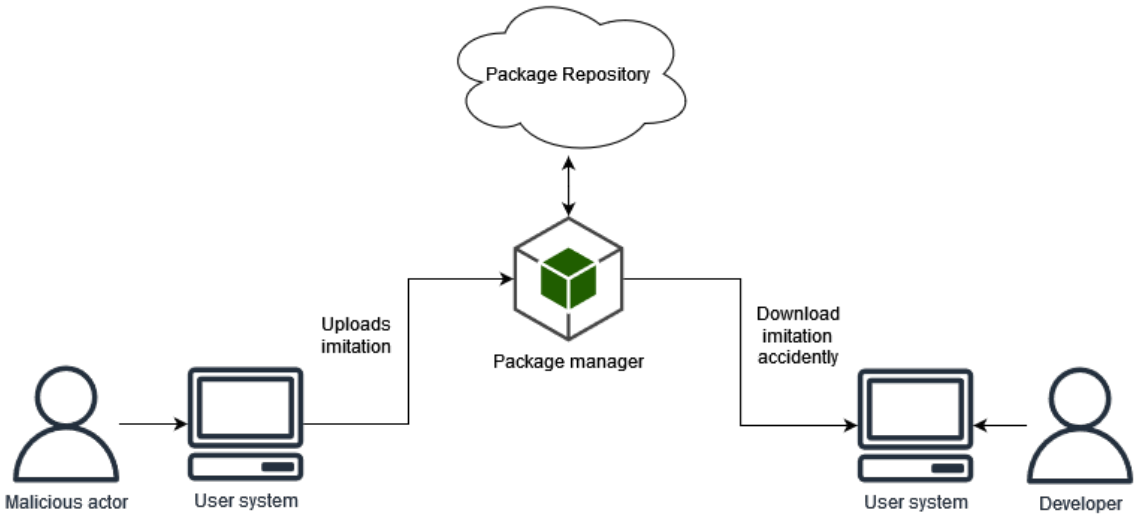

Figure 12: Simplified interaction between victim and actor

3.2. Differentiation of Typosquatting Types

There are different ways to categorize and define typosquatting types. For example, Tschacher's work divides them into three categories: Creative Typo-names, Stdlib Typos, and Algorithmically Determined Typo-names [13]. In the work of Meyers and Tozer, on the other hand, they are divided into two categories: Misspellings and Confusion Attacks [36]. The paper [37], in turn, distinguishes between Combosquatting and Typosquatting. In Tschacher's thesis, for example, Combosquatting would be a subset of the Creative Typo-names. There is no fixed definition for typosquatting attacks in the context of package managers. However, the consensus is that a package is a typosquatting package exactly when it imitates other packages on the metadata level and, in some instances, also at the functionality level.

Due to the various definitions, in the following, a thesis-specific definition of typosquatting will be introduced. Typosquatting packages are a subset of malicious packages, with the particular characteristic that they choose the imitation of legitimate packages as their propagation method. A package imitates another exactly when it is evident from the name or metadata of the package that there is a similarity between itself and another package. Not every package that resembles another package in name has malicious intent. It could be that the similarity is only coincidental. Therefore, an additional distinction must be made between typosquatting packages, i.e., packages that use the mentioned propagation method and have malicious intent, and typosquatting candidates, i.e., packages that have a similar name to other packages but do not have malicious intent.

In the paper [35], a quantitative analysis was performed on 174 malicious packages, and a set of primary goals of the malicious packages emerged iteratively from the analysis. These are:

- Opening Backdoors
- Exfiltration of data
- Placing a Dropper (placement of malware, which downloads further malware)
- Cause of Denial of Service
- Financial Gain through theft

In their quantitative analysis, malicious packages were also assigned to hybrid primary target sets, such as "Data Exfiltration & Backdoor". However, the existence of only one property is sufficient to classify such packages as malicious, which is why a package only needs to have at least one of these primary goals to have malicious intent attributed to it. Therefore a package has malicious intent exactly when at least one of the primary goals can be observed.

The mimicking of package names is classified into two main categories, *typographical errors* and *confusion*. Anything not assigned to the set typographical errors is assigned to the set confusion. A package name is assigned to the typographical errors set if it is intuitively clear that it is a typo or a clear similarity is evident. Usually but not exclusively indicated by the following pattern:

- Insertion (of neighboring keys): angular $\longrightarrow$ anguilar
- Deletion of characters: angular $\longrightarrow$ angula
- Transposition: angular $\longrightarrow$ angulra

If new semantics were added to the name, it would be assigned to the confusion category (the equivalence of the term combosquatting in the literature [37]). The reason for this categorization is that confusion is a gray area, where it is difficult to know by the package name with certainty whether the package is an imitation or whether it is a malicious package in general. Thus this thesis focuses on typosquatting attacks in the category of typographical errors. In the following, the term typosquatting packages refers to typosquatting packages leveraging typographical errors. Table 2 lists a few examples of typosquatting packages and their targets, as well as a categorization.

Imitation	**Original**	**Category**
5eact	react	Typographical error
crossenv	cross-env	Typographical error
fetch-node	node-fetch	Confusion
follow-rdeirects	follow-redirects	Typographical error
vue-style-gloder	vue-style-loader	Typographical error
twilio-npm	twilio	Confusion
colors-update	colors	Confusion

Table 2: Example of a categorization

3.3. Historical Typosquatting Attacks

3.3.1. Extraction of AWS Keys

In June 2022, Sonatype discovered several malicious packages [38]. The packages are *loglib-modules*, *pyg-modules*, *pygrata*, *pygrata-utils*, and *hkg-sol-utils*. For the former two, the targets are known, which were *loglib* and *pyg*. For the last three, the package to be imitated has yet to be discovered. The purpose of the malicious packages was to exfiltrate network information as well as environment variables, including AWS credentials, and send them to a remote host. The source code for this is displayed in Figure 13. The collected information on that host was publicly available. As a result, hundreds of text files containing sensitive information became publicly available. Figure 14 displays an excerpt of the sensitive data.

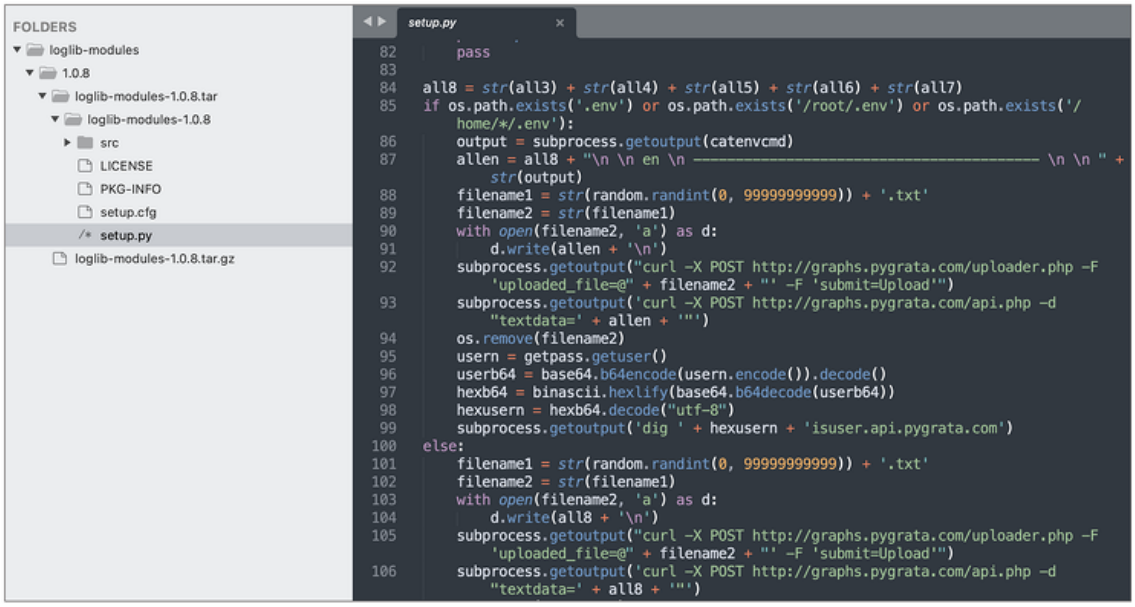

Figure 13: Malicious code present in loglib-modules [38]

Figure 14: Sensitive information uploaded by the malicious packages [38]

3.3.2. Stealing of SSH and GPG keys using two Typosquatting Packages

This attack was performed using the two typosquatting packages *python3-dateutil* and *jeIlyfish* (note that the first letter l in jeIlyfish is a capital i), both mimicked the *dateutil* and *jellyfish* packages. It is unlikely to mistype jellyfish to the point of swapping the letters l and i, which will be more clear in the following explanation. The attack was planned for almost a year. In this, the attacker first uploaded the imitation jeIlyfish on December 11, 2018; this package contained the malicious code to exfiltrate SSH and GPG keys of a user, examine some folder structures, and send them to an IP address [39]. For this, the obfuscated source code displayed in Figure 15 was used. This package remained undetected. After almost a year, the attacker registered a new package called python3-dateutil, which contained no malicious code. However, the package depended on the jellyfish imitation, which contains the malicious payload and is automatically installed by the package manager. Thus, the attacker used the python3-dateutil library as a Trojan horse for the introduction of the real malicious package jeIlyfish. Hence, a typo of the library jellyfish was not necessary to download the jellyfish imitation on a machine. Visually, it is also challenging to tell the difference between jeIlyfish and jellyfish.

```
1  import zlib
2  import base64
3
4  ZAUTHSS = ''
5  ZAUTHSS += 'eJx1U12PojAUfedXkMwDmjgOIDIyyTyoIH4gMiooTmYnQFsQQWoLKv76rYnZbDaz'
6  ZAUTHSS += 'fWh7T849vec294lXexEeTOXT6ScXpawkk+C9Z+yHK5JSPL3kg5h74tUuLeKsK8aa'
7  ZAUTHSS += '6SziySDryHmPhgX1sCUZtigVxga92oNkNeqL8Ox5/ZMeRo4xNpduJB2NCcROwXS2'
8  ZAUTHSS += 'wTVf3q7EUYE+xeVomhwLYsLeQhzth4tQkXpGipPAtTVPW1a6fz7oa2m38NYzDQSH'
9  ZAUTHSS += 'hCl0ksxCEz8HcbAzkDYuo/N4t8hs5qFOKtzHZxXQxBnXkXhKa5Zg18nHh0tAZCj+'
10 ZAUTHSS += 'oA+L2xFvgXMJtN3lNoPLj5XMSHR4ywOwHeqnV8kfKf7a2QTEl3aDjbpBfSOEZChf'
11 ZAUTHSS += '9jOqBxgHNKADZcXtc1yQkiewRWvaKij3XVRl6xsS8s6ANi3BPX5cGcr9iL4XGB4b'
12 ZAUTHSS += 'BWODeD5WWdYSLqHQbP2IciWp3zj+viNS5HxFsmwfyvyjEhbeOzgeXiOIy785bQJP'
13 ZAUTHSS += 'FaTlP1T+zoVR43anABgVOSaQ0kYYUKgq7VBS7yCADQLbtAobHM8T4fOX+KwFYQQg'
14 ZAUTHSS += '+hJagtB6iDWEpCzx28tLuC+zus3EXuSut7u6YX4gQpOVEIBGs/1QFKoSPfeYU5QF'
15 ZAUTHSS += 'MX1nD8xdaz2xJrbB8c1P5e1Z+WpXGEPSaLLFPTyx7tP/NPJP+9l/QteSTVWUpNQR'
16 ZAUTHSS += 'ZbDXT9vcSl43I5ksclc0fUaZ37bLZJjHY69GMR2fA5otolpF187RlZ1riTrG6zLp'
17 ZAUTHSS += 'odQsjopv9NLM7juh1L2k2drSImCpTMSXtfshL/2RdvByfTbFeHSOC29oyPiwVVNk'
18 ZAUTHSS += 'Vs4NmfXZnkMEa3ex7LqpC8b92Uj9kNLJfSYmctiTdWuioFJDDADoluJhjfykc2bz'
19 ZAUTHSS += 'VgHXcbaFvhFXET1JVMl3dmym3lzpmFv5N6+3QHk='
20
21
22 ZAUTHSS = base64.b64decode(ZAUTHSS)
23 ZAUTHSS = zlib.decompress(ZAUTHSS)
24 if ZAUTHSS:
25     exec(ZAUTHSS)
```

Figure 15: Malicious code present in jeIlyfish [40]

4. Current State of the Art and Related Literatures

This chapter summarizes recent literature on typosquatting and malicious package detection. Furthermore, it provides an outlook on related fields in the detection of typosquatting packages, e.g., approximate string matching.

4.1. Effectivity of Typosquatting Packages

The topic of security in package managers is more relevant than ever, with the two most significant attack vectors in 2021 on such systems being typosquatting and dependency confusion, which have been steadily increasing since 2022 [6], [8]. Researchers are also actively investigating different approaches in detecting such malicious packages in these ecosystems, e.g., the paper titled *Practical automated detection of malicious npm packages* was recently published in 2022, indicating an actively researched topic [41].

One of the first pieces of literature about typosquatting attacks on package managers was done by Nikolai Tschacher and published in 2016 [13]. In his work, Tschacher demonstrated the effectiveness of attacks on package managers using typosquatting packages he created himself. He was thus able to infiltrate over 17000 host systems and had access to administrative privileges on 43.6% of these systems. Tschacher classified typosquatting attacks into three categories and provided the following examples: Creative typo names (e.g., coffe-script instead of coffee-script, since coffe-script is more effective than cofee-script), Stdlib (e.g., urllib2) and algorithmically determined typo names (req7est instead of request). The author also demonstrated that typosquatting packages assigned to the Stdlib category accounted for 95.6% of installations and were almost only effective in PyPI. By now, the package manager PyPI has added restrictions regarding such packages. Package names that resemble native python libraries are now prohibited [3]. Therefore, the effectiveness of Stdlib typosquatting packages should be reduced. On npm, on the other hand, only algorithmically created and creative typosquatting names were successful, despite the lower number of installations. The lack of Stdlib typosquatting packages on npm is probably the result of the fact that npm already had a list of reserved package names at that time [4].

A similar empirical analysis was performed in the literature [42], where about 5000 typosquatting variants of popular Docker images were created and uploaded. These typosquatting variants were created mainly using typographical errors such as duplication and Fat-Finger (insertions), deletion, and permutation (transposition). The literature also mentioned misinterpretation (confusion of similar-looking letters and numbers, for instance, 1 and I) as one of its typosquatting variants. However, this variant is not considered in this thesis due to the nature of typos and this variant being highly unlikely and usually inferred by copying the package name instead of typing it. In 210 days, they obtained over 40000 pulls. However, only a handful of the typosquatting variants constituted the majority of the downloads, while most of them only had up to 5 downloads each.

4.2. Related Literature in Detection & Mitigation of Malicious Packages

As indicated in chapter Differentiation of Typosquatting Types, two properties must be met for a package to be classified as typosquatting.

1. It should be evident from the name that a typosquatting attack takes place. The first step is, therefore, the identification of typosquatting candidates.
2. A malicious intent in the source code must be recognizable

If both conditions are met, then the respective package is considered a typosquatting package. This structure is also reflected in the literature dealing with typosquatting attacks, further confirming the differentiation between typosquatting packages and typosquatting candidates [37], [43]. On the one hand, some works of literature focus on the detection

[3] https://pypi.org/help/#project-name (last visited: 14.01.2023)
[4] https://www.npmjs.com/package/validate-npm-package-name/v/2.2.2 (last visited: 14.01.2023)

based solely on the package's name to find typosquatting candidates, annotated as category (*a*). On the other hand, there is literature that has the general detection of malicious code in packages as their objective (*b*), but hybrid approaches, in which both approaches are combined to find such typosquatting packages, also exist (*c*). In the following, a brief overview of the existing literature is given in chronological order, as well as a classification of the literature.

4.2.1. Malicious Code Detection using Abstract Syntax Trees

The bachelor thesis [44] published in 2019 focuses on the PyPI ecosystem, using static code analysis and the Damerau-Levenshtein distance. Therefore it can be assigned to category *c*, methods that follow both approaches, with the focus on detecting malicious code. Investigations on the package name are merely secondary. Using the Damerau-Levenshtein distance with a maximum distance of 2, the author examined the 10000 most popular packages and discovered 21230 typosquatting candidates. For the analysis and discovery of malicious packages, the author pursued the approach of static code analysis. In particular, Abstract Syntax Trees (AST) formed its foundations. In the process, the Python source code is transferred into an AST and transformed using reduction rules. The source code located in the *setup.py* is used for the AST. The setup.py is automatically executed during the installation of the corresponding package. Subsequently, the AST is traversed. Should a match with a semantic rule occur during the traversal, then this is registered with further metadata. The Aura framework has been implemented, which was used to scan the entire PyPI repository. After the execution of a global scan using the Aura framework, a high number of false positives were detected, and no malicious packages could be identified.

4.2.2. Detection of Suspicious Package Updates

The paper [45], which was published in 2019 and can be categorized as belonging to category *b*, examines the payload rather than the package name. The work's primary goal is not to detect typosquatting packages but rather malicious code injections through updates of existing packages. This work uses the anomaly detection approach, in which unnatural updates are detected by their characteristics and behavior. Features were extracted from a package's metadata and source code for anomaly detection. The package manager in focus here is npm. The following features were extracted:

- The use of the libraries: *http*, *http2*, *https*, *net*, *fs*, *child_process*
- The use of the *eval* function
- Whether new JavaScript files were added
- Whether new dependencies were added
- Whether a new hookup script was added

For this, 1518 packages were clustered and used representatively for regular package updates. An outlier threshold was then determined using the distance between the centroid (center of a cluster of data points) ***A*** and the furthest data point of that respective cluster ***B***. If a new data record is inserted into the cluster model and it turns out that the distance between this data record ***C*** and the nearest centroid ***A*** is greater than the distance between the centroid ***A*** and the farthest distant data point ***B***, it is classified as suspicious. A visual example is displayed in Figure 16. The model flagged 539 updates as suspicious on a weekly basis, but no evaluation of the accuracy has been performed. Instead, the model was successfully tested on the library *eslint-scope*, a legitimate library that was injected with malicious code in the 3.7.2 update[5].

[5]https://security.snyk.io/vuln/npm:eslint-scope:20180712

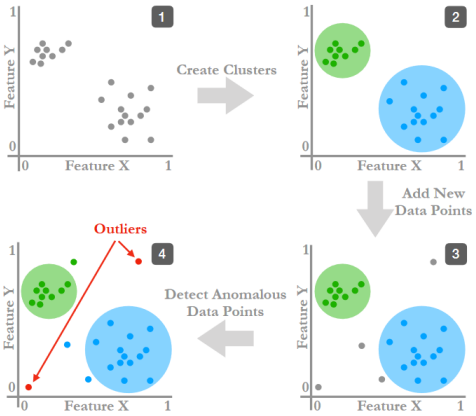

Figure 16: Clustering of benign and suspicious package update [45]

Another work in a similar domain is the literature [46]. It deals with the detection of anomalies during the introduction of malicious code. Expressly, it assumes and examines whether malicious packages introduce more artifacts during installation than benign packages. Contrary to the previous work, which uses machine learning procedures, dynamic code analysis techniques are used. Another difference is the scope their methods should apply on. Instead of scanning the package manager's ecosystem, the proposed framework should be integrated into a local development environment. For this, eight benign packages, which were attacked over time and injected with malicious code, serve as the data basis. The benign versions are then compared with the malicious version as well as their artifacts. Examples of such artifacts would be whether files were created or an internet connection was established. The packages' activity was monitored in a sandbox environment, which meant that system calls made by the package were recorded. The resulting observables or artifacts are then extracted from the system calls. The differences between the artifacts introduced by benign and malicious packages were analyzed to determine the responsible artifacts for malicious behavior. The result of this work is that the hypothesis of whether malicious updates introduce more artifacts than their benign counterpart could be confirmed; thus, most malicious versions introduced about 225% more artifacts than benign versions.

4.2.3. Detection of Typosquatting Packages using Signals

Two works can be included in this section, namely [47] and [12]. Both works can be assigned to category *a*, papers that detect typosquatting packages based solely on their name and eventually metadata. At first, the Levenshtein distance was used to detect typosquatting candidates, but this led to high false positives, which prompted a new approach. The proposed method is based on six trigger signals. If any of these six signals are triggered, the respective pair is marked as a typosquatting candidate. The triggers used in the literature are:

- Repeated characters (e.g. requeest attacks request)
- Omitted characters (e.g. comander attacks commander)
- Swapped characters (e.g. axois attacks axios)
- Swapped words (e.g. import-mysql attacks mysql-import)
- Common typos (substitutions based on physical locality e.g. uglify.js attacks uglify-js)
- Version numbers (e.g. underscore.string-2 attacks underscore.string)

In order to reduce the false positive rate, additional metadata of the respective packages were used, whereby in particular, the popularity or the number of the downloads is taken into account. The method mentioned here was capable of detecting approximately 60% of the typosquatting packages known so far, marked as such by npm. The 60% is due to

the difference in the definition of typosquatting packages between the researchers and the npm team. This framework's primary intention is to extend the installation process by introducing a verification step displayed in Figure 17.

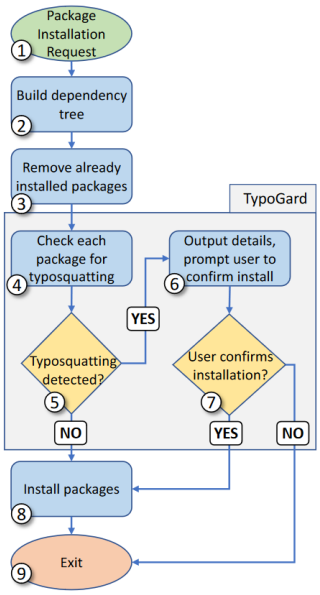

Figure 17: TypoGuard installation process [47]

4.2.4. Typosquatting and Combosquatting Attacks on the Python Ecosystem

The paper by [37], examines and identifies typosquatting and combosquatting packages via their names and metadata and is therefore located in category *a*. As the name suggests, the focus was limited to the PyPI repository. First, the following assumptions are made:

- Packages whose repository names on GitHub are identical are not squatting packages
- Packages with different names on the GitHub repository side and in PyPI need additional verification.

The Levenshtein distance is used to find typosquatting candidates, where the upper bound for the number of operation steps is set to two. This was determined heuristically based on 36 already known typosquatting packages. With this step size, 21 of the 36 squatting packages can be identified while the number of false positives is reduced. Furthermore, the package names are normalized to discover combosquatting packages (classified as confused in this thesis). Thus, frequently occurring prefixes of squatting packages, such as "python", are replaced by "*". This research has resulted in the process depicted in Figure 18.

This process was used to analyze 216,548 packages from the PyPI ecosystem. The findings were then statistically examined. An upper bound on the Levenshtein distance of 2 proved promising, as most typosquatting packages could be detected while keeping the number of false positives manageable. Although the number of false positives was not specified in more detail. A false positive could occur when the package name and GitHub repository name differ because the package's name might have already been taken. It has also been found that the existence of packages whose names have similarities to modules from the standard python libraries is not necessarily malicious. In the end, the following approaches have been suggested to reduce the number of false positives:

- Analysis of package metadata (e.g., author reputation, package popularity)
- Suspicious source code characteristics (e.g., suspicious API calls)

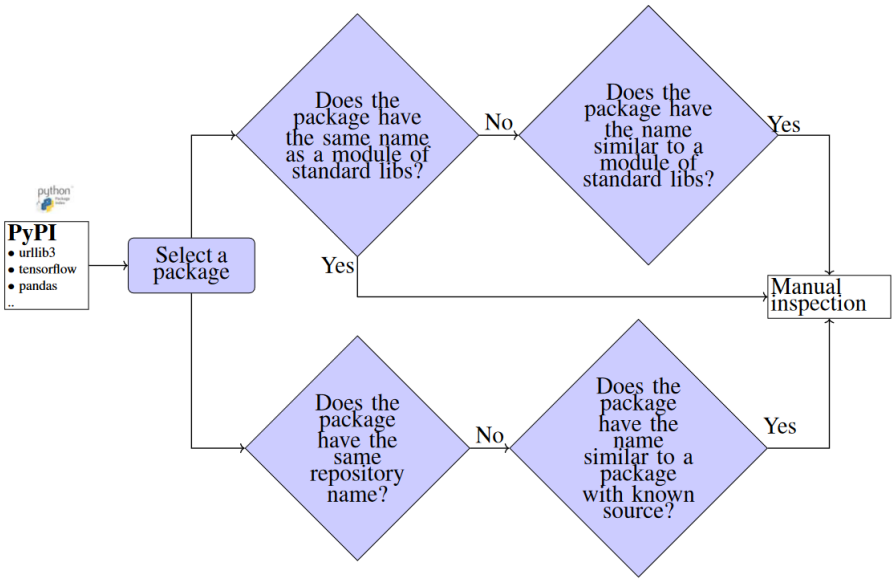

Figure 18: Indentification logic of squatting packages [37]

4.2.5. Malicious Package Detection Framework - MalOSS

The paper [43], belongs to category *c* and identifies typosquatting packages by package name and payload. While the framework is intended to detect malicious packages in general, it has explicitly included procedures to detect typosquatting packages. For this purpose, the framework MalOSS was introduced, which uses different areas of code analysis. The framework consists of four components: The metadata analysis, the static analysis, the dynamic analysis, and the verification phase, in which packages marked as suspicious are manually verified. The metadata analysis includes the examination of the package name with the use of edit distances, which were not specified in more detail. Nevertheless, metadata information such as author, downloads, dependencies, and what types of files are contained in the package are examined. In the static code analysis, the source code is analyzed with the help of three subcomponents: Manual API Labeling, API Usage, and Dataflow Analysis. In principle, during this stage, it examines which interfaces to the system the packages use, e.g., which libraries were used, and which semantics and behavior can be inferred from the source code. With the dynamic code analysis, the package is analyzed and observed during the runtime in a closed system. Thus one can determine the behavior of a package, for example, whether a library establishes a connection to the internet without understanding the source code. In addition to providing the MalOSS framework, a qualitative analysis of similarities and differences in the package managers PyPI, npm, and RubyGems was carried out. MalOSS used the following three heuristic groups:

- Metadata Analysis
  - Similar package names to other popular ones in the same package repository.
  - Same package names but different authors across package repositories.
  - Author of the package is known to have been involved in the distribution of malicious packages.
  - Package has older versions released around the time as known malware.
  - The package contains Windows PE files or Linux ELF files.
- Static Analysis
  - Package has customized installation logic.
  - Package adds recently released versions of network, process, or code generation APIs.

  - Package has flows from filesystem sources to network sinks.
  - Package has flows from network sources to code generation or process sinks.
- Dynamic Analysis
  - Package contacts unexpected or suspicious IPs or domains.
  - Package reads from sensitive file locations.
  - Package writes to sensitive file locations.
  - Package spawns unexpected processes.

This framework resulted in the discovery of 339 malicious packages that were previously undetected, spread across three different package managers, PyPI, npm, and RubyGems.

4.2.6. Supporting the Detection of Software Supply Chain Attacks through Unsupervised Signature Generation

Another paper [48] can be assigned to category *b* focusing on the payload [48]. The proposed approach works under the assumption that most attacks on those SSCs occur in waves. In a short time period, a large number of malicious packages are injected. Therefore malicious codes of the packages are often similar or even the same as one another. Additionally, similarities to packages from previous attack waves can also occur. In principle, clustering methods are used in which similar malicious code variants are grouped together. For this, the researchers examined the approach an analyst would perform to identify malicious packages; the process is as follows:

1. The analyst gets informed about the presence of a suspicious package
2. The analyst searches through all code files for suspicious code fragments
3. The suspicious code fragments are compared to malicious code fragments in the past, forming malware families
4. With this knowledge of malware families, the analyst scans the package repository to detect potential members of the malware families

The proposed approach attempts to automate steps 3 to 4 using clustering algorithms. Various clustering methods and their performance have been compared with each other. This examination has led to the conclusion that the Markov clustering method, in connection with the transformation of the source code into an AST, has led to the best result. An F1 score of 0.9851 has been achieved. As a data basis for the evaluation of the different procedures, 114 malicious packages from the datasets provided by the paper [35] were used. Afterward, this method was additionally tested on the entire npm ecosystem. When tested on the npm ecosystem, seven packages were discovered, four of which were confirmed as actual malicious packages. Two packages were labeled proof-of-concept and one non-malicious but dependent on a malicious library. The advantage of this method is that it is resistant to type 1 and type 2 modifications of code fragments (the renaming of variables and literals) and, thus, against simple obfuscation techniques.

4.2.7. Practical Automated Detection of Malicious NPM Packages

The paper [41] uses machine learning as well as various other methods for detecting malicious packages in general and is therefore classified in category *b*. The proposed process is called *Amalfi* and consists of three components, a classifier, a reproducer, and a clone detector. The features used for training the classifier are:

- Access to personally-identifying information (PII)
- Access to specific system resources
- Use of specific APIs
- Use of package installation scripts
- Presence of minified code (to avoid detection) or binary files

Three different machine learning methods were used and compared for the classifier: Decision Trees, Naive Bayes, and Support Vector Machines. These have the task of deciding

whether the package is benign or malicious. Depending on how the classifiers have classified the package, the clone detector and reproducer components do the rest of the analysis. Their primary purpose is to reduce the number of false positives. The reproducer checks whether the source code of a package can be reproduced with the repository provided by the maintainer. If that were the case, the package would be considered benign. At the same time, the clone detector creates a hash value for the packages and checks whether this is identical to the hash values of malicious packages that have already been detected in the past. The Amalfi model was then checked for the latest packages for seven days. The following results were obtained.

Date	# Versions	Decision Tree		Naive Bayes		SVM		Clones
		# TP	# FP	# TP	# FP	# TP	# FP	# TP
July 29	23,452	34+1	932-74	13+22	1453-107	20+5	102-11	0
July 30	13,849	2+0	22-1	0+0	6-0	0+0	14-3	0
July 31	7,042	17+0	16-1	0+0	1-0	18+9	4-0	0
August 1	6,050	1+0	6-2	1+0	3-2	0+0	13-0	0
August 2	13,562	2+1	17-0	4+1	12-1	0+0	10-0	1
August 3	15,269	6+0	15-2	1+0	9-1	0+0	41-3	0
August 4	17,063	16+2	9-1	1+0	9-1	1+0	10-1	17

Table 3: Result of Almafi's analysis [41]

As one can see, on the first day in Table 3, 23452 different package versions were analyzed, 35 of which turned out to be real malicious packages and therefore labeled as true positives. 858 false positives were detected, meaning packages were found to be malicious but were benign. As a result, some adjustments were made so that on the second day, the number of false positive messages was reduced. In total, Amalfi was able to detect 95 previously unknown malicious packages. Afterward, the model's suitability was re-evaluated on a dataset that had already been labeled. The results were as follows:

Dataset	Decision Tree		Naive Bayes		SVM	
	Prec.	Recall	Prec.	Recall	Prec.	Recall
Basic	0.98	0.43	0.90	0.19	(0.98)	(0.27)
MalOSS	0.35	0.64	0.62	0.64	0.73	0.61

Table 4: Amalfi's classification report [41]

As indicated in Table 4, the evaluation of Amalfi on the dataset used to train and test Amalfi (Basic) has a precision of at least 90% in all three machine learning methods. That means that the packages that were classified as malicious were actually malicious in 90% of the cases. However, the recall rate was relatively low at 19 - 43%. This means that only 19 - 43% (depending on the classifier) of the total number of malicious packages in the dataset could be correctly identified as such, leaving the rest undetected. The precision of Amalfi reduced to 35% when tested on the dataset provided by the literature "Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages," while recall improved by approximately 20%.

4.3. Approximate String Matching

For detecting typosquatting candidates, the Levenshtein or variations of it were mostly used in the examined literature. The problem with which one deals here is also called *Approximate String Matching* and was also examined frequently in the past [14]. Especially in the area of databases and record linking, where one tries to map the datasets from

different databases to each other, as well as in the census, such methods are frequently employed. For instance, by mapping the datasets correctly to each other despite typing errors [49]. The work [50], concerns itself with comparing four different distance metrics. For this purpose, the metrics were used to link datasets from two different hospitals. The four metrics used are the Levenshtein distance, Longest Common Substring, Jaro-Winkler similarity, and RMS, a combination of the previously mentioned metrics. The result of the work indicated that the Jaro-Winkler similarity had the highest number of true positives.

The work [20], on the other hand, compared the Jaro-Winkler similarity with the Ratcliff/Obershelp algorithm, also known as Gestalt-Pattern-Matching [22]. For the comparison, two databases were taken as the basis of correct words. One database contained 236000 entries, and one with 58000 entries. Subsequently, five test subjects were asked to type text passages from various sources, e.g., Wikipedia. Correcting the typing errors was not allowed. The algorithms now had to find the three most similar words based on the mistyped word. These algorithms were then evaluated depending on the position of the correct words in the list. The algorithms were assigned a score from 0 up to 3 points. Three points were awarded if the right word was in the first place, and 0 points if the right word was not in the list. The result was that the Ratcliff/Obershelp or Gestalt-Pattern algorithms performed 4-18.6% better than the Jaro-Winkler similarity.

4.4. Key Takeaways

Literature	Category	Methods used for names	Method used for payload	Year
Attacks on Package Managers	Malicious code, Package naming	Damerau-Levenshtein	Static code analysis	2019
Detecting Suspicious Package Updates	Malicious Code	None	k-means Clustering	2019
Towards Detection of Software Supply Chain Attacks by Forensic Artifacts	Malicious Code	None	Dynamic code analysis	2020
SpellBound: Defending Against Package typosquatting	Package naming	Custom defined 6 Signals, Levenshtein, Metadata analysis	None	2020
Typosquatting and Combosquatting Attacks on the Python Ecosystem	Package naming	Levenshtein, Metadata	None	2020
Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages	Package naming, Malicious Code	Not specified	Metadata analysis, static code analysis, dynamic code analysis	2021
Supporting the Detection of Software Supply Chain Attacks through Unsupervised Signature Generation	Malicious Code	None	Markov Clustering, static code analysis	2021
Practical Automated Detection of Malicious npm Packages	Malicious Code	None	Decision Trees, Naive Bayes, SVM, Reproducer, Clone Detection	2022

Table 5: Short overview of the mentioned literatures and their methods

As one can see from the specified literature, there are various approaches for detecting both typosquatting candidates and malicious code. However, only a few of them provide more detailed information about the accuracy and effectiveness of the proposed method [41], [48]. On the one hand, this is due to the scarcity of data, as upon discovery of malicious packages, these are immediately removed by the package manager authorities. On the other hand, it is in the nature of such problems that one does not know the quantity of actual malicious packages or benign packages in those repositories, which is why it is hardly possible to evaluate the effectiveness and accuracy in greater detail if applied to real-world environments. Thus, the only way to provide the presented method with more accurate metrics would be to run the method on already existing datasets. The work [35], is one of the few to offer a larger dataset of malicious packages.

Regarding typosquatting candidate detection using distance metrics, the Levenshtein distance and the Damerau-Levenshtein distance are the preferred candidates [37], [44]. Even though the problem of approximate pattern matching is being addressed here, literature in the field of record linking as well as spelling correction provides an outlook on other metrics that could be more effective than the methods tested so far. A combination of the (Damerau-)Levenshtein distance and the method proposed in the paper [47] would be worth an attempt. One of the reasons is that (Damerau-)Levenshtein distance does not take into account the characters' positions on a keyboard as in the literature [47] does. Section 6.2 explains in more detail the beneficial inherent characteristics of the Damerau-Levenshtein distance. Therefore, modifying the (Damerau-)Levenshtein distance could provide positive results. From the thesis [13], one can additionally infer that only a minor part of his typosquatting packages accounted for a significant part of the downloads. Whether such an observation is also the case for other typosquatting packages should be investigated. If such an assumption can be confirmed, it is possible to apply the methods specifically to those effective typosquatting packages, which have at least one crucial advantage. It would possibly reduce the number of false positives since one does not seek to find as many typosquatting packages as possible (which would be ideal but might cause a high false positive rate), but only those that are particularly effective. "Effective typosquatting packages" refers to packages that successfully trick users into downloading or using them by mimicking legitimate software or packages. Therefore a possible measurement is the number of victims deceived by that particular package. Moreover, effective typosquatting packages must be distinguishable from other typosquatting packages by certain properties. The more specific the search criterion is, the lower the false positive rate should be.

When it comes to literature comparing the various package managers, there is a need for more of these, whereby by package manager here explicitly application package managers are meant, which manage libraries for software development. Only one piece of literature could be found here, in which at least the package managers npm and PyPI are examined, namely [43]. The majority of research on package managers tends to focus on those that manage operating system tools, like apt for Linux [51], [52]. There are two crucial differences that present issues with this type of package manager. The first one is the smaller number of uploaded packages. The second and more crucial difference is that they are strictly regulated, and only authorized developers can upload to these repositories. Besides the permission, the proposed package to be uploaded is also verified by a trusted maintainer.

Rice's theorem states that it is impossible to algorithmically classify a package as either malicious or benign due to its close relationship with the undecidable halting problem. However, there are alternative approaches to identifying potentially malicious packages, such as using machine learning algorithms or statistical analysis to detect specific characteristics [41]. While these methods have shown promise in improving the overall detection rate of malicious packages, it is important to note that there may be other approaches as well that can be explored [43], [46]. Most of the works of literature presented here have one thing in common. In this respect, they extract features or certain characteristics from the malicious code and localize them into the categories "indicates malicious intent" or "is benign, is safe". Only the selection of features, their form, and their further purpose differ. How the extracted features are used is relatively balanced between machine learning and static/dynamic code analysis. The most common characteristic extracted from the packages is the detection of suspicious API calls either by static code analysis or dynamic analysis. The two works, which evaluated their procedures based on already known malicious packages and provide predictive values like a classification report, are, on the one hand, [48] and on the other hand [41]. Both provided good results when evaluated on already labeled datasets. However, there is a significant drawback to the paper [48]. Only malicious packages that have a high similarity or are identical to previously known

malicious packages are detected in this case. Malicious packages whose source code differs from the previously found packages will not be detected.

While the paper [41] provides a more robust approach, the evaluation of the proposed method on an unknown dataset resulted in a lower precision. Therefore the following methods (grouped by the scope of the problems) are proposed:

1. A dataset of typosquatting packages will be collected with their metadata and source code.
2. Package manager comparison
   a) A qualitative analysis will be performed on various package managers in regard to their vulnerability and properties.
   b) Derivation of a correlation between package manager properties and their vulnerability.
3. Measuring the effectiveness of various typosquatting packages and the detection of typosquatting candidates.
   a) A statistical examination of the collected typosquatting packages will be performed on their names and their effectiveness.
   b) Using the statistical insights, a threshold for the five proposed string metrics should be derived.
   c) Using the string metrics and their derived threshold, an evaluation of real-world packages of the npm ecosystem should be performed.
4. Detection of malicious intent in typosquatting candidates
   a) The dataset of typosquatting packages will be examined for commonalities and differences in the maliciousness of a package.
   b) Selected features of malicious packages will be proposed based on the commonalities of typosquatting packages.
   c) A dataset based on the proposed features will be generated and used to train a Decision Tree model while additionally extending them through a Random Forest model.
   d) Evaluate the models' accuracies based on the datasets collected for this thesis (internal dataset) and a dataset provided by the literature Backstabbers Knife Collection (external dataset) [35].

5. Methodology

To maintain clarity and comprehensiveness, this thesis has separated the discussion of selected approaches and their explanations into the sections Comparison of Package Managers, Effectivity of String Metrics for Typosquatting Packages, and Detection of Malicious Packages. This chapter focuses solely on the methodology for acquiring datasets and associated data preprocessing steps. Due to the scarcity of data in the field of typosquatting packages, as discussed in Key Takeaways, only a limited number of datasets are available. Therefore, new typosquatting packages must be collected. While The Backstabbers Knife Collection dataset contains numerous malicious packages, the attack targets for just over 100 typosquatting packages were known. Therefore, additional data was taken from the three most well-known companies specializing in malicious packages in software supply chain systems, Sonatype [6], Snyk[7] and Phylum[8].

5.1. Acquisation of Typosquatting Packages

The companies mentioned above provide their discoveries in various ways. Phylum and Sonatype, for example, regularly write blog entries listing their found malicious packages. On the other hand, Snyk makes their discoveries and information available in a database, some of which they also collect from other security companies. However, access to the database is currently limited to entries of a specific time period. The data entries were manually collected, grouped, and entered. In addition to Snyk's database, all blog entries at Phylum and Sonatype that offered information regarding the listed malicious packages were examined. There were 82 blog entries in total. The information provided was grouped according to two criteria: the package manager affected by the attack and whether it was a typosquatting package. The respective package was only assigned to the typosquatting package group if the target package of the attack was also specified. Otherwise, it was assigned to the Unknown Attack Vector group. Since the affected package manager was sometimes not specified, there was also an Unknown Package Manager group in addition to the other package manager groups. Even though the figures below display the distribution of npm, PyPI, Maven, and Unknown package managers, other package managers' data has also been collected. The process of data collection concluded on the 14th of December, 2022.

The Figures 19, 20, 21, one for each source of information, illustrate the number of malicious packages with unknown attack vectors and the number of typosquatting packages depending on the package manager. One can observe quite quickly that:

- Most of the malicious packages were detected for npm, packages with unknown attack vectors as well as typosquatting packages (for each source)
- For Maven, hardly any malicious packages were detected (for each source)
- For many of the packages, the affected package managers are not specified

In the case of Snyk, no information about the targets of the typosquatting packages has been provided, which is why we are only referring to malicious packages with unknown attack vectors here.

[6] https://www.sonatype.com/
[7] https://security.snyk.io/
[8] https://blog.phylum.io/

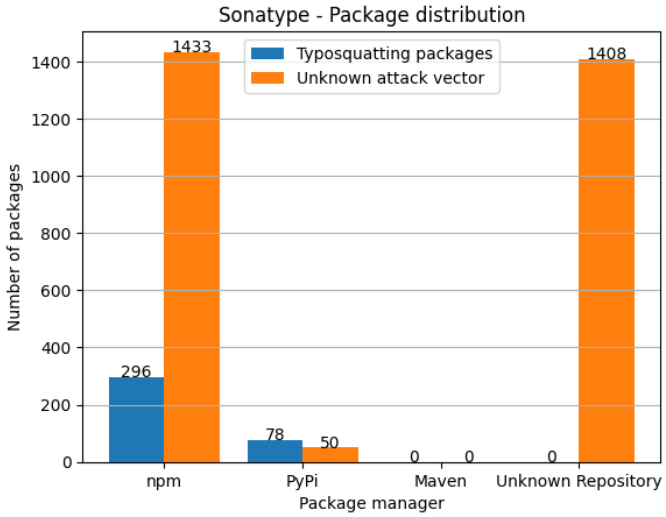

Figure 19: Distribution of data collected from Sonatype

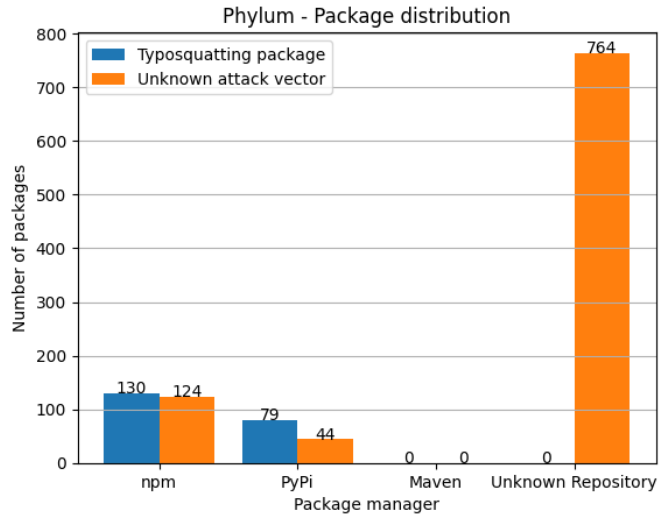

Figure 20: Distribution of data collected from Phylum

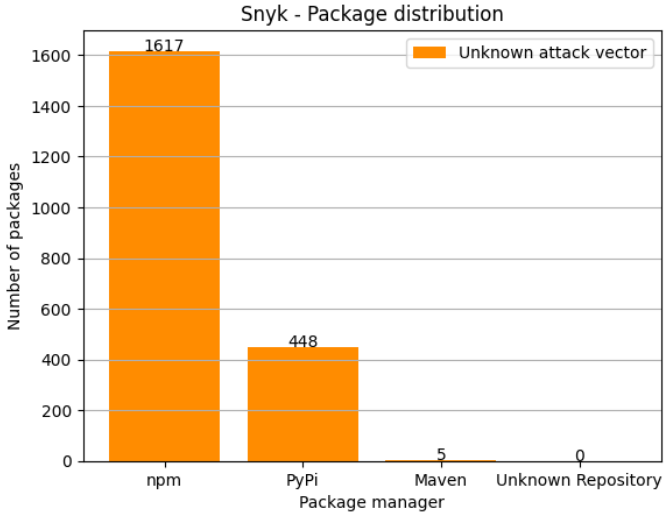

Figure 21: Distribution of data collected from Snyk

The number of packages for which the affected package manager is unknown is, at most, 2172 in the generated dataset. The upper limit of 2172 is due to the fact that both sources, i.e., Sonatype and Phylum, can list the same package. The API of the web application Libraries.io [9] was used to find the package managers where the attack was carried out. Libraries.io maintains a comprehensive copy of more than 32 package managers, which includes metadata like the package name, version, description, dependencies, and other pertinent information. It is crucial to note that while Libraries.io maintains a mirror of these package managers, it does not store the actual package files themselves, only some of the accompanying metadata. For this, all packages whose package managers are unknown were merged into a list. Then, duplicates were removed. Each library was queried via the search API. As a return value, one receives a list of possible packages to be matched, as well as the package manager on which the respective packages reside. In order to ensure accuracy, each item on the list of possible package matches was individually verified. The verification was achieved by comparing the name of the package being searched for against the names of the packages on the list. If an exact match was found, the corresponding package manager was flagged rather than aborting the search after the first match. This approach was necessary because multiple package managers could have packages with identical names. In such cases, any packages that could not be attributed to a specific package manager were marked as unknown. Otherwise, if only one package manager were flagged, it would be assigned to that flagged package manager. Packages assigned to the category Unknown Package Manager were further analyzed manually. Figure 22 shows a visualization of the process.

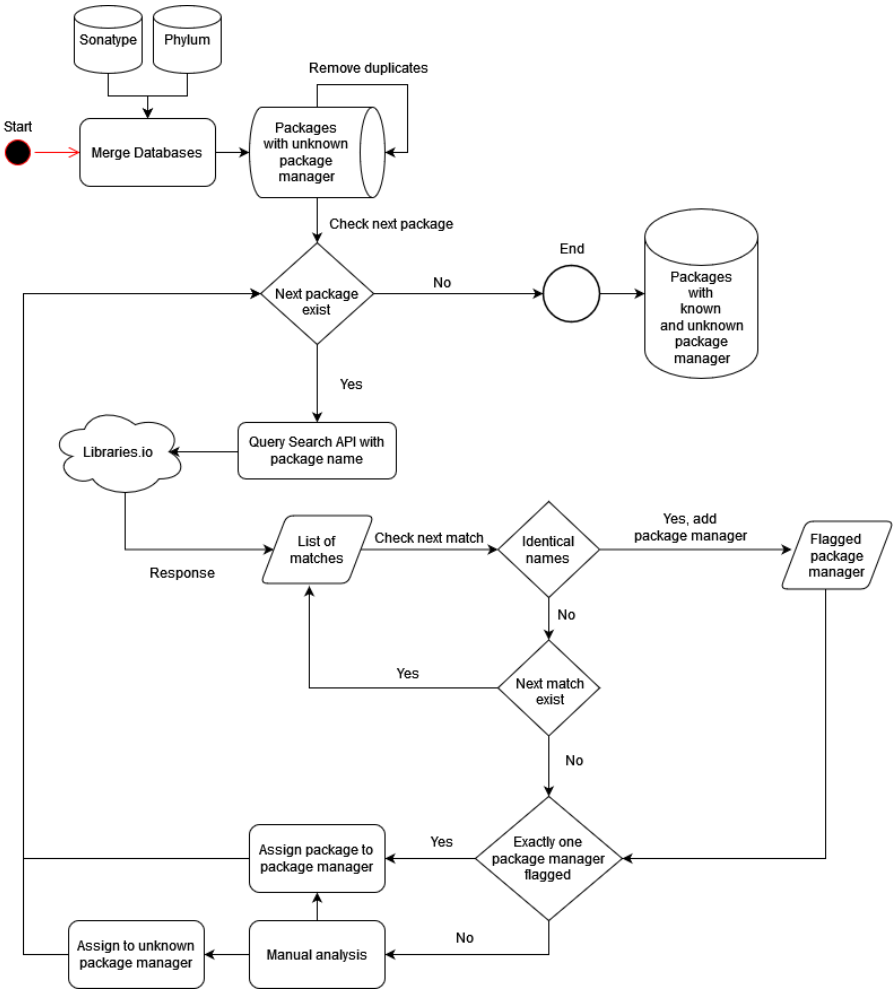

Figure 22: Package manager identification process

[9] https://libraries.io/

Of the 1408 packages extracted from Sonatype with an unknown package manager, 1153 could be assigned to npm. 231 were assigned to PyPI, and 24 remained unknown and were analyzed manually. Among the 24 manually analyzed packages, seven could be assigned to the package manager npm and four to the package manager PyPI, resulting in only 13 packages for Sonatype that could not be assigned to any package manager. For the 764 packages from Phylum with an unknown package manager, 710 could be assigned to npm and 48 to PyPI. Of the six which remained unknown, three could be assigned to the package manager PyPI through manual analysis. Figures 23 and 24 further visualize the distribution of the identification process.

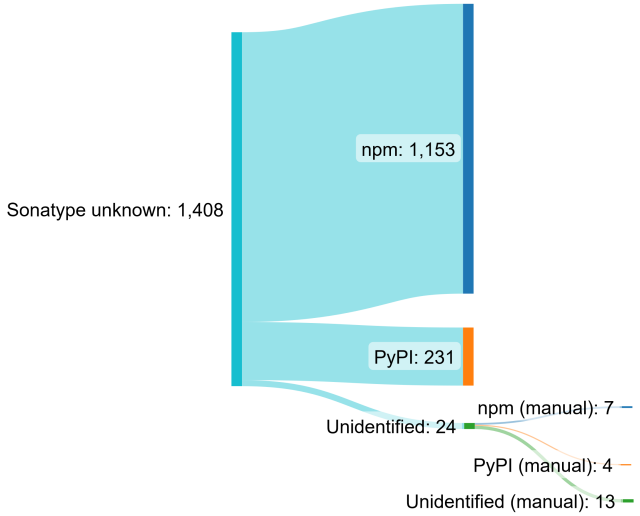

Figure 23: Sankey diagram of packages with unidentified package manager from Sonatype

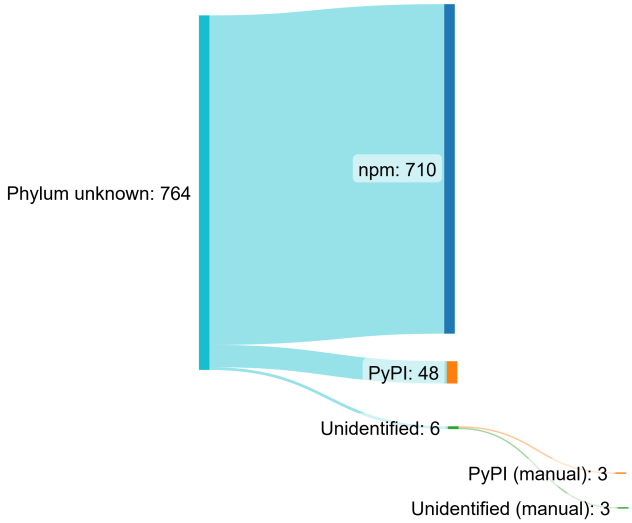

Figure 24: Sankey diagram of packages with unidentified package manager from Phylum

There could be several reasons why further identification of the attack package manager could not be achieved. One example would be that the package's name was incorrect. Another reason may be that the package was removed before Libraries.io could produce a copy. Thus packages that could not be assigned to any package manager are not further considered in this thesis. The adjusted final distribution from the different sources is shown in Figures 25, 26.

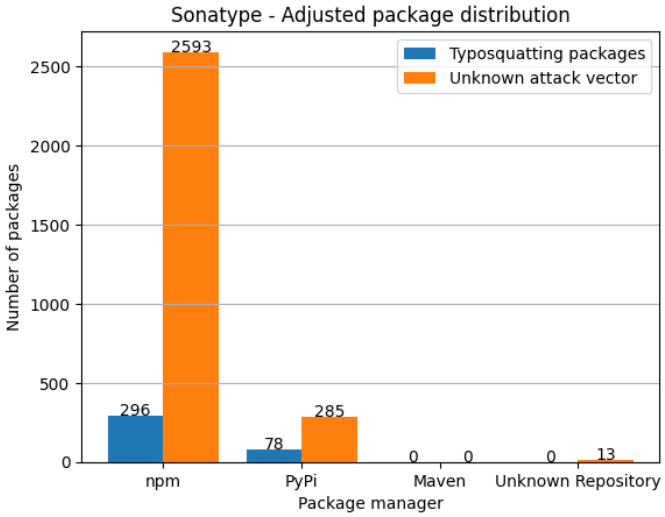

Figure 25: Adjusted Sonatype package distribution

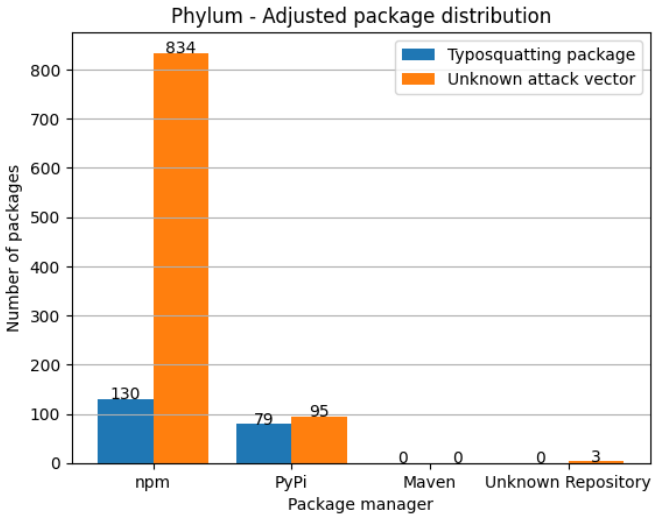

Figure 26: Adjusted Phylum package distribution

5.2. Data Cleansing and Preprocessing

For further processing, the datasets from all three sources were combined. That means all typosquatting packages from the different sources were merged into one list, and all packages for which the attack vector is unknown were merged into a second list. Subsequently, all duplicates in both lists were removed. If the same packages were named with different attack targets in the typosquatting packages, the attack target with the highest downloads was taken. Example: Two legitimate packages named *jsonb* and *json* exist. The typosquatting package *jsonm* is listed at Sonatype with the legitimate package jsonb as the attack target. At the same time, Phylum names the typosquatting with the legitimate package json as an attack target. In such cases, the legitimate package with the larger download count would always be taken as the attack target. In addition, the list of malicious packages, for which the attack vector is unknown, was checked to verify that they were not listed in the list of typosquatting packages either. In this case, they were removed from the list of packages with unknown attack vectors. After the first step of preprocessing was done, for each typosquatting package and its attack targets, a manual check was performed to ensure that each of the listed packages had been correctly reported; in other words, whether the respective typosquatting package existed and its attack target existed. The reason for this is that when listing the packages in the Sonatype or Phylum

blog posts, additional typos may have been introduced. Therefore the following processes have been applied to the datasets:

1. Merge the datasets from different sources into one dataset, respective to their categories
2. Remove duplicates for each dataset
3. Adjust typosquatting packages with the same package name but different targets
4. Check if typosquatting packages are listed in both categories
5. Check if every typosquatting package and its targets existed

Additionally, two of the five malicious packages mentioned in Snyk for Maven seem to be false positives or preventional flags. The versions that were identified as malicious were never released on Maven, only on npm. The names of the flagged artifacts are *org.webjars.npm:coa* and *org.webjars.npm:rc*. Malicious packages for the package manager RubyGems, Crates, and Packagist were mostly provided by Snyk and Sonatype, while no malicious packages were found for NuGet and Go. No special processing was required for the other package managers due to the limited datasets. In the case of RubyGems, while the blog entry mentioned that the 725 packages were typosquatting packages since no attack targets were mentioned for the packages listed, the definition for packages in the typosquatting packages category defined in section 5.1 is used here and thus categorizes the 725 packages as packages with unknown attack vectors. The execution of the processes in the mentioned order leads to the following final result for all package managers:

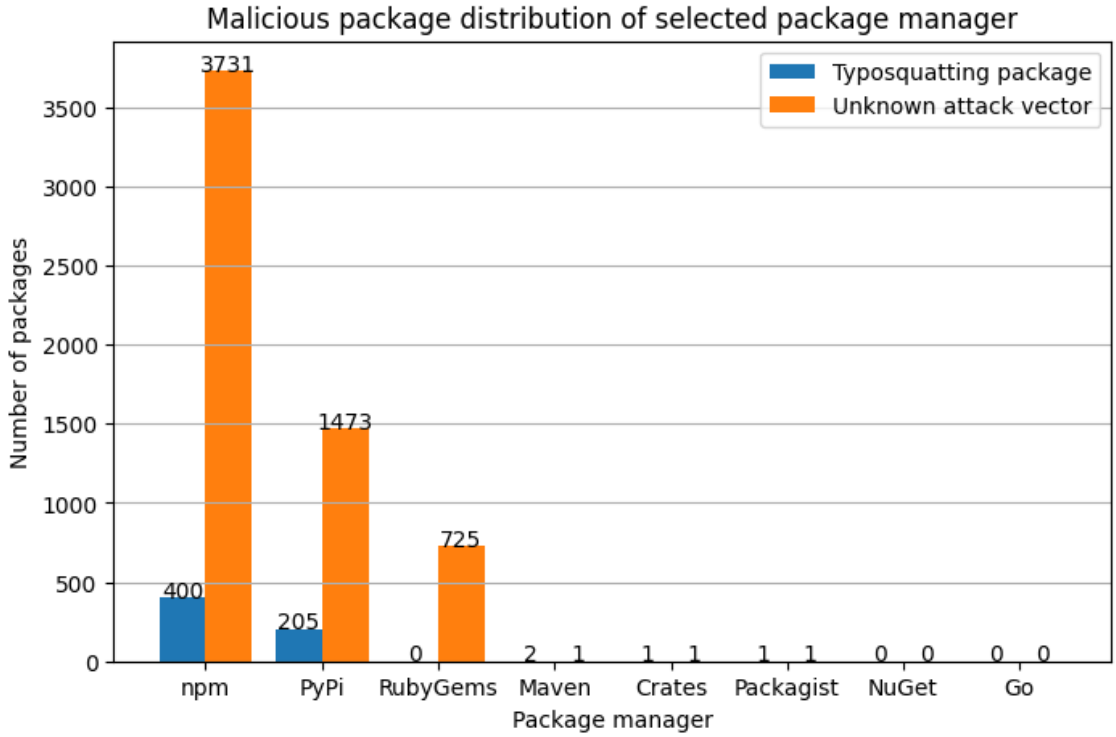

Figure 27: Overall package distribution

5.3. Acquisation of Metadata & Source Code

As indicated in the figure, npm has the most typosquatting packages. Therefore further in-depth investigation will be focused on them. Currently, only the names of the typosquatting packages and the names of their targets were retrieved. The source code and the metadata of the packages have yet to be obtained. If a malicious package has been reported to npm, only the source code is typically removed, but the package can still be installed using the command line command. However, only a package without source code and thus without functionality is downloaded. Sometimes, libraries are completely

removed, leaving hardly any metadata available. The metadata was retrieved from the API of npm itself when available. Thus, for the 400 npm typosquatting packages, the following metadata has been collected:

- Date of release of the package
- Date of removal of the package
- The last version number
- The number of versions
- Number of downloads since the release until 01/30/2023
- Whether the package was removed completely

Among the 400 typosquatting packages, the metadata could be retrieved for only 266 of the packages. The metadata for the remaining packages are not retrievable since the packages had been removed entirely. In addition, the source code for the 400 typosquatting packages must now be acquired. This a relatively tricky task since npm attempts to remove all traces of such packages. The only way to gather the source code of such malicious packages is via other npm mirrors, replication from blog entries, or GitHub issues; Some of the packages' source codes were also present in the Backstabber's Knife Collection dataset. In the end, the source code of 396 out of 400 typosquatting packages could be tracked down. The forthcoming analysis is divided into three phases. The first analysis focuses on the correlation between package manager properties and their vulnerability in terms of malicious package uploads and typosquatting packages. The second analysis deals exclusively with the package name and its impact on the effectiveness of typosquatting packages. For this phase, 266 datasets with metadata are available. The third phase deals with the detection of malicious intent in the source code. For this, 390 datasets with source codes are available. For the targets of the typosquatting packages, the metadata and the source code have also been retrieved and will serve as a dataset for benign packages in this thesis.

6. Mitigation and Detection of Typosquatting Packages

As mentioned in the section Outline, the following chapter is the central part of this thesis. The first subchapter examines how the vulnerability of package managers is related to their properties in order to derive possible conclusions. However, at this point, it will be noticed that this is not entirely straightforward and that the realization of the proposed properties is not practicable. Therefore we will focus on an alternative approach, the detection of typosquatting packages. For this, the subsequent two subchapters are dedicated to this topic. The first subsection examines the effectiveness of different string metrics in finding typosquatting candidates. The second subchapter deals with the follow-up method for automatically classifying typosquatting candidates. Here, typosquatting candidates will be examined for malicious code to classify them as typosquatting packages.

6.1. Comparison of Package Managers

The following eight package managers were chosen: npm, PyPI, Maven, RubyGems, Packagist, NuGet, Go, and Cargo. The reasoning behind this choice is that according to Libraries.io (as of 19.02.2023), these are the eight largest package managers in software development. The largest package manager is npm, with 2.3 million packages, followed by PyPI, Go, NuGet, Packagist, RubyGems, and Cargo, listed in the order of available packages. An exact breakdown of the number of packages (as of 19.02.2023) is given in Table 6. If possible, the information was taken from the package managers themselves. Otherwise, Libraries.io was taken as a source.

Package manager	**Number of packages**	**Number of malicious packages**	**Ratio**
npm	2,306,551	4131	0.179%
Maven	524,731	3	0.00057%
PyPI	443,554	1678	0.378%
Go[10]	439,577	0	0%
Packagist	363,789	2	0.00055%
NuGet	342,855	0	0%
RubyGems	175,307	725	0.413%
Cargo	105,297	2	0.0019%

Table 6: Package manager statistics

A distinction is made between dependent and independent variables. In this thesis, it is assumed that the dependent variable, for example, the number of malicious packages, is dependent on the independent variables, the properties of a package manager. The properties selected to compare each package manager were partly obtained from the paper [43], adopting the points that contribute to the vulnerability of a package manager in terms of malicious package injection. For instance, if multiple malicious packages were published in a short period of time by multiple user accounts, it is plausible that the attacker used automation to create multiple accounts to perform the attack. For this, CAPTCHAs may provide a resolution. It is, therefore, necessary to determine the correlation between the number of malicious packages and the properties of the package managers. In addition to the statistics given in Table 6, further properties for each package manager were collected

[10]Retrieved from libraries.io

to the best of our knowledge. The various libraries were examined for the following characteristics:

- Execution of scripts during the installation (Install Hooks)
- CAPTCHA (CAPTCHA)
- Mandatory specification of a repository in which the source code resides (Mandatory VCS)
- Installation of packages through command line interface (Install through CLI)
- Verification of uploaded packages both manually and automatically (Code Scan)
- Namespace system (Namespace system)
- Effort for publishing new packages (Complexity)
- Number of packages (Number of Packages)
- Popularity for the supported programming language (Popularity Rank) [11]

In more detail, the Install Hooks describe the ability to execute arbitrary scripts and code during installation. During the observations, the following instances for this property could be observed: "native", natively supported, "workaround" if a workaround is necessary, "partly" (e.g., through the build process), and "no" for no support at all. The CAPTCHA property describes whether a CAPTCHA must be solved to successfully finish the registration process of a new account, indicated by "no" if not needed or "yes" if needed. For this, indirect CAPTCHA integration has also been considered. For instance, some package managers require a GitHub account to publish a new package. For GitHub, a CAPTCHA has to be solved to create an account. The property "Mandatory (VCS)" is the necessity to upload the package onto a version control system (VCS), e.g., GitHub, and linking these together, the package manager will then use the provided repository as their source for the package. Package managers here can be categorized into two types. The first type uses the mentioned approach. The second one, implemented by most package managers, is that the packages reside directly in the package manager's repository, and a link to a VCS repository can be specified optionally. The source for the package is thus only uploaded to the package manager's repository. Even if the VCS repository is specified, the package manager will still use the package which resides in the package manager's repository as the source. Type one is indicated by the value "yes" and type two with "no". The property "Code Scan" describes whether a code scan is performed manually or automatically. If a code scan is performed, this is indicated by "yes". Otherwise, the value is "no". The namespace system describes the characteristics of the naming system used for naming packages. They are distinguished into three categories: "name" if only a package name is needed, "group (optional) and name", if the specification of a group is optional and a name is necessary, and "group and name", if a group identifier is mandatory. A possible package name for the category "group and name" could be <groupId>.<package name> in case of Maven or <groupId>/<package name> in case of Packagist. The property "Number of Packages" is an integer value describing the number of published packages in a package manager. The property "Popularity" describes the percentage of people who voted for the supported programming language of the package manager. For instance, if respondents voted 48.07% in favor of the programming language Python, then the popularity value of PyPI is 48.07%. One has to mention that respondents were allowed to vote for multiple programming languages. Thus 48.07% of the respondents voted for Python. The following procedure has been performed to determine the complexity of publishing a package. Firstly, the upload process of a new library was recorded in steps across all package managers. Subsequently, the different steps were compared with one another and examined for commonalities. After comparison, all package managers could be classified into three groups. The first group, corresponding to the complexity "simple", can be described by the following process.

1. Creation of a metadata file

[11] https://survey.stackoverflow.co/2022/#technology-most-popular-technologies

2. Execute command to start the initialization of a package
3. Perform authentication
4. Upload the package with a command

An additional intermediate step is necessary for the complexity "advanced ", where the respective package has to be uploaded to a repository or version control system. For the complexity "complex ", permission for uploading a library has to be granted in addition to the steps of the complexity "simple" by the authorities. The information for the properties of the various package manager was retrieved either from documentation, contributions, or determined by practical tests.

After the creation of this dataset, normalization was performed in two ways. The first type of normalization converts absolute values into a relational order. Instead of using, for example, the raw numbers of malicious packages detected per package manager, a mapping was performed to rank the package managers according to the number of malicious packages contained within said managers. Such a ranking was performed for the popularity, the number of all packages, and the number of malicious packages. The background to this is the relative ranking of the various package managers according to their vulnerabilities and the determination of a possible correlation. Additionally, a new property has been added, "Ratio rank", denoting the ratio of malicious packages and all packages in a package manager in a relational order. The ranking started at position 0 for being the least vulnerable. Therefore the higher a value is, the more vulnerable a package manager is, indicated by the specified dependent variables (Malicious package rank or Ratio rank). If package managers were assigned the same values before the relational normalization, they were given the same rank.

The second type of normalization is done on the properties of the package managers. Here, an estimation was performed for each instance of a property. For example, to normalize the property namespace system, all instances of this property are first mapped to a whole number. The higher the number, the less practical the realization of such an instance is for a package manager's security. For the property namespace system, there are the following instances: "name", "group (optional) and name", and "group and name". In total, there are three instances. If one positions themselves as described above, the value of an instance decreases as it becomes more advantageous to the system. Therefore the rank would start from zero to two for the following instances: "group and name", "group (optional) and name" and "name". This order suggests that "group and name" will benefit a system's security the most, whereas "name" will have the least positive impact. This procedure has been performed on all properties. In the case of the property "Install Hooks" the instances were: "native", "workaround", "partly" and "no". For this, the instances "workaround" and "partly" received the same value of one.

After the normalization, a correlation matrix will be calculated using the Spearman correlation. This is because the created dataset does not fulfill certain requirements for the Pearson correlation. In order to use the Pearson correlation, the data must be normally distributed and continuous [53]. Both requirements are not satisfied by the provided dataset. For the Spearman coefficient, however, this is not necessary. In fact, the Spearman correlation works especially well on nonnumerical data as long as they can be classified through, for instance, a relational order [54].

6.1.1. Results

The values provided in Table 7 result from the data acquisition mentioned in the section above. This data frame contains all the package managers with the properties and the dependent variables. After performing the normalization described in chapter 6.1, the matrix in Table 8 is created. A correlation matrix is performed using the Spearman correlation to determine a correlation in this normalized table. The resulting matrix can be seen in Figure 28.

	Package Manager	Malicious Packages	Namespace system	Install Hooks	Captcha	Mandatory VCS	Install through CLI	Code Scan	Popularity	Publishing complexity	Number of Packages
1	npm	4132	group (optional) and name	native	no	no	yes	no	65.36%	simple	2306551
2	PyPI	1677	name	native	no	no	yes	no	48.07%	simple	443554
3	RubyGems	804	name	workaround	no	no	yes	no	6.05%	simple	175307
0	Maven Central	3	group and name	no	yes	no	possible, but unusual	no	33.27%	complex	524731
7	Crates.io	2	name	no	yes	no	yes	no	9.23%	simple	105297
4	Packagist	1	group and name	no	yes	yes	yes	no	20.87%	advanced	363789
5	NuGet	0	group (optional) and name	partly	yes	no	yes	yes	27.98%	simple	342855
6	Go	0	group and name	no	yes	yes	yes	no	11.15%	advanced	439577

Table 7: Comparison of package managers

	Package Manager	Malicious package rank	Namespace system	Install Hooks	Captcha	Mandatory VCS	Install through CLI	Code Scan	Popularity Rank	Publishing complexity	Package number rank
0	Maven Central	3	0	0	0	1	0	1	6	0	6
1	npm	6	1	2	1	1	1	1	8	2	7
2	PyPI	5	2	2	1	1	1	1	7	2	5
3	RubyGems	4	2	1	1	1	1	1	1	2	1
4	Packagist	1	0	0	0	0	1	1	4	1	3
5	NuGet	0	1	1	0	1	1	0	5	2	2
6	Go	0	0	0	0	0	1	1	3	1	4
7	Crates.io	2	2	0	0	1	1	1	2	2	0

Table 8: Normalized package managers matrix

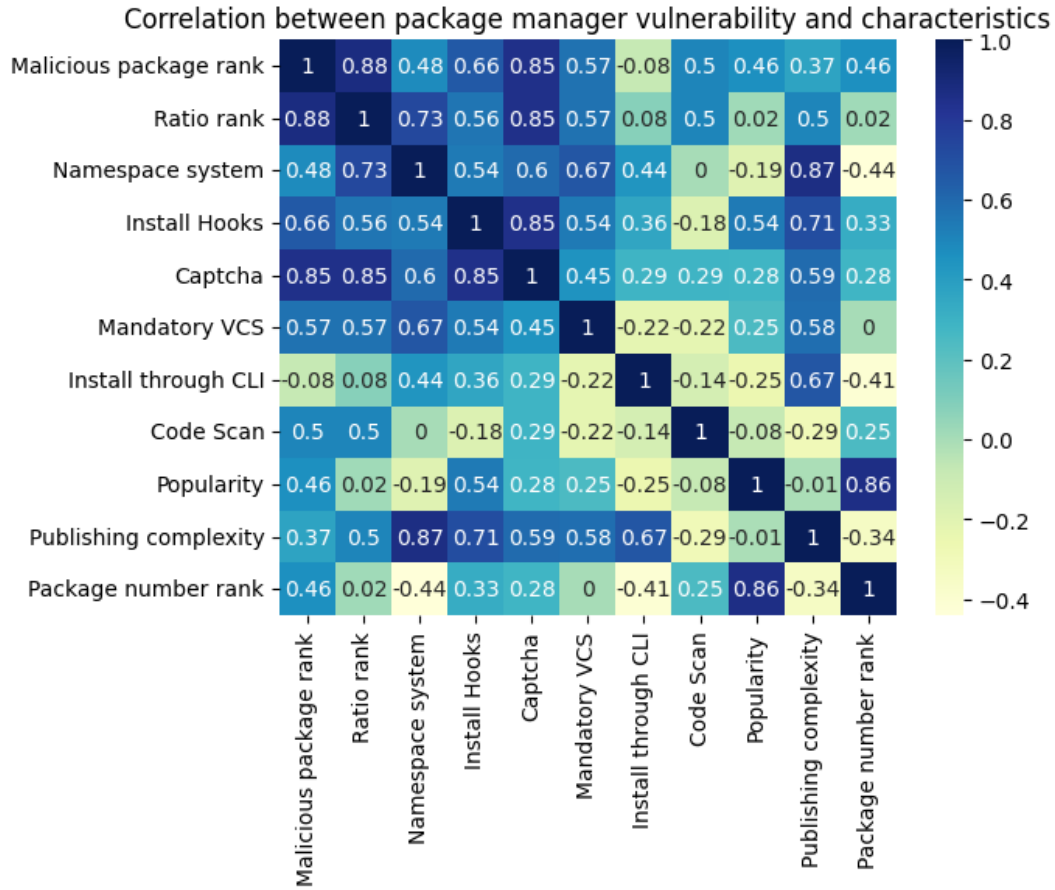

Figure 28: Spearman correlation matrix

6.1.2. Discussion & Interpretation

Figure 28 shows an apparent color distinction. The Spearman correlation was used to calculate the correlation matrix there. It indicates how much two properties correlate with each other. The value -1 indicates an opposite correlation, 0 for no correlation, and 1 for a perfect correlation [54]. For example, if properties X and Y correlate with a positive correlation coefficient larger than 0. Thus Y becomes larger when X becomes larger, and vice versa. The larger the value, the stronger the correlation. Since only the correlations between the dependent variables (Malicious package rank and Ratio rank) and the independent variables (package manager's properties) are interesting for this thesis, only the values of the first two columns are to be examined. Alternatively, one could only observe the first two rows due to the symmetry.

According to the Spearman correlation matrix, the most significant positive correlation for both dependent variables is caused by the property CAPTCHA. This indicates that a package manager that does not use a CAPTCHA has more malicious packages. A plausible hypothesis might be that the inclusion of CAPTCHAs can reduce the number of automated attacks. For example, an attacker can use scripts more efficiently to carry out automated attacks against a package manager that has not included CAPTCHAs. The datasets or malicious packages collected for this thesis indicate a high level of automation. Several hundred typosquatting packages were uploaded to the npm ecosystem within minutes to seconds. Whether a new user account was created each time in the process can no longer be inferred from the metadata, as the authorities at npm overwrote it. Assuming the attacker uploaded all malicious packages from the same user account, then this is also a flaw that should be investigated. Further examination has also shown that there are users on npm who uploaded thousands of packages on the same day. A total of 33485 non-functional packages were uploaded by a particular user[12], with each package being one Levenshtein distance apart from the other. This polluted nature of the npm repository may explain why previous literature received a lot of false positives. The automation is amplified by providing a command line interface (CLI). Even though every package manager supports the installation or addition of libraries through the CLI, only some package managers seem vulnerable to typosquatting packages. Herefore, more than the existence of support for CLIs alone is needed to cause an increased number of attacks. This observation is also reflected in the correlation matrix with a value of nearly 0, indicating no correlation.

The second largest correlation is caused by Install Hooks when looking at the dependent variable malicious package rank. Furthermore, it is the third-largest correlation when looking at the ratio rank. Here the cause is relatively straightforward. If a package manager supports the execution of arbitrary scripts, attackers are provided with an easy way to increase the effectiveness of the attack. If an attacker initiates an attack via the Install Hook, the user usually cannot prevent it and does not even know that he has been the victim of an attack. The only way to avoid this attack in npm is to set a flag in the CLI. For PyPI, there is no way to prevent the execution of a script except by not installing the library in the first place. If, on the other hand, the attacker relies on his malicious code using the integration or import into the victim's software as an entry point, two conditions usually have to be satisfied. Firstly, in some instances, the victim has to make a second typing error when importing the library, and secondly, the victim has to run the program after the library has been imported. The probability of a second typo alone, which must be identical to the previous one, is already relatively low. Some integrated development environments (IDE) may reduce the need for a second typo due to auto-completion. Nevertheless, the chance of detecting that a typo was made is increased.

The third largest correlation by both dependent variables is caused by the obligatory upload to a VCS repository. The repository in question is used as the source for the package involved. One possible justification for such a correlation is the additional overhead. Thus,

[12] https://web.archive.org/web/20230220023533/https://www.npmjs.com/ infinitebrahmanuniverse?activeTab=packages

the attacker must create an account for both platforms, that of the package manager and that of a version control system such as GitHub, whereby most platforms that offer such services are additionally protected against automatic attacks by CAPTCHAs. Another possible explanation is the disclosure of the source code, which increases the risk of being detected.

The second-largest correlation in terms of ratio rank is the namespace system, with a correlation value of 0.78. But "only" with a correlation value of 0.48 with the malicious package rank. The most straightforward answer is that the attack vectors, typosquatting, and dependency confusion benefit from a weak naming system. Both attack vectors use the naming systems as a point of entry. Although typos can still occur in package managers with more complex naming systems, it is plausible that these occur at a reduced rate. One possible reason is that group identifiers lengthen package names, leading to copying the library name instead of typing it. In the case of Maven, one must also prove that one owns an associated group domain. Dependency confusion benefits from this because companies, for example, use private repositories to manage their internal libraries. Attackers may now upload packages that have the identical package name as the packages in the private repository. If, in addition, the malicious package has a higher version and the package manager client has been misconfigured, the package manager will look in the public repository to see if a library with the same name has a higher version there. If this is the case, it will be downloaded instead. In this case, dependency confusion attacks benefit more from simple naming systems than typosquatting attacks since the typosquatting can be moved from the package name to the group name instead.

The correlation coefficients for popularity, the number of packages, code scan, and the namespace system are in the same order of magnitude when correlated to the malicious package rank. They positively affect package managers' security, with a correlation coefficient of about 0.5. However, the correlation between the ratio rank and the popularity and package number rank is rather close to zero. One possible explanation is that the ratio removed the influence of the number of packages as well as the influence of popularity. For instance, if a package manager contains 100 packages, of which one is malicious, then the ratio is 1%. Another package manager may have 1000 packages, of which ten may be malicious, also resulting in a ratio of 1%. However, if ranked by the number of malicious packages, the latter will have a higher correlation in popularity because the popularity correlates with the number of packages and, therefore, with the number of malicious packages. With the ratio rank, this has been reduced.

While the Spearman correlation gives approximate correlation directions, there may be flaws in the accuracy of the correlation. An example of such a deficiency would be that all properties, as well as individual elements used for the calculation, are weighted equally. For example, the properties of Maven, which has five times as many packages as Crates but still has nearly the same number of malicious packages, are weighted equally. Crates may have fewer malicious packages, not due to the package manager's protective properties but rather its lower popularity. One possibility to counteract this would be, instead of using the absolute number or rank of the malicious packages, the use of a ratio between malicious packages and the number of all packages might be better. However, the popularity would not be taken directly into the weighting, but only indirectly, since the popularity of the package manager can be assumed to correlate with the number of packages, which indicates that the Spearman correlation has another disadvantage, and that is transitive considerations. They can only be considered if they are implicitly specified as a property. A better approach could be a complex multivariant analysis.

Besides the limited possibilities of the Spearman correlation, another weak point is the dataset. Knowing exactly how many malicious packages each package manager has would be ideal. Since the number of malicious packages is only a sample, which comes from the sources mentioned in chapter 5, it is plausible that the samples could have a certain bias. Even if this bias becomes smaller by taking the samples from multiple sources, this assumption of a small bias is nullified if all sources have the bias. Herefore,

while the correlation matrix provides general directions that make a package manager less vulnerable, how strongly these measures affect security is still questionable. Another issue with the data is that the sample size may be too small, and the calculated correlation is only coincidental. However, one could logically justify how certain properties might contribute to increased safety. Therefore, the correlation matrix specifies directions that ensure that a package manager is less vulnerable, but how strongly these measures affect security is still questionable.

If the findings were true, one would run into another problem. The implementation of the CAPTCHA in the package manager is simple. The realization of the other characteristics takes much work. Since one must change all already existing packages under certain circumstances, for example, if the Install Hooks were to be removed, some of the packages and the software that integrates these packages would no longer work since these packages depend on the Install Hooks. The same applies to the namespace system. If one wanted to change this, all packages would have to specify a group id afterward. Therefore every software that uses such packages must be updated both at the import and the metadata level. Using a repository as a source for packages would also be invasive since not all packages on npm have a repository, and furthermore, one would have to change the structure of, for instance, npm entirely as a result. In addition, when implementing the features considered in this thesis, one must also perform a trade-off analysis between usability and security, which is a discipline of its own [55]. Thus the least invasive intervention besides a CAPTCHA would be a code scan. According to theoretical computer science, the classification of source code into malicious or benign is an undecidable problem. However, for undecidable problems, approximations can be made using, for example, statistical analysis, and a decision can be made based on this. This is why the following analysis focuses on detecting malicious packages, particularly a subset of malicious packages, typosquatting packages.

6.2. Effectivity of String Metrics for Typosquatting Packages

For this examination, 266 of the 400 typosquatting packages were used due to the availability of their metadata. Firstly, the list was cleansed of libraries that are not considered typosquatting packages. These packages were considered combosquatting packages or categorized as confusion attacks according to the definition provided in section 3.2. The following packages were therefore removed:

- ca-bucky-client with the target bucky
- noblox.js-proxy with the target noblox.js
- stringjs_lib with the target string
- twilio-npm with the target twilio
- discord.dll with the target discord.js
- support-colors, colors-update, colors-support, colors_express, sync-colors, colors-3.0, colors-helper, colors-help, colors2.0 with the target colors

As such, out of the total dataset used in this chapter, which consists of 266 typosquatting packages, confusion attacks or combosquatting packages account for only 5.26%. Therefore, the remaining 252 typosquatting packages will be analyzed in detail in this chapter. To conduct a statistical investigation on the effectiveness of typosquatting packages, it is necessary to define the criteria that distinguish an effective package from a non-effective one. In this regard, the number of downloads serves as an appropriate metric for measuring the effectiveness of typosquatting packages. Since each package was released at different times and naturally had different public lifespans for a download to occur, this value must be normalized accordingly. The normalization was done by taking the ratio between the total downloads and the days since the release. The end date was taken as 30th January 2023. This normalized value, i.e., the average number of downloads per day, is now considered the dependent variable. The following Figure 29 presents the normalized download rate of the respective packages, whereby the datasets were additionally sorted by the average download of each package.

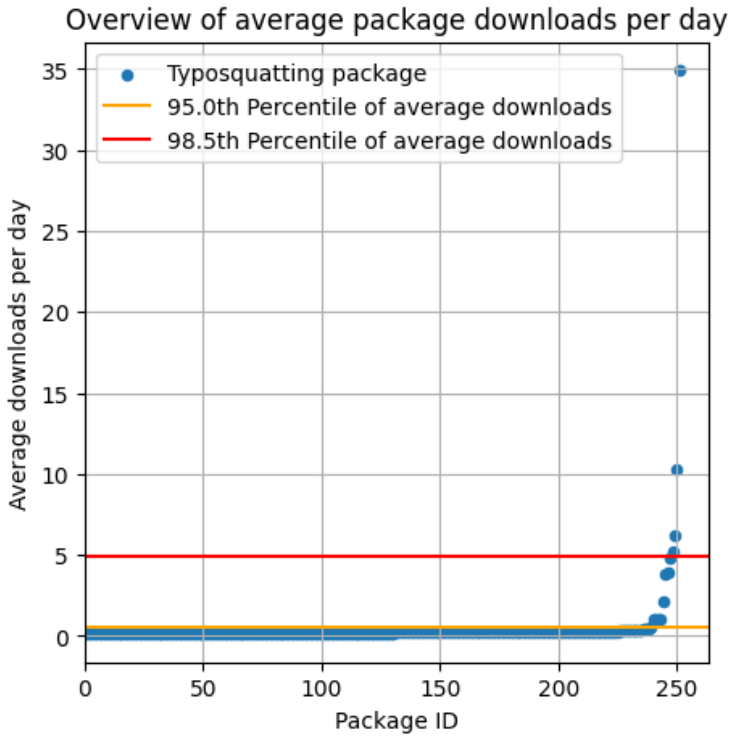

Figure 29: Average downloads per day of typosquatting packages

Figure 29 emphasizes that only a handful of typosquatting packages seem particularly effective, while the majority of the typosquatting packages are not. To determine the property affecting the effectiveness of typosquatting packages, four string metrics Levenshtein, Damerau-Levenshtein, Jaro-Winkler, and the Gestalt-Pattern-Matching algorithms were applied using the Python libraries *jellyfish* and *difflib*. The Modified-Damerau-Levenshtein distance, the fifth metric, has been implemented using Python[13]. This algorithm is based on the Damerau-Levenshtein algorithm. Moreover, this algorithm incorporates a specific property from the literature [47] that takes into account the adjacency of letters on the English keyboard. It achieves this by iteratively examining whether the compared letters are adjacent to each other on the keyboard and setting the corresponding costs that contribute to the edit distance computation. For each iteration, the cost is reduced to 0.5 if the letters are adjacent and increased to 2 if they are not. The selected values of 0.5 and 2 are based on the rationale that plausible typos incur a lower cost, while implausible values incur a higher cost. Specifically, the cost for a plausible typo is halved, resulting in a value of 0.5, whereas the cost for an implausible value is doubled, resulting in a value of 2. These costs are then utilized in the computation of the edit distance. The modified version introduced the following rules:

- The cost for transposition is set to 0.5 e.g., raect ⟶ react
- The cost for substitution of adjacent characters is set to 0.5 e.g., react ⟶ resct (the key "s" is next to "a" on the keyboard)
- The cost for inserted adjacent characters is set to 0.5 e.g., react ⟶ reacvt (the key "v" is next to "c" on the keyboard)
- The cost for omitted characters is set to 0.5 e.g., react ⟶ rect; the cost is set to 1 if the first character is omitted.
- Normalization of package names which include the suffix ".js" have a cost of 0.5 e.g., react ⟶ react.js
- Everything else costs either 0 if the compared letters are equal or 2 otherwise.

There are three main reasons why the Damerau-Levenshtein has been taken as the basis. The first reason is the support for manually adjusting the cost by simply changing one variable. This is only possible for the Jaro-Winkler and the Gestalt-Pattern through drastic changes in the equations or algorithm. The second reason is that the Damerau-Levenshtein distance supports the transposition of adjacent characters. In the case of the Levenshtein distance, this feature is not supported directly but indirectly through two operations. However, introducing an accidental transposition is caused by a timing issue of a few milliseconds. For instance, the difference between "react" and "raect" could be the results of the key "a" being typed a few milliseconds faster by accident than the key "e", and therefore describing the transposition using only one operation step might be more fitting. The last reason is that the Damerau-Levenshtein natively already performs checks on the equality of letters. In this case, the check can be easily expanded by a neighbor function that checks whether two keys on the keyboards are neighbors. If that is the case, the cost will be adjusted accordingly. Therefore the Damerau-Levenshtein distance has optimal requirements to be extended using the approaches from [12].

The normalized number of downloads defines the effectiveness of typosquatting packages. In contrast, the independent variable is represented by similarity or distance to the target package. The scatter plots in Figure 30 illustrate the downloads with respect to the distances or similarity between the typosquatting packages and the attack targets, with each plot representing a specific metric. For example, the point in the plot titled "Levenshtein vs. Average downloads per day", assigned the X-value of 1 and the Y-value of 35, indicates that the package has an average download count of 35 per day and a Levenshtein distance to its attack target of 1. Each point in this plot is a typosquatting package. In the graph of the Levenshtein and the Damerau-Levenshtein distance, it becomes evident that all the typosquatting packages having a distance higher than 2 to their attack targets have

[13]The source code is provided in the appendix

a download rate close to zero. For the scatter plot of the Jaro-Winkler distance, it can be observed that most of the typosquatting packages with a significantly higher download rate are found starting with a similarity value of 0.95. Though there are some outliers before the value of 0.95, too, and packages whose download rate is less significant but not equal to 0. Particularly noticeable are the typosquatting packages, which, according to the Jaro-Winkler distance, have no similarities at all and thus have a similarity value of 0. This was the case for the typosquatting packages *5rn* and *9mz* with target packages *rn* and *mz*. The Gestalt-Pattern, on the other hand, has a similar but slightly larger distribution. Still, in contrast to the Jaro-Winkler similarity, no point with a similarity value of 0 exists. The Modified-Damerau-Levenshtein distance has a similar distribution to the original Damerau-Levenshtein distance, as well as the Levenshtein distance, except that it has a higher granularity. In addition, some of the packages that were previously assigned to distance 2 in the Levenshtein and Damerau-Levenshtein distance are now assigned to a distance of 0.5. The scattering of packages with higher download rates has thus been slightly reduced, but the scattering of typosquatting packages with lower download rates has been increased.

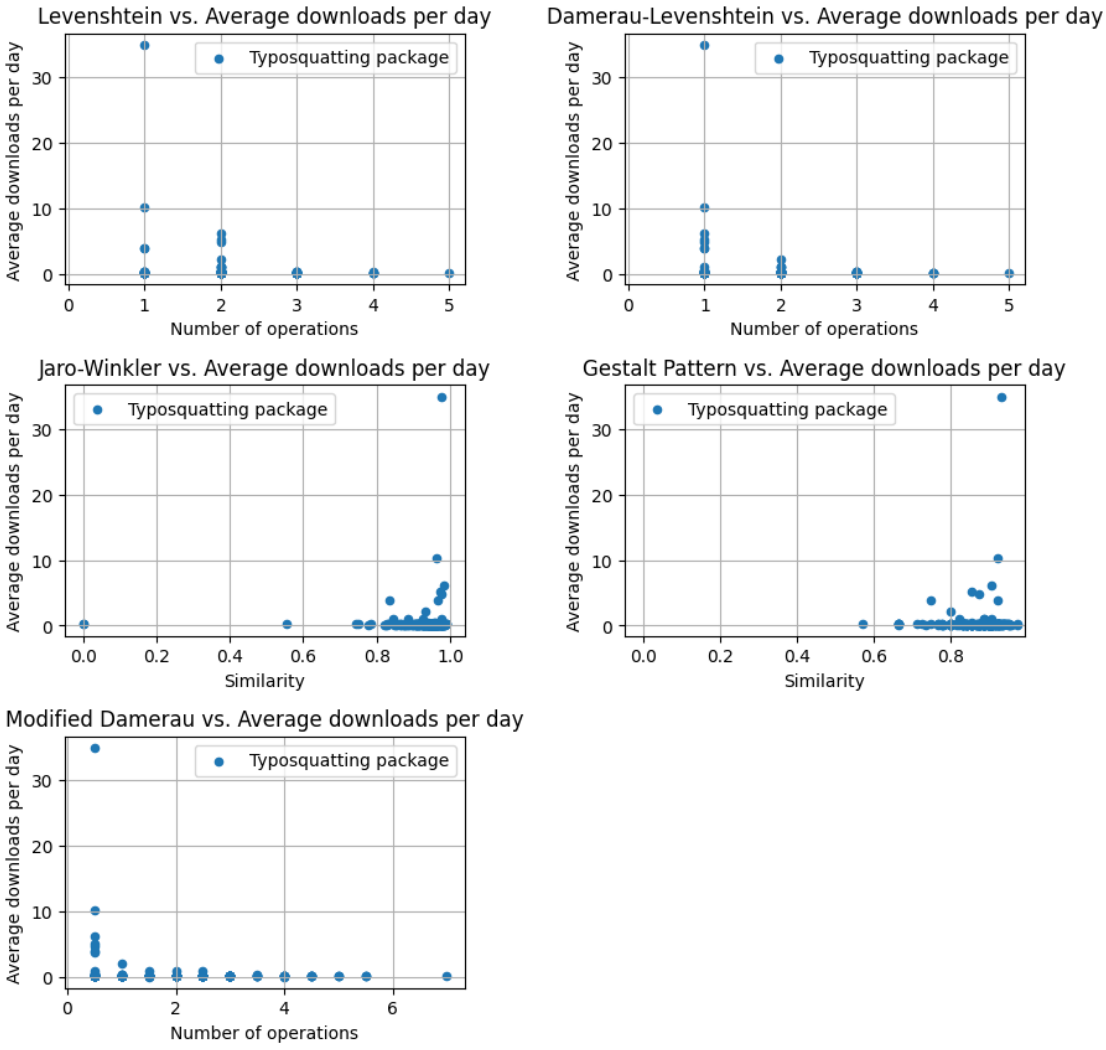

Figure 30: Downloads per day based on similarity

In this regard, Figure 30 provides some initial indications of how the thresholds for the various metrics could be chosen. According to the table in Figure 30, it is advisable to choose a Levenshtein and Damerau-Levenshtein distance of 2 since these cover all typosquatting packages that are downloaded at least once a day. For the Jaro-Winkler distance, there are three possible approaches for the selection; either one selects the threshold value to cover all typosquatting packages that are not considered outliers. In the second

option, one selects the threshold starting from the point where a typosquatting package is downloaded significantly more often than less effective typosquatting packages. With the first approach, the threshold would be about 0.75, while the second approach results in a threshold of about 0.82. Using the same approach for the Gestalt pattern algorithm, a threshold of about 0.7 would be used for the first approach, while a threshold of 0.75 would be chosen for the second approach. For the Modified-Damerau-Levenshtein distance, either a distance of 1 applies if all significant packages are to be included or 0.5 if the typosquatting package, which has a download rate of 2 at a distance of 1, is to be considered an outlier. The third possible approach is explained in the following paragraph by utilizing the Figure 31.

The data have been observed from a different perspective using a pie chart to analyze further how much of the downloads are covered by each distance class. Therefore the pie charts in Figure 31 depict the distribution of the download rate depending on the similarities between the typosquatting packages and their attack targets. In the Levenshtein distance, for example, it can be seen that all typosquatting packages with a Levenshtein distance of 1 account for about 50% of the total downloads. For the Levenshtein distance of 2, about 90% of all downloads are covered. The Damerau-Levenshtein distance also covers about 90% of the downloads starting from a distance of 2. Compared to the Levenshtein distance, a Damerau-Levenshtein distance of 1 covers significantly more downloads than the most strict Levenshtein distance threshold. For the pie charts of the Jaro-Winkler distance and the Gestalt-Pattern algorithm, the threshold was chosen to cover about 90% of the downloads, following the minimum requirements set by the previous pie charts. Thus the threshold for such coverage is set to 0.89 for the Jaro-Winkler distance and 0.8 for the Gestalt-Pattern algorithm. The distribution of the Modified-Damerau-Levenshtein distance covers 90% of all downloads from a distance of up to 1.5. As with the original Damerau-Levenshtein distance, most downloads are covered by the smallest distance of 0.5.

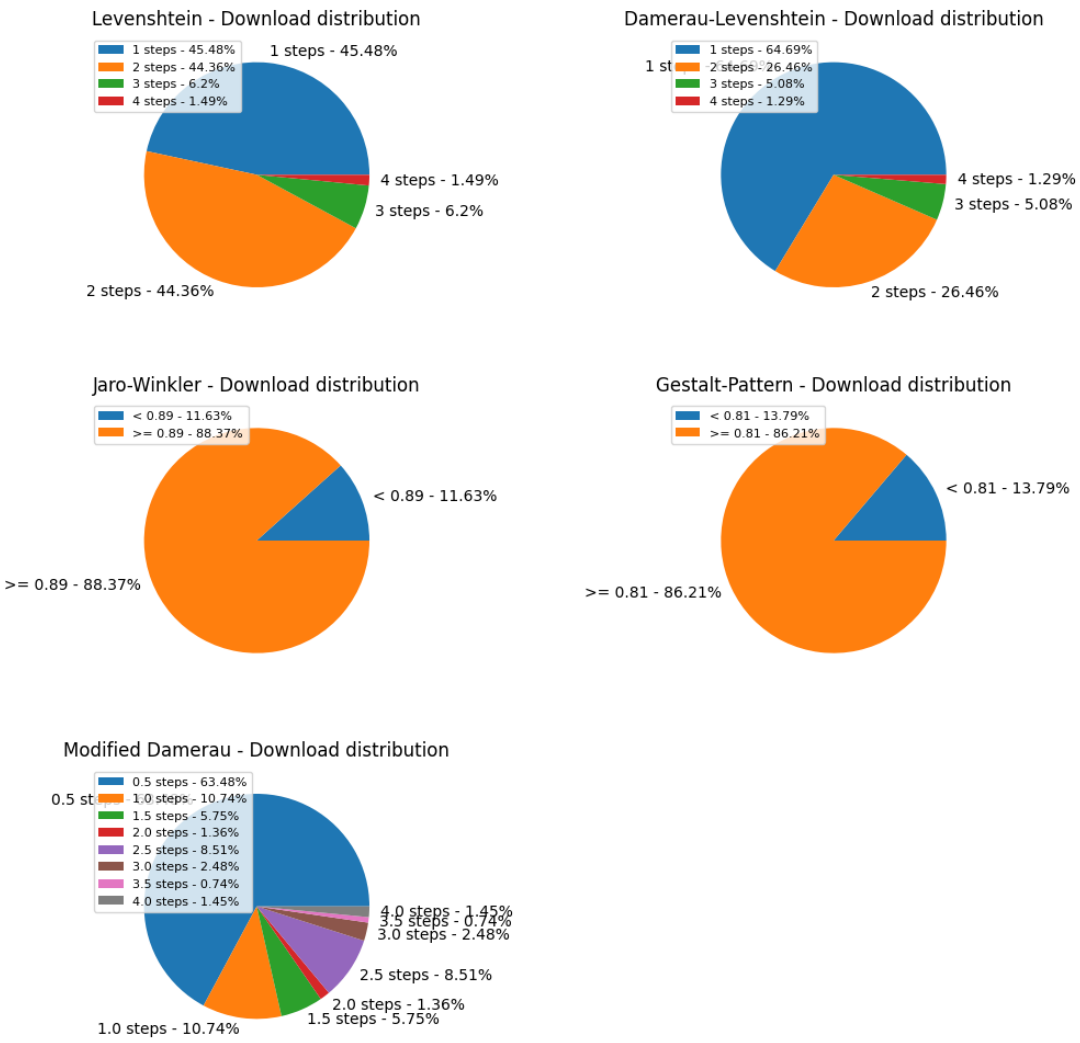

Figure 31: Distribution of downloads based on similarity

Upon further inspection, the packages with the lowest number of downloads have, on most days, the same number of downloads. For example, consider the seven libraries with the lowest downloads: *url-w.parse*, *vue-style-gloder*, *xpostcss-loadsr*, *follow-rdeirects*, *traecr*, *vniyl-fs*, *wcebpack-bunde-analyzer*, *wnode-acche* and *xbcrypt-nohejs*. For some of the seven typosquatting packages, the chances of a typo are relatively low. For instance, to make a typo in the library node-cache, one must accidentally press the key for "w" and transpose the letters "a" and "c" to install the typosquatting package wnode-acche. These two keys are not remotely adjacent to their neighboring letters. For further analysis, the download history of the typosquatting packages has been investigated. The download history can be seen in Figure 32. Except for very few differences, all seven have an almost identical download history, indicated by the overlapping of the functions (laying on top of each other). The probability that one needs all seven libraries and makes, in addition, the same typing errors is very unlikely. One possible explanation for the downloads that still appear in the data could be that a mirror, a copy of a repository, made copies and downloaded the packages at the time of observation, as shown in Figure 32. Alternatively, it is possible that a security researcher downloaded the packages for research purposes. Therefore it was checked for each day whether all packages had been downloaded on that specific day. If that was the case, the download value represented by most of the packages was taken. For instance, on 23rd October 2022, all packages were downloaded twice, except for one, which was downloaded thrice. Therefore the noise value for that specific day was set to 2. This procedure was performed for each day until 30th January 2023.

The sum of the noise values for each day was then summed and divided by the number of days since all packages were available.

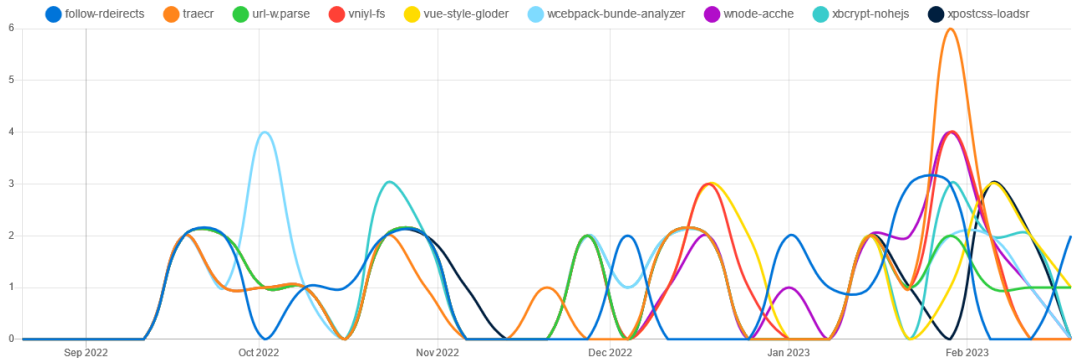

Figure 32: Typosquatting package download noise

The result of this normalization can be seen in the adjusted pie chart in Figure 33. 80% of all downloads are now caused by typosquatting packages with a minimum Damerau-Levenshtein distance of 1 to their attack target instead of the previous 67%. For the Levenshtein distance, the download rate covered by the lowest distance increased from 47% to 59%, and for the modified version of the Damerau-Levenshtein distance, from 65.06% to 79.85%. The limits of the Jaro-Winkler and the Gestalt-Pattern algorithm have thus been adjusted accordingly and now only need to meet the 80% coverage requirement provided by the previous distances. They do so at a threshold of 0.95 for the Jaro-Winkler similarity and 0.86 for the Gestalt-Pattern algorithm. The threshold values for the various string metrics are therefore defined as follows:

- Levenshtein: 2
- Damerau-Levenshtein: 1
- Jaro-Winkler: 0.95
- Gestalt-Pattern: 0.86
- Modified Damerau-Levenshtein: 0.5

These values were selected to have equal coverage of 80% of the number of downloads while minimizing a possible false positive value. Additionally, the targets of the typosquatting packages have been examined, resulting in the insight that approximately 90% of the packages targeted the top 1000 most depended upon libraries[14].

[14]https://gist.github.com/anvaka/8e8fa57c7ee1350e3491

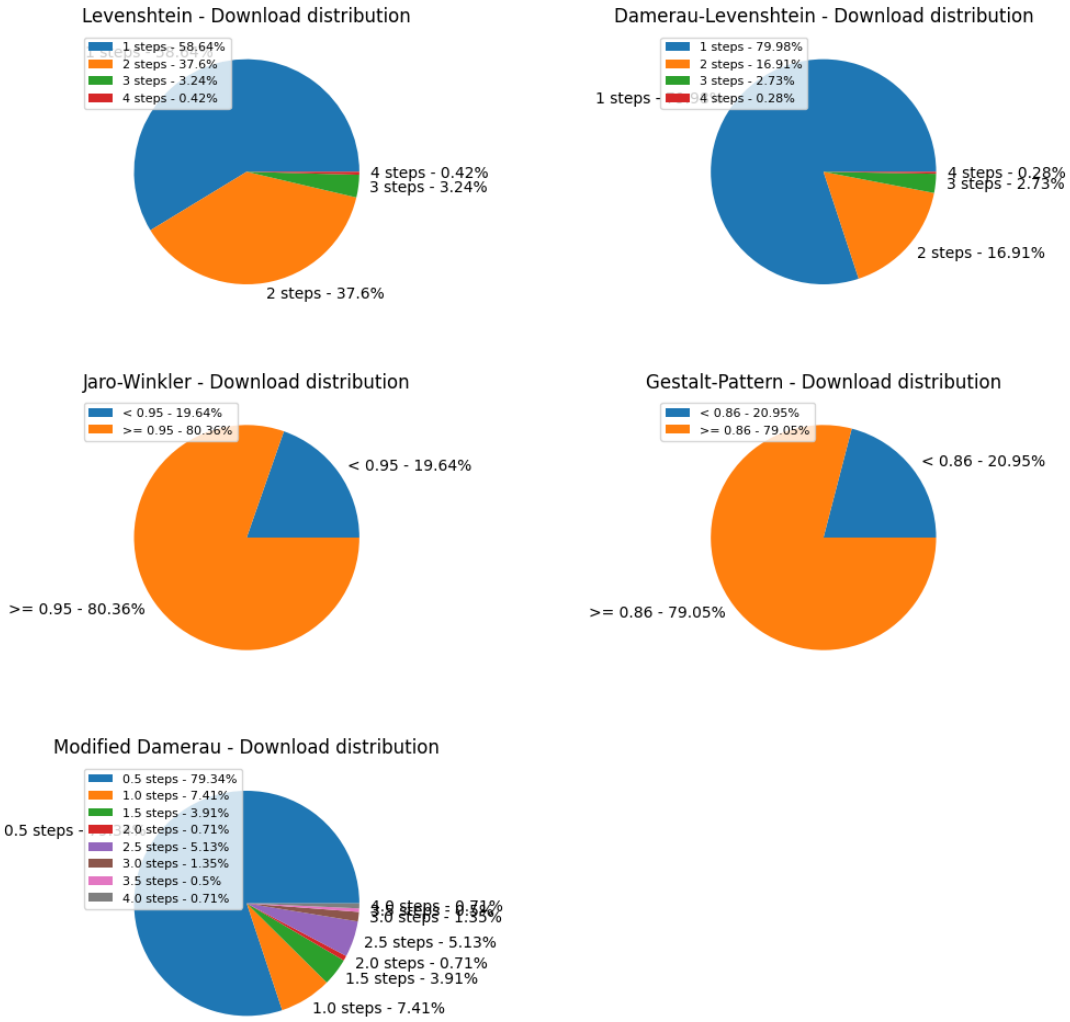

Figure 33: Adjusted distribution of downloads based on similarity

6.2.1. Results

The above insights revealed that about 90% of the typosquatting packages targeted the most popular libraries. Therefore, the various string metrics were evaluated in terms of accuracy on the top 5 most popular libraries. The threshold values for the evaluation were derived from the section above. The five libraries were then tested on the entire npm ecosystem, and the resulting typosquatting candidates were added to a list. Each typosquatting candidate was then checked to verify whether it was a typosquatting package confirmed by npm. To determine whether a package name had been used as a typosquatting package in the past, the metadata was utilized. Finally, the set of confirmed cases was divided by the set of all typosquatting candidates for the respective target. The result of this process can be seen in Figure 34.

For each string metric, a runtime test has been performed. Figure 35 below displays a regression of the runtime in seconds depending on the number of comparisons. An AMD Ryzen 5 5600X 6x 3.70GHz processor and 32 GB RAM were used to perform the calculations. Additionally, a slightly adjusted version of the Python Modified-Damerau-Levenshtein has been implemented to improve the runtime. This was done by converting some aspects of the algorithm, for instance, the types, into C code. However, some aspects are still written in native Python, e.g., the lookup of neighboring keys.

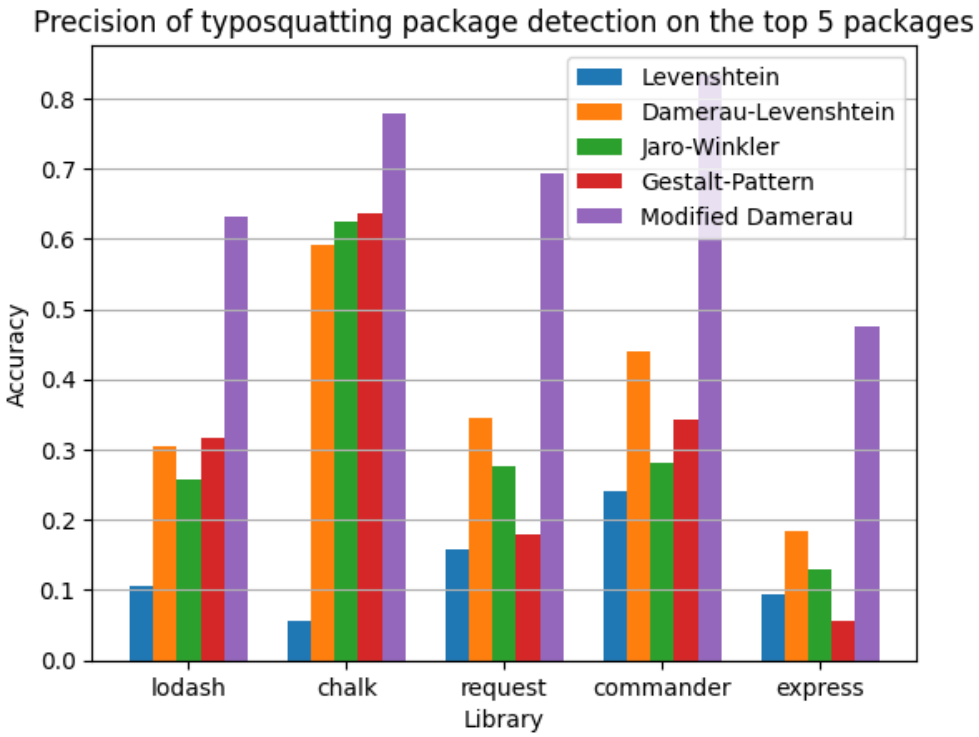

Figure 34: Precision of various string metrics in detecting typosquatting packages

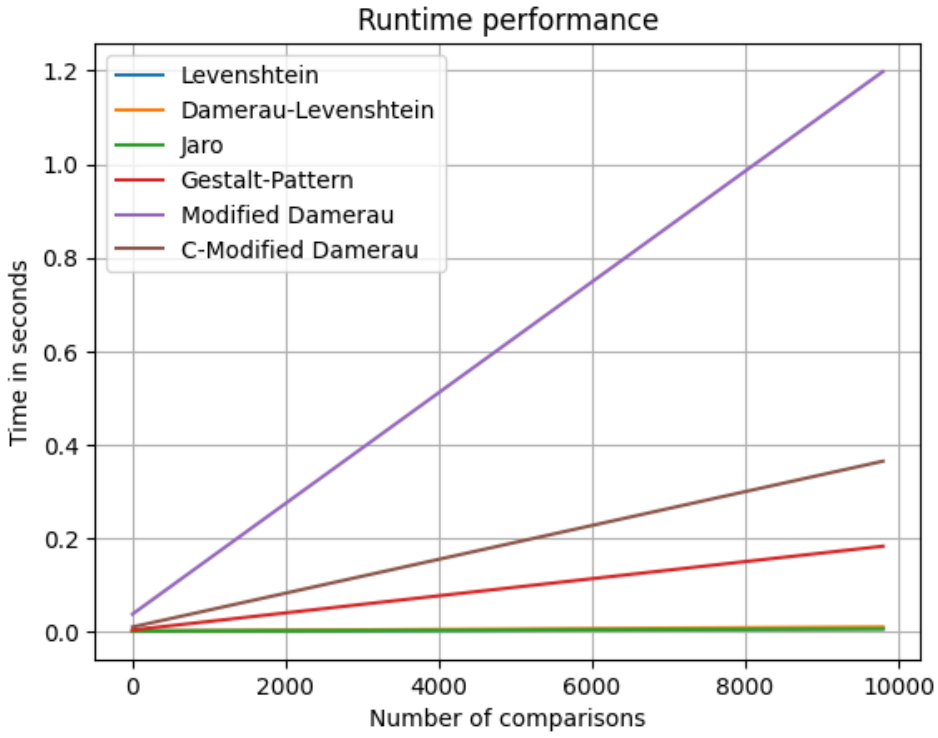

Figure 35: Runtime comparison with C code modification

Running the above metrics on the entire npm repository would require 5.29×10^{12} comparisons (2.3 million packages to the power of 2), whereby the symmetry is left out of the equation for the time being. The required computation time using the hardware specified above can be seen in Table 9. The values in the table are rounded to the next whole integer.

Metric	Seconds	Days
Levenshtein	4515722	52
Damerau-Levenshtein	8938630	103
Jaro-Winkler	3589921	42
Gestalt-Pattern	150078889	1737
Modified Damerau	880570145	10192
Optimized M-Damerau	206687399	2392

Table 9: Runtime for the whole npm ecosystem

6.2.2. Discussion & Interpretation

The results given in section 6.2.1 are promising. Trivially, it can be assumed that the stricter the threshold or the criteria chosen, the lower the false positive rate. In contrast, looser criteria provide for a higher false positive rate. However, stricter criteria may also increase the false negative rate since the criteria have been chosen so strictly that not all cases of true typosquatting packages are covered. Figure 29 indicates that only a tiny fraction of the typosquatting packages appear to be effective. The majority of typosquatting packages, or nearly 95% of typosquatting packages, are downloaded on average less than once a day. Therefore only effective typosquatting packages have to be focused on. This has the advantage of allowing the algorithms and metrics to be better tailored to the effective typosquatting packages and thus allowing stricter similarity metrics to be defined to reduce the number of false positives.

Suppose one selects the criteria in such a way, as indicated in the section 6.2, and applies these to the five most popular libraries. In that case, one receives the results from Figure 34. It can be seen that the metric, which performed almost everywhere poorest, is the Levenshtein distance. This result also mirrors the experiences perceived in the literature [47]. The Jaro-Winkler and Gestalt-Pattern metrics, as well as the Damerau-Levenshtein distance, performed much better than the Levenshtein distance and achieved similar results across all packages, with a few exceptions. The differences between the three string metrics are about 10-15%. The modified Damerau-Levenshtein distance, on the other hand, achieved significantly better results for all packages. Depending on the package, an increase of 20-50% percent could be achieved.

However, not all approaches are feasible due to their accuracy and runtime. The modified version of the Damerau-Levenshtein distance, for instance, performs the worst in terms of runtime and takes almost three times as long for the same amount of comparisons as the Gestalt-Pattern algorithm. There are several reasons for the higher runtime. The most obvious reason is that the modified version of the Damerau-Levenshtein distance is written in pure Python, while the other metrics are partially based on C implementations. If one transfers some simple elements of the original algorithm into C, the performance increases by approximately 300%, making the runtime almost equal to that of the Gestalt-Pattern algorithm. This increase and the various string metrics' runtime can be observed in Figure 35 or in Table 9. Further optimization possibilities would be using an adjacent matrix to represent neighboring keys instead of a dictionary of lists. Suppose one maximized the performance of this presented metric. In that case, one would likely achieve a runtime equivalent to the original Damerau-Levenshtein distance.

As one can see in Table 9, even for the fastest algorithm of the five presented here, a minimum runtime of 40 days is required to examine the entire npm ecosystem for all typosquatting candidates. A runtime that is deemed acceptable, but as discussed earlier, it results in a significantly higher number of false positives. There are some possibilities to

reduce the runtime of the more accurate string metrics. The first would be to parallelize the algorithm. The parallelization can be realized by splitting the set of all packages into n subsets and running them on n different systems. This would reduce the runtime by n times. One must also consider that this examination must be performed only once on all packages. After this, only new packages have to be examined with this method. The second approach is based on the knowledge that about 90% of the typosquatting packages in the collected dataset choose the 1000 most popular packages as a target. Hence, one would reduce the number of comparisons from 5.29×10^{12} to 2.3×10^{9}. The runtime for such an approach can be seen in Table 10. Alternatively, one could use the Modified-Damerau-Levenshtein distance to prohibit packages that have a distance of 0.5 from already existing packages. However, there is one crucial disadvantage of the Modified-Damerau-Levenshtein distance. Due to the modifications, it loses its symmetry property. The cause for this is the different cost values when inserting or removing letters. In the original version, a cost value of 1 was calculated for both removal and insertion. In the modified version, the value for removing a letter is always 0.5, and inserting a letter is calculated dynamically, depending on the position on the keyboard of the letter to be compared. For instance, in the algorithm, an insertion must be performed to transform a string from ”rect” to ”react”, but an omission must be performed to transform a string from ”react” to ”rect”. This issue can be resolved in two ways, either by modifying the insertion and deletion in such a way that the costs are equal or by the usage of the symmetry property of iterating through package comparisons. However, the latter solution would increase the computation time by two.

Metric	**Seconds**	**Hours**
Levenshtein	1963	0.55
Damerau-Levenshtein	3886	1.08
Jaro-Winkler	1561	0.43
Gestalt-Pattern	65252	18.1
Modified Damerau	382857	106.35
Optimized M-Damerau	89864	24.96

Table 10: Estimated runtime for the top 1000 most dependet upon packages

From the sheer size of possible packages to compare with each other, it can be assumed that the accuracy also suffers under the same conditions. The execution of the algorithm would therefore lead to a large number of false positives when applied to the entire npm ecosystem while applying it to the top 1000 libraries yields significantly better results. Thus, the hypothesis as to whether it is possible to find and mitigate typosquatting packages at the package manager level can be answered with a conditional ”yes”. In order for the method to be efficient, one could apply them to a certain set of packages, the most popular ones. Therefore, this option is ideal since 90% of the analyzed typosquatting packages were found to be targeting the top 1000 most popular packages. Applying the method to the entire npm ecosystem would result in a prohibitively high runtime and an unmanageable number of false positives, rendering it impractical and meaningless for analysis. Another approach would be using the Modified-Damerau-Levenshtein distance as a prevention measure; packages with a distance of 0.5 from another package could be prohibited using the Modified-Damerau-Levenshtein distance. As of now, the algorithm of the Modified-Damerau-Levenshtein distance would take approximately 90 seconds, while the Levenshtein would take 2 seconds, to compare one package to all other packages. Further code optimization of the Modified-Damerau-Levenshtein may significantly decrease the runtime, but not faster than the Levenshtein distance. Nevertheless, the methods

mentioned here are only one of the first steps to detecting typosquatting packages. This part has only dealt with the detection of typosquatting candidates. The next chapter will present a method that detects whether a typosquatting candidate is also a typosquatting package.

6.3. Detection of Malicious Packages

After searching for suitable typosquatting candidates using string metrics, these candidates must be examined for malicious code to categorize them as typosquatting packages. The intention of maliciousness is evident if the primary goal of a package can be assigned to one of the primary goal groups mentioned in section 3.2. Tools like modules, functions, or libraries are usually used to achieve these primary goals, which was additionally confirmed in the mentioned pieces of literature [41], [43], [45]. Therefore, the following hypothesis will be proposed: "The intention of a package can be identified by the combination of the libraries used and several other characteristics". The other characteristics are namely:

- Usage of the eval function
- Usage of script hooks
- Presence of IP or URL addresses
- Existence of binary or script files

These characteristics were extracted from the literatures mentioned in chapter 4 for instance [43], [45], to name a few. In order to test the hypothesis, 396 typosquatting packages were analyzed manually for the libraries used and their characteristics.

To compare the source code of a typosquatting package and the target attack code, a delta, similar to a Git diff[15], was initially generated between the typosquatting package's source code and the attack target's source code. Subsequently, newly introduced libraries and modules have been added to a set from the delta of the source codes. Thus iteratively, a set of libraries was formed. Additionally, the typosquatting packages were checked for the characteristics mentioned above. Subsequently, for each typosquatting package, a data record was created, in which the set of the used libraries was mapped into a boolean feature vector. A small sample can be seen in Table 11. A total of 15 features were collected for each of the typosquatting packages. If a typosquatting package was obfuscated, it was deobfuscated and included in the dataset; if this was not possible, the respective package was discarded. Though obfuscated code may not necessarily be malicious, the use of obfuscated code goes in itself contrary to the nature of OSS. Additionally, as a stakeholder, it would be reasonable to prevent the introduction of obfuscated code due to the higher risks associated with it. Finally, these typosquatting packages were labeled as malicious. The attack targets of the typosquatting packages were used as datasets for benign packages. The same features as for the typosquatting packages were used and added. A total of 396 datasets were collected as a sample for malicious packages and 211 for benign packages. The two datasets were combined. Furthermore, from these datasets, a training dataset and a test dataset were created for the Decision Tree. The ratio of training data to test data was set to 80:20, a ratio that is often used for splitting [56].

	package_name	entry_through_script	fs	node-fetch	has_ip_or_address	child_process	https_or_http	crypto	os	node-serialize	axios	has_bash_file
0	1y8n	True	True	False	True	True	True	False	False	False	False	False
1	levl	True	False	False	True	True	False	False	False	False	False	False
2	mongoos	True	False	False	True	True	False	False	False	False	False	False

Table 11: Small sample of feature dataset

For the creation of a Decision Tree as well as the Random Forest, the Python library *scikit-learn* has been used. For the hyperparameters, the default values have been taken. However, according to the paper [57], the difference between the usage of Entropy or Gini coefficients is diminishing; therefore, the Gini coefficient has been used as the impurity measure of choice. The usage of a Random Forest reduces overfitting and provides a more robust approach than the usage of a Decision Tree [28]. Both models were trained on 80% of the dataset and tested on 20% of the dataset. Additionally, during the training of the Random Forest, multiple Decision Trees were generated using bootstrapping, where each tree is trained on a random subset of the training data. The out-of-bag sets are then

[15]Git diff is a command that shows the differences between two sets of code

used to evaluate the performance of each tree on the samples not included in its bootstrap sample. This helps to estimate the generalization performance of the model. Additionally, the whole Random Forest has been evaluated on the test dataset not included during the training phase.

The model was also tested on a dataset from the Backstabber's Knife Collection [35] to check if the model was not simply tailored to the internal dataset. For this purpose, the model, which was trained using the data collected for this thesis, was tested with data from external sources. From the 3124 npm malicious packages, about 350 packages were randomly selected, and their features were extracted and fed in as a dataset. From these, 23 were removed due to duplicates. The choice of sample size is derived from the literature [58] and is based on the population of 3124 malicious packages, with a confidence interval of 95%. One has to mention that the information extracted from the malicious packages provided by the Backstabber's Knife Collection was done using a script, and no manual inspections were performed.

6.3.1. Results

The Decision Tree was able to classify 95.86% of the test data from the internal dataset correctly. The exact results are shown in Table 12, describing the performance of the binary classification model, in this case, that of the Decision Tree. The terms *Malicious* and *Benign* are the classes of the data entries. For the evaluation of each class, a precision, a recall, and an F1 score can be specified. In Table 12, a total of 121 records have been tested, with 40 being true benign and 81 being true malicious. The precision for the class Malicious is 0.97, meaning that if the Decision Tree classifies a package as Malicious, it is correct about 97% of the time. The recall value of the class Malicious with a value of 0.96, on the other hand, describes that the Decision Tree could identify 96% of all packages that were true Malicious, that is, 96% of all typosquatting packages were thus identified. The model's accuracy is 96%, meaning that if a classification was performed, the model was correct 96% of the time. The corresponding Decision Tree can be seen in Figure 36.

	Precision	**Recall**	**f1-Score**	**Support**
Benign	0.93	0.95	0.94	40
Malicious	0.97	0.96	0.97	81
Accuracy			0.96	121
Weighted Avg.	0.96	0.96	0.96	121

Table 12: Classification report for Decision Tree

The Decision Tree shown in Figure 36 has a depth of four and five nodes with six leaves. The first line of each node contains the condition. The first node's condition child_process <= 0.5" can be interpreted as follows: "Does the package use the module child_process?", where a value 0 means False, and a value 1 denotes True. The left path of a node represents yes, while the right path stands for no. A package is thus passed on from node to node and, depending on the respective condition in the node, is either passed on via the left or right path until it can be assigned to a leaf. The last line in a leaf represents the assigned class.

The accuracy of the Random Forest model, consisting of 1000 Decision Trees, is 97.52%. As a seed for splitting the test and training dataset, the random state value was left at 0. Consecutively, the exact results can be seen in Table 13. The classification report of the test performed on the dataset from the Backstabber's Knife Collection is depicted in Table 14.

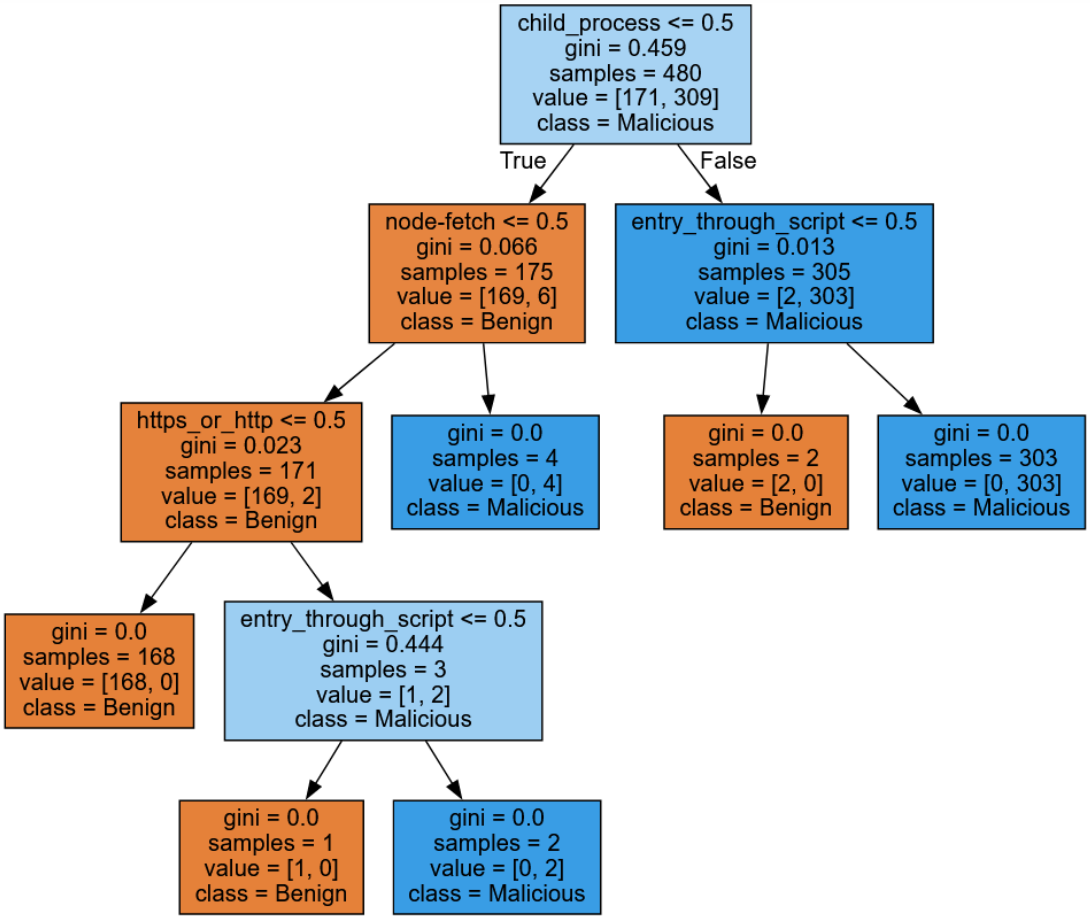

Figure 36: A Decision Tree instance based on the dataset

	Precision	**Recall**	**f1-Score**	**Support**
Benign	0.93	1.0	0.96	40
Malicious	1.0	0.96	0.98	81
Accuracy			0.98	121
Weighted Avg.	0.98	0.98	0.98	121

Table 13: Classification report for Random Forest

	Precision	**Recall**	**f1-Score**	**Support**
Benign	0.77	1.0	0.87	211
Malicious	1.0	0.81	0.90	327
Accuracy			0.88	538
Weighted Avg.	0.91	0.88	0.89	538

Table 14: Classification report for Random Forest on external data

6.3.2. Discussion & Interpretation

The methods carried out in section 6.3 show promising results. In evaluating the Decision Tree, an accuracy of 95.86% was achieved. The evaluation was performed on 121 test data, of which 40 were benign, and 81 were malicious packages. Of all packages that were identified as benign, 93% were correctly classified, while of all packages classified as malicious, 97% were correctly classified as such. On the other hand, in the paper [41], a precision of 98% was achieved based on their own dataset. The recall value for the class Benign of the presented Decision Tree is 95% and for the Malicious class 96%, i.e., if a package is truly harmless, it is recognized as such 95% of the time, and if it is malicious, it is recognized as such 96% of the time. Thus the recall values are clearly above the recall value from the literature [41], displayed in Table 4. There, the value is 43%, meaning that only 43% of all malicious packages from the dataset could be detected as such. The number of packages evaluated on external datasets is as follows:

- 538 packages were used during the evaluation in this thesis
- The paper [41] used 372 packages during evaluation
- The paper [48] used 114 packages during the evaluation

One problem that was considered in the results is the variability. The model's accuracy depends on the data with which it was trained. Since some randomization is involved in splitting the data into test and training data, one gets different results depending on which seed one chooses. For example, with the Decision Tree trained in section 6.3, a random state value of 5 yields better results than a random state value of 0. Therefore, an extended form of the Decision Tree was used, the Random Forest, which reduces such variability using bootstrapping and out-of-bag methods. This results in a slightly improved accuracy and a more robust model, which according to the literature mentioned in section 2.6.3, is less susceptible to overfitting. The performance of the Random Forest model is demonstrated in Table 13.

In order to check whether overfitting may have occurred, this model was additionally tested on an external dataset, and its results are presented in Table 14. It can be seen here that the accuracy has decreased in certain areas. There are several possible reasons for this. One possible explanation is that the samples collected for this work are biased; for instance, many samples can originate from the same attacker. As a result, the tests perform better on the samples collected for this thesis, as they may contain that bias, but perform worse on external datasets, as the bias may not be present in these datasets. Manual analysis has indicated that a large proportion of malicious packages can be assigned to the same attackers. As a result, packages from the same attacker may be more similar to each other but more different from malicious packages from other attackers. However, this can be easily fixed by either adding the malicious packages from the Backstabber's Knife Collection or other sources to the datasets or by introducing a custom weighting such that the influence of the malicious packages is normalized. For example, packages with the same source will only be added once in the dataset. Nevertheless, when testing an external dataset the proposed model performed better than the model in paper [41], a higher precision, as well as recall, could be achieved, on a larger dataset.

Another apparent cause is that the datasets from the Backstabbers Knife Collection were automatically extracted using scripts. This has two possible implications. The first implication would be an error in the script, but since the script was also re-tested on the self-collected datasets and the results were compared to the manual analysis, this error is less likely to occur. The second implication is that automatic feature collection only works on malicious packages that have not been obfuscated entirely. The reason for this is that the feature extraction script uses regular expressions. Transforming the source code into an AST and extracting the modules might result in better performance, but another solution is needed for highly obfuscated source code. Therefore with highly obfuscated code, one can hardly recognize the use of libraries, which suggests that they exhibit characteristics of benign packages. This hypothesis is indicated by the values

from the classification report presented in Table 14. The precision of the class Benign and the recall of the class Malicious are conspicuous compared to the other scores. A Precision value of the class Benign of 0.77 means that 77% of the packages that were classified as benign were actually benign. Nevertheless, 100% of the malicious packages that were classified as such were correctly classified. Conversely, only 81% of the malicious packages were detected. The existence of obfuscated malicious code could explain such a characteristic in the classifications report.

Thus, the modeling of such machine learning methods was considered under different aspects, allowing for promising results to be achieved. Regarding the quality of the model, it was able to significantly outperform the model from the literature [41] in terms of the evaluation based on internal datasets and external datasets. Particularly in evaluating external datasets, better results could be achieved in the various metrics, in some cases by 20-50%. Nevertheless, it would be necessary to investigate more closely how the ratio of malicious packages to benign packages in the test dataset affects the model's performance.

7. Conclusion and Future Work

The question of whether differences exist in the attack frequency depending on the package manager can be answered with a clear yes. Although, the distribution shown in Figure 27 is only an excerpt of the malicious packages that are or were in the various package managers. The difference in the number of malicious packages depending on package managers is significant enough to assume that this is not due to a bias of the source material. The mentioned sources, Sonatype, Phylum, and Snyk, also cover almost all the package managers examined here. Furthermore, the distribution of the number of attacks on each package manager is indicated in the paper [35]. The package managers npm and PyPI are the most frequent attack target. According to the examination performed on the various package managers, the following four characteristics contribute the most to the protection of the package manager. According to the correlation matrix, these are CAPTCHAs, the absence of Install hooks, the use of repositories as the source for the library in question, and a more complex naming system. Nevertheless, with these findings, it must be questioned how accurate the correlation matrix is due to the shortcomings mentioned in section 6.1.2. The conclusion drawn from this investigation is that, among the mentioned characteristics, only CAPTCHAs can be implemented without encountering additional issues. Implementing the remaining features leads to problems, which could cause breaking changes and thus endanger already existing software products. The analysis presented in section 6.1 can provide valuable insights in the event that a new package manager is developed. Thus the question of whether one can derive approaches or methods from less susceptible package managers can be answered with a yes, the realization of such features for already existing package managers, with a no or not without sizeable additional expenditure. Therefore, subsequent research in this thesis focused on the detection of malicious packages, particularly the detection of typosquatting packages.

Different string metrics were compared and evaluated to detect typosquatting candidates. As a result, a new string metric was derived. A modified variant of the Damerau-Levenshtein distance was implemented. This string metric additionally uses a mapping of the keyboard to adjust the cost accordingly. It has been shown in this study that the Modified-Damerau-Levenshtein variant is significantly more accurate than the other string-matching algorithms examined in this thesis and, thus, more accurate than the majority of the metrics in the mentioned literature. However, there is still a need for optimization in terms of runtime. Moreover, this work has shown that applying even the fastest string-matching algorithm of the five mentioned requires a minimum runtime of about 40 days to compare all packages from the npm ecosystem (neglecting symmetry). Instead, one could only apply the string metrics to new packages or the most popular packages. This solution is supported by the statistical analysis performed in this thesis. For instance, 90% of the typosquatting packages collected for this thesis targeted the top 1000 libraries with the most dependents. Another finding is that the effectiveness of typosquatting packages with a (Damerau)-Levenshtein distance greater than 2 is approaching 0. In order to verify that a found typosquatting candidate is a typosquatting package, a Decision Tree was trained and used. This approach has been further improved by using a Random Forest model, which correctly classifies the packages with an accuracy of 98% on internal datasets and 88% on external datasets. The model can be further optimized by improving the data basis or performing a hyperparameters adjustment. The question of whether it is possible to detect typosquatting packages on the package manager level can be answered with a theoretical yes, whereby it must be said that the presented system is weak against obfuscated code. Here possibly, the works of literature [59], [60] might provide more insight. These works of literature were able to identify obfuscated code with high precision. Alternatively, the proposed approach can be combined with dynamic code analysis, where the behavior of the malicious packages can be observed instead of inferring behavior through the source code.

Ultimately, the presented method's performance must be tested further in the npm ecosystem. An outlook for further investigations would be applying the presented proce-

dure to the 1000 most popular libraries and newly uploaded packages and evaluating the proposed methods in terms of precision. In terms of recall, this is hardly possible for the npm ecosystem because one has to know all malicious packages located therein. On the other hand, the method provided could also be tested on the general detection of malicious packages. Hence, based on the aforementioned reasons, the subsequent perspective is suggested:

- Optimization of runtime performance for the Modified-Damerau-Levenshtein distance
- Optimization of the classifier by providing better datasets and hyperparameter adjustment
- Extend the proposed approach by adding methods to detect obfuscated code or use methods resistant to obfuscated code
- Testing the proposed methods on the most popular packages (for typosquatting packages and malicious packages in general)

In conclusion, the analysis in this thesis suggests that while some mitigation measures may be challenging to implement, alternative approaches show promising results. By incorporating these approaches, a 20-50% increase in precision in detecting typosquatting candidates was achieved, surpassing all other string metrics tested in all five cases. Additionally, an accuracy of 88% was achieved using a Random Forest model without hyperparameter optimization evaluated on an unknown external data set. On average, the model's precision and recall are 91% and 88%, respectively, with potential for further improvement.

8. Bibliography

[1] Z. Whittaker, "Two years after wannacry, a million computers remain at risk," *TechCrunch*, 2019-05-12. [Online]. Available: https://techcrunch.com/2019/05/12/wannacry-two-years-on/ (visited on 12/21/2022).

[2] B. Chappell and S. Neuman, "U.s. says north korea 'directly responsible' for wannacry ransomware attack," *NPR*, 2017-12-19. [Online]. Available: https://www.npr.org/sections/thetwo-way/2017/12/19/571854614/u-s-says-north-korea-directly-responsible-for-wannacry-ransomware-attack (visited on 12/21/2022).

[3] T. Haselton, "Credit reporting firm equifax says data breach could potentially affect 143 million us consumers," *CNBC*, 2017-09-07. [Online]. Available: https://www.cnbc.com/2017/09/07/credit-reporting-firm-equifax-says-cybersecurity-incident-could-potentially-affect-143-million-us-consumers.html (visited on 12/21/2022).

[4] D. H. Kass, *Fbi: Covid-19 cyberattacks spike 400% in pandemic - mssp alert*, 2020. [Online]. Available: https://www.msspalert.com/cybersecurity-news/fbi-covid-19-cyberattacks-spike-400-in-pandemic/ (visited on 12/21/2022).

[5] A. Luttwak and A. Schindel, *Log4shell 10 days later: Enterprises halfway through patching | wiz blog*, 2021. [Online]. Available: https://www.wiz.io/blog/10-days-later-enterprises-halfway-through-patching-log4shell (visited on 12/21/2022).

[6] B. Mayhew, S. Magill, M. Howard, *et al.*, *State of the software supply chain 2021: The 7th annual report on global open source software development*, Sonatype, Ed., 2021. [Online]. Available: https://www.sonatype.com/hubfs/Q3%202021-State%20of%20the%20Software%20Supply%20Chain-Report/SSSC-Report-2021_0913_PM_2.pdf?hsLang=en-us (visited on 12/06/2022).

[7] L. Mcbride, *Software supply chains: An introductory guide*, 2021. [Online]. Available: https://blog.sonatype.com/software-supply-chain-a-definition-and-introductory-guide (visited on 12/24/2022).

[8] A. Aklson, A. Del Duke, A. VanKanegan, *et al.*, *State of the software supply chain report: Sonatype's industry-defining research on the rapidly changing landscape of open source*, Sonatype, Ed., 2022. [Online]. Available: https://de.sonatype.com/state-of-the-software-supply-chain/introduction (visited on 11/18/2022).

[9] GitHub, *Build software better, together*. [Online]. Available: https://github.com/search (visited on 12/21/2022).

[10] F. Bals, "2022 ossra discovers 88% of organizations still behind in keeping open source updated," *Synopsys*, 2022. [Online]. Available: https://www.synopsys.com/blogs/software-security/open-source-trends-ossra-report/ (visited on 12/21/2022).

[11] J. Humble and D. G. Farley, *Continuous delivery: Reliable software releases through build, test, and deployment automation* (A Martin Fowler signature book). Upper Saddle River, NJ, Toronto, and London: Addison-Wesley, 2011, ISBN: 978-0321601919.

[12] M. Taylor, "Defending against typosquatting attacks in programming language-based package repositories," Bachelor Thesis, University of Kansa, Kansas, 2020.

[13] N. P. Tschacher, "Typosquatting in programming language package managers," Bachelor Thesis, University of Hamburg, Hamburg, 2016. [Online]. Available: https://incolumitas.com/data/thesis.pdf.

[14] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001, ISSN: 0360-0300. DOI: 10.1145/375360.375365.

[15] M. A. Jaro, "Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida," *Journal of the American Statistical Association*, vol. 84, no. 406, p. 414, 1989, ISSN: 01621459. DOI: 10.2307/2289924.

[16] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[17] L. Boytsov, "Indexing methods for approximate dictionary searching," *ACM Journal of Experimental Algorithmics*, vol. 16, 2011, ISSN: 1084-6654. DOI: 10.1145/1963190.1963191.

[18] L. Jiang, *Levenshtein demo: Levenshtein distance calculator*, 2018-07-13. [Online]. Available: https://phiresky.github.io/levenshtein-demo/ (visited on 02/08/2023).

[19] F. J. Damerau, "A technique for computer detection and correction of spelling errors," *Communications of the ACM*, vol. 7, no. 3, pp. 171–176, 1964, ISSN: 0001-0782. DOI: 10.1145/363958.363994.

[20] I. Ilyankou, "Comparison of jaro-winkler and ratcliff/obershelp algorithms in spell check," 2014. [Online]. Available: https://ilyankou.files.wordpress.com/2015/06/ib-extended-essay.pdf.

[21] W. E. Winkler, "String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage," pp. 354–359, 1990. [Online]. Available: https://www.researchgate.net/publication/243772975_String_Comparator_Metrics_and_Enhanced_Decision_Rules_in_the_Fellegi-Sunter_Model_of_Record_Linkage.

[22] J. W. Ratclif, *Pattern matching: The gestalt approach*, Dr. Dobb's, Ed., 1988. [Online]. Available: https://www.drdobbs.com/database/pattern-matching-the-gestalt-approach/184407970?pgno=5 (visited on 01/30/2023).

[23] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and Regression Trees*, 1st ed. Boca Raton and Ann Arbor, Michigan: Routledge and ProQuest, 1984, ISBN: 9781351460484.

[24] T. Segaran, *Programming collective intelligence: Building smart web 2.0 applications*, First edition. Beijing, Cambridge, and Paris: O'Reilly, 2007, ISBN: 9780596517601. [Online]. Available: https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=443469.

[25] J. R. Quinlan, "Induction of decision trees," pp. 81–106, 1986. [Online]. Available: https://link.springer.com/article/10.1007/BF00116251.

[26] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948, ISSN: 00058580. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[27] J. Frochte, *Maschinelles Lernen: Grundlagen und Algorithmen in Python*. München: Hanser, 2018, ISBN: 3446452915.

[28] L. Breiman, "Random forests," pp. 5–32, 2001. [Online]. Available: https://link.springer.com/article/10.1023/A:1010933404324.

[29] Y. Liu, Y. Wang, and J. Zhang, *New Machine Learning Algorithm: Random For: Third International Conference, ICICA 2012, Chengde, China, September 14-16, 2012, Revised Selected Papers* (Information Systems and Applications, incl. Internet/Web, and HCI), 1st ed. 2012. Berlin, Heidelberg: Springer Berlin Heidelberg, Imprint, and Springer, 2012, vol. 7473, ISBN: 9783642340628. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-34062-8_32.

[30] B. Efron, "Bootstrap methods: Another look at the jackknife," *The Annals of Statistics*, vol. 7, no. 1, 1979, ISSN: 0090-5364. DOI: 10.1214/aos/1176344552.

[31] T. Fawcett, "An introduction to roc analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006, ISSN: 01678655. DOI: 10.1016/j.patrec.2005.10.010.

[32] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*, Second edition. Beijing et al.: O'Reilly, 2019, ISBN: 9781492032618. [Online]. Available: https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=5892320.

[33] H. Aldawood and G. Skinner, *Contemporary cyber security social engineering solutions, measures, policies, tools and applications: A critical appraisal*, 2019. [Online]. Available: https://f.hubspotusercontent30.net/hubfs/8156085/WhitePaper%20-%20IJS%20-%20Contemporary%20Cyber%20Security%20Social%20Engineering%20Solutions[1].pdf (visited on 02/08/2023).

[34] T. Holgers, D. E. Watson, and S. D. Gribble, "Cutting through the confusion: A measurement study of homograph attacks," Paper, University of Washington, Washington, 2006. [Online]. Available: https://www.researchgate.net/publication/220881094_Cutting_through_the_Confusion_A_Measurement_Study_of_Homograph_Attacks.

[35] M. Ohm, H. Plate, A. Sykosch, and M. Meier, *Backstabber's knife collection: A review of open source software supply chain attacks*, 2020. [Online]. Available: http://arxiv.org/pdf/2005.09535v1.

[36] J. S. Meyers and B. Tozer, *Bewear! python typosquatting is about more than typos - in-q-tel*, In-Q-Tel, Ed., 2020. [Online]. Available: https://www.iqt.org/bewear-python-typosquatting-is-about-more-than-typos/ (visited on 11/18/2022).

[37] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Typosquatting and combosquatting attacks on the python ecosystem," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, IEEE, 2020, pp. 509–514, ISBN: 978-1-7281-8597-2. DOI: 10.1109/EuroSPW51379.2020.00074.

[38] A. Sharma, *Python packages upload your aws keys, env vars, secrets to the web*, 2022. [Online]. Available: https://blog.sonatype.com/python-packages-upload-your-aws-keys-env-vars-secrets-to-web (visited on 02/12/2023).

[39] C. Cimpanu, "Two malicious python libraries caught stealing ssh and gpg keys," *ZDNET*, 2019-12-04. [Online]. Available: https://www.zdnet.com/article/two-malicious-python-libraries-removed-from-pypi/ (visited on 02/12/2023).

[40] L. Martini, *Psa: There is a fake version of this package on pypi with malicious code · issue #984 · dateutil/dateutil*, GitHub, Ed., 2019. [Online]. Available: https://github.com/dateutil/dateutil/issues/984 (visited on 02/12/2023).

[41] A. Sejfia and M. Schäfer, *Practical automated detection of malicious npm packages*, 2022. DOI: 10.1145/3510003.3510104. [Online]. Available: https://arxiv.org/pdf/2202.13953.

[42] G. Liu, X. Gao, H. Wang, and K. Sun, "Exploring the unchartered space of container registry typosquatting," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, 2022, pp. 35–51, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/liu-guannan.

[43] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Proceedings 2021 Network and Distributed System Security Symposium*, A.-R. Sadeghi and F. Koushanfar, Eds., Reston, VA: Internet Society, 2021, ISBN: 1-891562-66-5. DOI: 10.14722/ndss.2021.23055.

[44] M. Čarnogurský, "Attacks on package managers," Bachelor Thesis, Masaryk University, Brünn, Tschechien, 2019. [Online]. Available: https://is.muni.cz/th/y41ft/thesis_final_electronic.pdf (visited on 11/23/2022).

[45] K. Garrett, G. Ferreira, L. Jia, J. Sunshine, and C. Kastner, "Detecting suspicious package updates," in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, IEEE, 2019, pp. 13–16, ISBN: 978-1-7281-1758-4. DOI: 10.1109/ICSE-NIER.2019.00012.

[46] M. Ohm, A. Sykosch, and M. Meier, "Towards detection of software supply chain attacks by forensic artifacts," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, M. Volkamer and C. Wressnegger, Eds., New York, NY, USA: ACM, 2020, pp. 1–6, ISBN: 9781450388337. DOI: 10.1145/3407023.3409183.

[47] M. Taylor, R. K. Vaidya, D. Davidson, L. de Carli, and V. Rastogi, *Spellbound: Defending against package typosquatting*, 2020. [Online]. Available: http://arxiv.org/pdf/2003.03471v1.

[48] M. Ohm, L. Kempf, F. Boes, and M. Meier, *Supporting the detection of software supply chain attacks through unsupervised signature generation*, 2021. [Online]. Available: https://arxiv.org/pdf/2011.02235.

[49] W. E. Yancey, "An adaptive string comparator for record linkage," *Statistical Research Division - U.S. Bureau of the Census*, pp. 1–24, 2004. [Online]. Available: https://www.census.gov/content/dam/Census/library/working-papers/2004/adrm/rrs2004-02.pdf.

[50] S. J. Grannis, J. M. Overhage, and C. McDonald, "Real world performance of approximate string comparators for use in patient matching," pp. 43–47, 2004. [Online]. Available: https://www.researchgate.net/publication/8353095_Real_world_performance_of_approximate_string_comparators_for_use_in_patient_matching.

[51] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "Package management security," 2008. [Online]. Available: https://www.researchgate.net/publication/228989436_Package_management_security.

[52] V. A., R. M. Patil, P. Kantanavar, and S. G., "A comparative study of various linux package-management systems," pp. 37–44, 2019. [Online]. Available: https://www.ripublication.com/acst19/acstv12n1_04.pdf.

[53] A. Field, *Discovering statistics using IBM SPSS statistics* (SAGE edge), 5th edition. Los Angeles et al.: SAGE, 2018, ISBN: 9781526419514.

[54] "Spearman rank correlation coefficient," in *The Concise Encyclopedia of Statistics*, Springer, New York, NY, 2008, pp. 502–505. DOI: 10.1007/978-0-387-32833-1{\textunderscore}379. [Online]. Available: https://link.springer.com/referenceworkentry/10.1007/978-0-387-32833-1_379#citeas.

[55] C. Braz, A. Seffah, and D. M'Raihi, "Designing a trade-off between usability and security: A metrics based-model," in *Human-computer interaction - INTERACT 2007*, ser. Lecture Notes in Computer Science, C. Baranauskas, P. Palanque, J. Abascal, and S. D. J. Barbosa, Eds., vol. 4663, Berlin: Springer, 2007, pp. 114–126, ISBN: 978-3-540-74799-4. DOI: 10.1007/978-3-540-74800-7{\textunderscore}9.

[56] V. R. Joseph, "Optimal ratio for data splitting," *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 15, no. 4, pp. 531–538, 2022, ISSN: 1932-1864. DOI: 10.1002/sam.11583.

[57] L. E. Raileanu and K. Stoffel, "Theoretical comparison between the gini index and information gain criteria," *Annals of Mathematics and Artificial Intelligence*, vol. 41, no. 1, pp. 77–93, 2004, ISSN: 1012-2443. DOI: 10.1023/B:AMAI.0000018580.96245.c6.

[58] P. Dattalo, *Determining Sample Size: Balancing power, precision, and practicality*. Oxford: Oxford University Press, 2008, ISBN: 9780195315493. DOI: 10.1093/acprof:oso/9780195315493.001.0001.

[59] A. Alazab, A. Khraisat, M. Alazab, and S. Singh, "Detection of obfuscated malicious javascript code," *Future Internet*, vol. 14, no. 8, p. 217, 2022. DOI: 10.3390/fi14080217.

[60] R. Kumar and A. R. E. Vaishakh, "Detection of obfuscation in java malware," *Procedia Computer Science*, vol. 78, pp. 521–529, 2016, ISSN: 1877-0509. DOI: 10.1016/j.procs.2016.02.097. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050916000995.

A. Appendix

A.1. Package Manager Properties

Dependent variables with higher values or ranking indicate a higher vulnerability. For instance, a package manager ranked eight is more frequently attacked than a package manager of rank three and, therefore, is more vulnerable.

Property	**Domain**
Malicious Packages	An integer representing the number of malicious packages
Malicious Package Rank	An integer representing the rank by the number of malicious packages
Ratio	A percentage representing the ratio between malicious packages and all packages
Ratio rank	An integer representing the rank by the ratio of malicious packages

Table A.1: Package Manager dependent variables

Instances of a property or independent variable with higher values or ranking indicate less beneficial for the package manager in terms of security, while smaller values indicate more beneficial measures. For instance for the property ”Captcha” two instances exists. Either ”yes” or ”no”. ”yes” receives the smaller value due to being more beneficial for a package manager in terms of security than ”no” CAPTCHAs. The instances listed in ”Domain” are ordered from most beneficial to least beneficial.

Property	**Domain**
Namespace system	”group and name”, ”group (optional) and name”, ”name”
Install Hooks	”no”, ”workaround” and ”partly”, ”native”
CAPTCHA	”yes”, ”no”
Mandatory VCS	”yes”, ”no”
Install through CLI	”possible, but unusual”, ”yes”
Code Scan	”yes”, ”no”
Popularity	Percentage
Publishing complexity	”complex”, ”advanced”, ”simple”

Table A.2: Package Manager independent variables

A.2. Malicious Package Feature

The following features were extracted from the packages metadata and source code. The value for each feature is a boolean denoting the existence or absence of that feature.

Features	**Notes**
axios	Library
child_process	Built-in module
crypto	Built-in module
dns	Built-in module
entry_through_script	Whether the package uses "Hook Install" scripts
eval	Whether the package uses the eval function
fs	Built-in module
has_bash_file	Whether files like ".sh", ".bash", ".bat" and ".zsh" are contained in the package
has_ip_or_address	Wether an IP or URL address could be found in the source codes
https_or_http	use of the libraries http or https
node-fetch	Library
node-serialize	Library
os	Built-in module
path	Built-in module
querystring	Built-in module

Table A.3: Features extracted for malicious code detection dataset

A.3. Core Implementations

Keyboard Mapping

```
en_layout = {
    "1": ["Q", "2"],
    "2": ["1", "3", "Q", "W"],
    "3": ["2", "4", "W", "E"],
    "4": ["3", "5", "E", "R"],
    "5": ["4", "6", "R", "T"],
    "6": ["5", "7", "T", "Y"],
    "7": ["6", "8", "Y", "U"],
    "8": ["7", "9", "U", "I"],
    "9": ["8", "0", "I", "O"],
    "0": ["9", "-", "O", "P"],

    "Q": ["1", "2", "A", "W"],
    "W": ["2", "3", "Q", "E", "A", "S"],
    "E": ["3", "4", "W", "R", "S", "D"],
    "R": ["4", "5", "E", "T", "D", "F"],
    "T": ["5", "6", "R", "Y", "F", "G"],
    "Y": ["6", "7", "T", "U", "G", "H"],
    "U": ["7", "8", "Y", "I", "H", "J"],
    "I": ["8", "9", "U", "O", "J", "K"],
    "O": ["9", "0", "I", "P", "K", "L"],
    "P": ["0", "-", "O", "L"],

    "A": ["Q", "W", "S", "Z"],
    "S": ["W", "E", "A", "D", "Z", "X"],
    "D": ["E", "R", "S", "F", "X", "C"],
    "F": ["R", "T", "D", "G", "C", "V"],
    "G": ["T", "Y", "F", "H", "V", "B"],
    "H": ["Y", "U", "G", "J", "B", "N"],
    "J": ["U", "I", "H", "K", "N", "M"],
    "K": ["I", "O", "J", "L", "M"],
    "L": ["O", "P", "K", "."],

    "Z": ["A", "S", "X"],
    "X": ["Z", "S", "D", "C"],
    "C": ["X", "D", "F", "V"],
    "V": ["C", "F", "G", "B"],
    "B": ["V", "G", "H", "N"],
    "N": ["B", "H", "J", "M"],
    "M": ["N", "J", "K"],
    "-": ["0", "P", "_"],
    "_": ["-"],
    ".": ["L"],
    "@": [],
    "/": [],
    "!": [],
    "(": [],
    ")": [],
    "~": [],
    "'": [],
    "*": []
}
```

Algorithm for Modified-Damerau-Levenshtein

```
def modified_damerau_levenshtein(s1, s2, keyboard_map):
    js_cost = 0
    if ".js" in s1 or ".js" in s2:
        s1 = s1.replace(".js", "")
        s2 = s2.replace(".js", "")
        js_cost = 0.5

    rows = len(s1) + 1
    cols = len(s2) + 1
    dist = [[0 for x in range(cols)] for y in range(rows)]
    for i in range(1, rows):
        dist[i][0] = i
    for i in range(1, cols):
        dist[0][i] = i
    for col in range(1, cols):
        for row in range(1, rows):
            if s1[row - 1] == s2[col - 1]:
                cost = 0
            else:
                cost = 0.5 if s1[row - 1].upper() in keyboard_map[s2[
                    col - 1].upper()] else 2
            dist[row][col] = min(dist[row - 1][col] + 0.5,
                                 dist[row][col - 1] + cost,
                                 dist[row - 1][col - 1] + cost)
            if row > 1 and col > 1 and s1[row - 1] == s2[col - 2] and
                s1[row - 2] == s2[col - 1]:
                dist[row][col] = min(dist[row][col], dist[row - 2][
                    col - 2] + 0.5)

    return dist[rows - 1][cols - 1] + js_cost
```

Decision Tree Classifier

```
import graphviz
import pandas as pd

from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

# Loading Typosquatting Target Data
target_packages = pd.read_csv('./data/target_package_data.csv')
target_packages.drop("modules_used", axis=1, inplace=True)
target_packages["Malicious"] = False

# Loading Typosquatting Packages Data
typo_packages = pd.read_csv("./data/typosquatting_data.csv")
typo_packages.drop(["dependency_imitation_injection", "
    uses_obfuscation"], axis=1, inplace=True)
typo_packages["Malicious"] = True

# Preparing internal Data
internal_data = pd.concat([typo_packages, target_packages], axis=0).
    sort_index(axis=1)
internal_data.dropna(inplace=True)
internal_data_X = internal_data.drop(["Malicious", "package_name"],
    axis=1)
internal_data_Y = internal_data["Malicious"]

# Split Data into Training and Test Data 80:20
X_train, X_test, Y_train, Y_test = train_test_split(internal_data_X,
    internal_data_Y, test_size=0.2, random_state=0)

# Generate Decision Tree Model based on internal Training Data
clf = DecisionTreeClassifier()
clf.fit(X_train, Y_train)

# Evaluate model on internal Test Data
accuracy = clf.score(X_test, Y_test)
y_pred = clf.predict(X_test)
print("Accuracy:", accuracy)
print(classification_report(Y_test, y_pred))
```

Random Forest Classifier

```
import graphviz
import pandas as pd

from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split

# Loading Typosquatting Target Data
target_packages = pd.read_csv('./data/target_package_data.csv')
target_packages.drop("modules_used", axis=1, inplace=True)
target_packages["Malicious"] = False

# Loading Typosquatting Packages Data
typo_packages = pd.read_csv("./data/typosquatting_data.csv")
typo_packages.drop(["dependency_imitation_injection", "
    uses_obfuscation"], axis=1, inplace=True)
typo_packages["Malicious"] = True

# Loading Backstabbers Collection Data
backstabber_packages = pd.read_csv("./data/backstabber_data.csv")
backstabber_packages.drop("modules_used", axis=1, inplace=True)
backstabber_packages["Malicious"] = True

# Preparing internal Data
internal_data = pd.concat([typo_packages, target_packages], axis=0).
    sort_index(axis=1)
internal_data.dropna(inplace=True)
internal_data_X = internal_data.drop(["Malicious", "package_name"],
    axis=1)
internal_data_Y = internal_data["Malicious"]

# Preparing external Data
external_data = pd.concat([backstabber_packages, target_packages],
    axis=0).sort_index(axis=1)
external_data.dropna(inplace=True)
external_data_X = external_data.drop(["Malicious", "package_name"],
    axis=1)
external_data_Y = external_data["Malicious"]

# Split Data into Training and Test Data
X_train, X_test, Y_train, Y_test = train_test_split(internal_data_X,
    internal_data_Y, test_size=0.2, random_state=0)

# Generate Random Forest Model on internal Training Data
rfc = RandomForestClassifier(n_estimators=1000, random_state=0,
    oob_score=True, bootstrap=True)
rfc.fit(X_train, Y_train)

# Evaluate model on internal Test Data
accuracy = rfc.score(X_test, Y_test)
y_pred = rfc.predict(X_test)
print("Accuracy:", accuracy)
print(classification_report(Y_test, y_pred))

# Evaluate Random Forest Model on external Data
y_pred = rfc.predict(external_data_X)
accuracy = rfc.score(external_data_X, external_data_Y)
print(accuracy)
print(classification_report(external_data_Y, y_pred))
```