Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services

Clustering IoT Malware Samples based on Binary Similarity

Bachelor's Thesis

Author
Márton László Bak

Advisors
Dorottya Futóné Papp
Levente Buttyán, habil. PhD

December 11, 2019

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Bak Márton László*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2019. december 11.

Bak Márton László
hallgató

Kivonat

A dolgok internete (Internet of Things, IoT) a beágyazott rendszereket összekötő, gyorsan fejlődő technológia, ami az elkövetkező években hozzávetőlegesen több milliárd eszközt fog felölelni. Erre válaszul a támadók új rosszindulatú programcsaládokat fejlesztettek ki, amelyek kifejezetten IoT eszközöket támadnak, mint például a Mirai botnet, az Amnesia rootkit, és más egyéb családok. Az új rosszindulatú programminták hatékony elemzése érdekében fontos, hogy a víruskereső cégek képesek legyenek pontosan besorolni őket ismert családokba. A TrendMicro által a közelmúltban javasolt, TLSH néven ismert hasonlósági lenyomat-készítő eljárásról megmutatták, hogy felülmúlja a többi megközelítést ugyanazon rosszindulatú programcsalád variánsainak detektálásában.

A dolgozatban azt vizsgálom, hogy hogyan lehet a TLSH hasonlósági értékeket felhasználni rosszindulatú programminták csoportosítására, illetve hogy a meglévő klaszterezési algoritmusok képesek-e helyesen osztályozni a hasonló mintákat. A kutatás során 12993 mintát gyűjtöttem a VirusTotalról, melyek mindegyike 29 IoT specifikus rosszindulatú programcsaládból való. Az adathalmazt két, széleskörben elterjedt algoritmussal klaszteresztem: a Partitioning Around Medoids (PAM) csoportba tartozó k-Medoid algoritmussal, illetve a sűrűség-alapú OPTICS algoritmussal. Megállapítottam, hogy a tesztelt klaszterezési algoritmusok feltételezései nem állnak fenn rosszindulatú programcsaládok variánsainak azonosításakor, és a kiszámított klaszterek nem tükrözik pontosan az adathalmazunk szerkezetét. A kutatásom konklúziója, hogy a meglévő klaszterezési algoritmusok nem felelnek meg a gyakorlatban valószínűleg fellelhető rosszindulatú programokból álló mintahalmazok klaszterezésére.

Abstract

The Internet of Things (IoT) is a rapidly growing technology of interconnected embedded devices, estimated to encompass billions of them in the upcoming years. In response, attackers have been developing new malware families specifically targeting IoT devices, such as the Mirai botnet, the Amnesia rootkit and others. In order to efficiently analyze new malware samples, it is important for antivirus companies to accurately classify them as members of known families. To this end, the industry employs a number of techniques including similarity digests, e.g. ssdeep and sdhash. A recently proposed similarity digest by TrendMicro, called TLSH, has been shown to outperform other approaches in the context of detecting variants of the same malware family.

In this paper, I study how TLSH similarity scores can be used for clustering malware samples and whether existing clustering algorithms can correctly classify similar samples. I collected 12993 samples for my study from VirusTotal, each of which is associated with one of 29 IoT specific malware families. I cluster the dataset using two widespread algorithms: the Partitioning Around Medoids (PAM) algorithm k-Medoid, and the density-based algorithm OPTICS. I find that the assumptions of the tested clustering algorithms do not hold in the context of identifying variants of existing malware families, and the calculated clusters do not accurately reflect the underlying structure of the sample set. I conclude that existing clustering algorithms are not appropriate for clustering malware sample sets likely to be encountered in practice.

Chapter 1

Introduction

The concept of malicious software – or malware, as it is called in the computer security community – is almost as old as computers themselves. The theory of self-reproducing automata was originally proposed by János Neumann himself back in the 1960's, and his theory was put in practice just a few years later in 1971 by Bob Thomas, who developed *Creeper*, the first self-replicating computer program[1]. Creeper ran on DEC PDP-10 machines and moved between machines using the ARPANET, the precursor of today's Internet. Creeper did not do any harm, it was developed for experimental purposes only. However, malware created in later years, such as viruses and worms, used similar self-replication and propagation from one machine to another, and besides that, they often carried out malicious activities as well. Interestingly, with Creeper, the first anti-virus program, *Reaper*, was also born: it was developed to search out for copies of Creeper and destroy them.

After writing malware essentially for fun in early years, malware development became a profitable business for miscreants at the end of the last century with the growing number of personal computers and the proliferation of Internet connectivity and Web based services. Later, at the beginning of the millennium, smart phones appeared, and attackers started developing malware for mobile devices too. And today, we are witnessing a new trend: all sorts of embedded devices, including home entertainment systems, industry equipment, vehicles, transport infrastructure devices, and medical equipment are being connected to the Internet, which is rapidly transforming into an *Internet of Things*, or IoT for short. Not surprisingly, malware development followed the new trend, and malware is now developed for embedded IoT devices as well.

A significant problem is that the number of IoT devices is already large and grows exponentially, which means that connected IoT devices can be converted into a substantial

[1] `https://en.wikipedia.org/wiki/Creeper_(program)` (visited: 25 Oct 2019)

attack infrastructure by infecting them with malware and organizing the infected devices into botnets. Actually, such botnets have already appeared in the wild. An infamous example is the *Mirai* botnet, and the importance of the problem is illustrated by the fact that it holds the record for the most intensive DDoS attack in history ever [2]. Of course, malware infected IoT devices can be used not only for building botnets, but also for all sorts of other misdeeds, such as click fraud and bitcoin mining.

On the defense side, some companies, such as Symantec, Kaspersky, and McAfee, develop and sell anti-virus programs, which detect known malware samples based on their characteristic byte patterns (called signatures) or other heuristics. To avoid detection by anti-virus programs, modern malware uses polymorphism and metamorphism, which means that self-replication is combined with mutation that results in new samples that are functionally equivalent but their binary code is different. If done well, then the new variants escape detection by the anti-virus programs. Also, malware developers can add new features to their creatures, resulting in new variants that are still somewhat similar to the previous version of the malware. So anti-virus companies must constantly keep track of the appearance of new malware variants and update their signature databases. As tens of thousands of new malware variants may appear every day, sample analysis, extraction of signatures, and update of the signature database require a huge effort.

Anti-virus companies often rely on malware classification methods to identify relating samples. Classifying malware into malware families makes sense, as members of the same family, while being different at the binary level, exhibit similar behavior. So if the analysts of the anti-virus company have already analyzed samples of a given family, then a new variant from the same family may not need to be analyzed, because its behavior can be assumed to be known already. This greatly reduces the load on the analysts and they can focus on samples that are not similar to any known sample, hence, they probably have completely new features.

In this document, I address the problem of clustering malware samples based on their binary similarity. Such clustering can be used to identify groups of samples that are related, and hence, probably belong to the same malware family. The data set I used to demonstrate my results is restricted to malware samples developed for embedded IoT devices. I decided to focus on IoT malware due to the importance of this new trend. I note, however, that my clustering method is applicable to other types of malware too.

I obtained around 12 000 samples from Virus Total[2] that belong to 9 IoT specific malware families according to their labels produced by different anti-virus tools. After some cleaning (e.g., filtering out packed and encrypted samples), my corpus shrank to around 9 000 samples. I studied the performance of two clustering algorithms, k-medoid and OPTICS, on this data set in terms of resulting cluster sizes, cluster diameters, and the distribution

[2] `https://www.virustotal.com/` (visited: 25 Oct 2019)

of anti-virus labels assigned to samples in each cluster. Both clustering algorithms rely on the ability to measure distances between data points to be clustered. In my case, data points are malware samples, and their distances are measured with the TLSH similarity metric[3], where TLSH is a fuzzy hash algorithm developed by Trend Micro [9]. I found that neither of the two clustering algorithms has acceptable performance: k-medoid produced clusters with unacceptably large diameter, meaning that it put unrelated samples into the same cluster, whereas OPTICS failed to cluster more than half of the samples in the data set. So I developed a new clustering algorithm, which is based on OPTICS, and achieves a performance superior to both k-medoid and OPTICS.

My main contributions are summarized as follows:

- I propose to cluster malware samples based on the TLSH similarity measure.
- I study two distance-based clustering algorithms, k-medoid and OPTICS, and evaluate their clustering performance on a large corpus of IoT malware samples. My results show that neither of them achieves acceptable performance, and hence, cannot be used for clustering malware samples.
- I propose a new clustering algorithm that outperforms both k-medoid and OPTICS, and which is suitable for malware clustering.

The organization of the document is as follows: In Section 2, I provide some background on malware analysis practices, program similarity measures, and clustering. In Section 3, I give an overview on my research methodology, parts of which are expanded in later sections. In particular, Section 4 describes the way I collected the samples and obtained my initial corpus, and Section 5 explains how this initial corpus was cleaned to obtain the corpus I finally used in my evaluation. In Section 6, I introduce the k-medoid and the OPTICS clustering algorithms and evaluate their performance on my corpus of samples. This section also describes how the TLSH similarity threshold is selected. I describe my own clustering algorithm, evaluate its performance, and compare it to that of k-medoid and OPTICS in Section 7. Finally, I conclude my report and sketch some possible future work in Section 8.

[3]I note that the TLSH similarity metric is not really a distance measure in the mathematical sense.

Chapter 2

Background

2.1 Industry Approach to Malware Analysis

In response to polymorphism and metamorphism employed by modern malware, anti-virus companies have began to utilize multi-layered approaches[1][2][3] in order to detect malware. The layers include techniques for metadata-analysis, static analysis, dynamic analysis and machine learning. In order to achieve better protection, companies also deploy network-based techniques, however, these are out of scope for this report.

Metadata-analysis consists of checking a file's reputation, the origin point of download and additional threat analysis reports based on often global threat intelligence networks. Metadata-analysis allows early detection of malware. It often results in blacklists: files on the blacklist are automatically neutralized from analyzed systems.

During static analysis, the instructions and bytes of the analyzed sample are analyzed. Because the sample is not executed, such approaches typically scale better and can provide a quick first glance at the sample. A number of techniques can be used as static analysis, including signature detection, heuristics and program similarity. My study is concerned with the applicability of a specific program similarity technique called TLSH. Thus, I provide more details on program similarity in Section 2.2.

Dynamic analysis of samples requires their execution, which is usually done either by emulation or in a sandbox environment. Such techniques allow companies to perform behavior monitoring and extract precise information about execution. This includes logging which files were modified by the sample, which system calls and in what order were invoked, etc.

[1] `https://www.kaspersky.com/enterprise-security/wiki-section/home` (visited: 25 Oct 2019)
[2] `https://www.eset.com/int/about/technology/` (visited: 25 Oct 2019)
[3] `https://www.symantec.com/products/atp-content-malware-analysis` (visited: 25 Oct 2019)

Extracted information from both static and dynamic analyses can be used as features for machine learning. Using machine learning, companies aim to extract models of malicious behavior based on large data sets of benign and malicious samples. The expected benefit of machine learning in the future is a generalized model of malware which is capable of detecting previously unseen malicious samples. Another benefit of learning algorithms is the ability to classify the large number of samples anti-virus companies need to handle daily. In this report, I study the applicability of two clustering algorithms, k-medoid and OPTICS, whose details are presented in Section 2.3.

The classification of previously encountered samples all collected in so called malware families. Families are constructed such that members of the same family share common features, e.g. exploit techniques, communication patterns and protocols, malicious activities and required technology on the victim's machine. Malware families tend to specialize for specific use-cases and technologies, some of them targeting the IoT ecosystem specifically. Such families include trojan horses, backdoors and remotely controlled botnets.

Accurately categorizing samples into families is challenging. Implementations of separate families may share code segments, e.g. after Mirai's source code was leaked[4], it was reused in other families such as Hide N' Seek[5]. Families also slightly change their features as new versions are released. The different versions are often referred to as variants. For example, Mirai has several improved versions of its code, including Satori, Okiru and Masuta [6].

2.2 Program Similarity

The goal of program similarity is to compare programs and find similar instances. There are several approaches to this problem, including locality sensitive hashes and similarity digests. Locality sensitive hashing (LSH) algorithms, including context triggered piecewise hashing algorithms, have the property that a small change to the file being hashed results in a small change to the hash [9]. As a result, comparing hash values reveal similar files. `ssdeep` is a context triggered piece-wise hashing algorithm (CTPH), which is the de facto approach in industry, used by companies such as VirusTotal, VirusShare and Malwr. It generates string hashes roughly up to 100 bytes that are concatenations of 6-bit piecewise hashes. The hash value then can be compared with other hashes to measure how many character operations are necessary to transform one string into the other. Because of the bounded-size hash it produces, it quickly loses granularity and only works for relatively small files of similar sizes.

[4] `https://securityaffairs.co/wordpress/51868/cyber-crime/mirai-botnet-source-code.html` (visited: 25 Oct 2019)

[5] `https://www.fortinet.com/blog/threat-research/searching-for-the-reuse-of-mirai-code--hide--n-seek-bot.html` (visited: 25 Oct 2019)

Similarity digests attempt to solve the nearest neighbor problem using a digest that is superficially similar to a cryptographic hash. `sdhash` [12], another popular similarity digest tool, is slower then `ssdeep`, however, it overcomes its main limitation of being sensitive to byte ordering. However, the scoring method used by `sdhash` results in the undesirable property that similarities are not transitive. The authors treat any score in the range of $[21, 100]$ as "strong" in terms of correlation, which does not provide enough flexibility [11].

TLSH [9] stands for Trend Micro Locality Sensitive Hash, bearing the name of the developing company. Since its release in 2013 it hasn't received much attention, probably due to its static approach which is considered ineffective in malware classification. However, recent research [10, 15] and have showed that TLSH is not only more precise than previous methods, including `ssdeep` and `sdhash`, it is also applicable for malware classification.

The TLSH digest of an input byte string, e.g. malware sample, is calculated in the following four steps.

1. The byte string is processed in a 5-byte-long sliding window and counters associated with byte triplets are incremented as the triplets are encountered in the input.
2. Based on the value of counters, quartile points are calculated such that counter values are separated into four equal regions.
3. The 3-byte-long digest header is constructed. The first byte is a checksum of the byte string. The second byte represents the logarithmic length of the byte string (modulo 256). The third byte is derived from the quartile points.
4. The remainder of the digest is computed based on the counter values.

The result is a 70-byte-long digest. The algorithm uses the sliding window in order to capture the correlation between neighboring instructions. Such neighboring instructions designate the program's functionality. In order to calculate similarity between files, TLSH differences have to be computed. TLSH differences are in the range of $[0, \sim 1100]$, with 0 indicating identical files.

TLSH works well as long as the input binary is not packed. Packed executables contain only a small portion of executable code, most of the files' contents are filled with high entropy data. These highly dissimilar portions decrease the accuracy of TLSH, resulting in large differences even for files with the same origin.

2.3 Clustering

Clustering is a machine learning problem with the goal of grouping observations together such that members of a group are similar to each other, while also being different from members of other groups. Such algorithms can be both supervised and unsupervised. In this document, my goal is to cluster malware samples based on their TLSH differences such that clusters represent variants of malware families. In this scenario, I can measure similarity but I have no information about the number and nature of variants, i.e. I do not know the correct labeling of the input data points. As a result, I use unsupervised learning which does not require labeling of samples before analysis. I evaluate the performance of two widely-used algorithms: k-medoid and OPTICS.

K-medoid [5] is a PAM-based algorithm in which clusters can have only valid data points as their centers (also called medoids). The algorithm has one input parameter, k, which determines how many clusters will be present in the output of the algorithm. The algorithm first selects k medoids, then it tries to fit all data points to the nearest cluster head. Medoid selection and re-clustering is repeated iteratively until an optimum is reached. The measure of goodness for the algorithm is $s(k)$. It is calculated for every data points, denoted $s(i)$, measuring the gain in assigning the data point to a specific cluster based on distance. In order to compute $s(k)$, individual $s(i)$ values are summed up. Singleton clusters (clusters with only one data point) are punished by reducing their $s(i)$ value to 0. In order to find the optimal clustering setup, $s(k)$ has to be maximized.

OPTICS [1] is a density-based algorithm capable of identifying dense and sparse regions in the input data set. It can also cluster data points using the the extracted structural information. It takes two parameters, ϵ_{max} and $minPts$. ϵ describes the radius of an area which contains at least $minPts$ number of samples. The algorithm dynamically calculates ϵ values for data points such that data points have at least $minPts - 1$ samples in their ϵ radius. This output is referred at as the reachability distribution. If no initial ϵ_{max} value is set, the algorithm uses ∞ as its upper bound. OPTICS also have a built-in clustering algorithm, ξ, which clusters data points by detecting abrupt changes in the reachability distribution.

Chapter 3

Methodology

The high-level overview of the methodology I followed during this research is shown in Figure 3.1. The methodology can be divided into three main steps:

1. data collection, which results in a data set of IoT malware samples,
2. filtering, which removes packed and/or encrypted samples from the data set, and
3. clustering, which identifies variants in the data set by grouping malware samples based on their pair-wise TLSH differences.

In order to acquire a data set of IoT malware samples, I first need to select a specific IoT platform which the data set should target. This is a required step as the different instruction sets could cause TLSH to measure big differences between variants compiled for different platforms. For this study, I selected samples targeting the ARM platform, since it is widespread use in the IoT world. Secondly, I compile a list of malware family names based on previous studies of the IoT malware landscape [4, 3, 16]. I use the compiled list to search for and download malware samples from VirusTotal[1], a publicly available site to which users can upload executables and submit URLs. The site scans uploaded executables with a number of anti-virus tools and returns to the user the collected results, including the malware family names under which tools detected the executable. In addition, the site can perform more in-depth analysis of samples, e.g. extracting information from the files' headers. For a subscription fee, users can also download samples from VirusTotal's database. I download not only relevant samples but the corresponding anti-virus scan results as well. The scan results are fed to AVClass [14], which outputs the most likely malware family name. Throughout my study, I use AVClass's output as the ground truth for all samples.

[1] `https://www.virustotal.com` (visited: 25 Oct 2019)

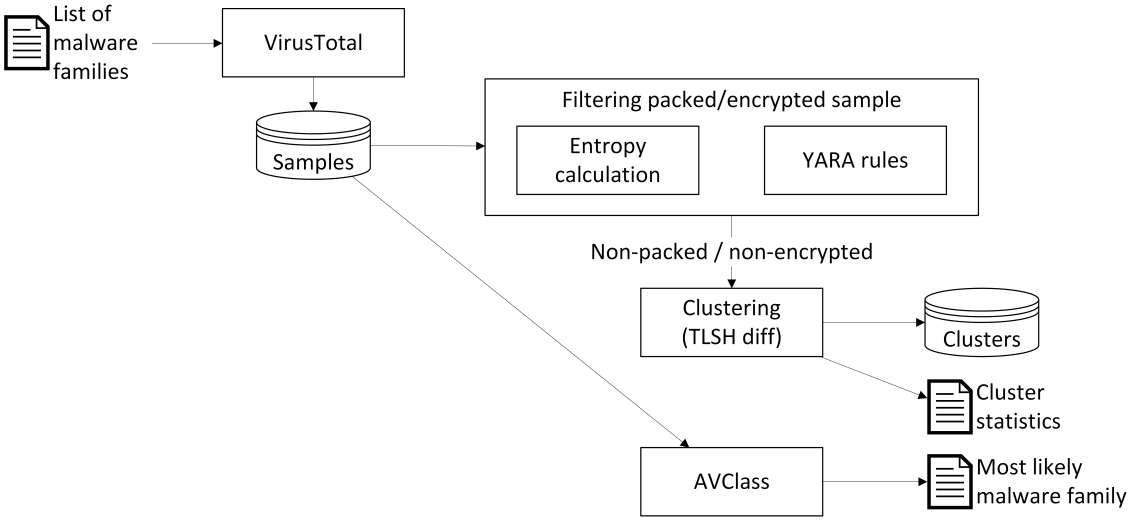

Figure 3.1: Overview of Methodology

The second step in my methodology is to filter the downloaded samples. The step is required because calculating TLSH differences is a static analysis technique and as such, it cannot efficiently work with packed and/or encrypted samples. I use two approaches for filtering my data set. Firstly, I use binary entropy calculation [7], which calculates the empirical entropy of a file based on the contained bytes. There exist best practices telling which calculated entropy values signal packed executables. Secondly, I use YARA rules[2], a technique commonly employed in malware research. YARA rules allow human analysts to describe various strings and sequences of bytes in the binary, whose presence signal the fulfillment of a semantic criterion. For example, YARA-rules can be written to detect whether an executable was packed with a specific packer as certain packers leave traces in the binary, e.g. their names.

The final step in methodology is clustering. My goal is to group samples based on their TLSH difference, thereby detecting variants of malware families. Initially, I cluster the data set with two widespread algorithms, k-medoid and OPTICS. However, the resulting clusters prove to be difficult to interpret which can be attributed to a mismatch between the algorithms' assumptions and the setting. As a result, I develop a new clustering algorithm, which I describe in Chapter 7.

[2] `https://yara.readthedocs.io/en/latest/` (visited: 25 Oct 2019)

Chapter 4

Data Collection

As discussed in Chapter 3, the first step of my methodology is data collection. My goal is to acquire a data set consisting of malware samples from IoT-specific malware families. To this end, I first reviewed existing literature [4, 3, 16] for relevant malware family names and compiled a list of 29 names. Afterwards, I queried VirusTotal in order to find and download malware samples as well as their anti-virus scan results. By the end of this phase, my corpus consisted of 12 993 samples.

4.1 Malware Families in Scope

I compiled a list of 29 malware family names based on existing literature [4, 3, 16], which is shown in Table 4.1. These malware families specifically target the IoT ecosystem. Many of them implement the ability to infect other machines and connect them to an existing botnet. The botnet is remotely administrated by the attacker via various channels, e.g. IRC or HTTP-based communication. Samples from these families take various commands

hydra	psybot	chucknorris
dofloo (spike, mrblack)	gafgyt (bashlite, lizkebab)	elknot (billgates)
themoon	pnscan	persirai
remaiten (ktnrm, routrem)	newaidra (irctelnet)	mirai (satori, okiru, masuta, puremasuta)
ballpit (lizardstresser)	ddostf	chinaz
aidra (lightaidra, zendran)	muhstik	mayday
darrloz (zollard)	luabot	jenx (jennifer)
znaich	bossabot	amnesia
zorro	ddoser	tsunami (kaiten)
xorddos (xarcen)		

Table 4.1: List of In Scope Malware Families

from the attacker via a command & control server, for example, the kind of attack to carry out (denial of service attacks, remote code execution, etc.), configuration options related to attacks and other management options. As I discussed previously, families also share similar traits as they are known to copy features from each other.

4.2 VirusTotal Searches

In order to acquire samples belonging to any of the previously listed 29 malware families, I queried VirusTotal through its API service. The API allows developers and researches to automate the search and download process of required samples. VirusTotal provides two kinds of API, a private and a public one. While many of their endpoints and features are freely available to registered users, some of them are restricted to premium customers only[1]. The public API allows users to download anti-virus scan results of specific samples. The private API, on the other hand, provides additional features. Users can query samples based on various search criteria, e.g. source metadata, and it allows users to download both the samples and their extended report files, including all metadata.

I implemented Python scripts to query the private API based on the following search criteria. Samples must be detected by at least one engine as a family present on my list; they must have the ELF format and must be written for the ARM platform. I then compiled a list of the returned SHA256 hashes and downloaded the samples as well as their extended reports. The search returned 11 957 samples, however, out of the 29 families, only 9 were present.

I repeated the query a few days later with slightly modified search criteria: instead of allowing any engine to detect my samples, I focused solely on Kaspersky's and Symantec's engines. My expectation was that the returned list by these queries would yield a subset of the original query, however, this wasn't the case. The query for Kaspersky's engine returned 11 222 samples, but only contained samples out of 5 families. The query for Symantec's engine returned 5 804 samples from 3 families.

4.3 Corpus

The result of all three of my queries is shown in Figure 4.1. The total number of unique hashes returned by VirusTotal was 12 993, of which 5 295 was returned in all three queries. There were unique hashes in all three queries, 624 and 23 samples by Kaspersky and Symantec, respectively. When no specific engine was specified, VirusTotal produced 1651

[1] `https://developers.virustotal.com/reference#getting-started` (visited: 25 Oct 2019)

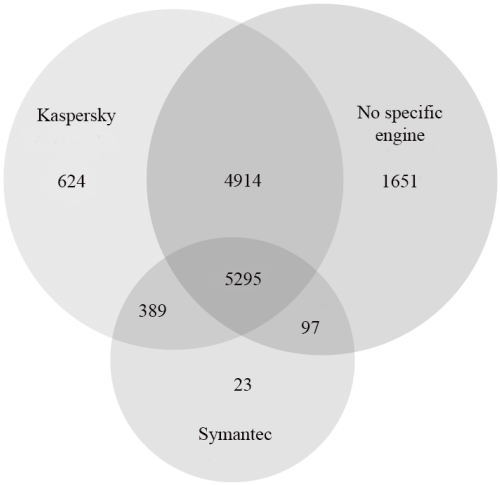

Figure 4.1: VirusTotal query results

unique hashes. All the other hashes in my data set were returned by more then one queries. A likely explanation for this is that VirusTotal's database is working with a sliding window, that only indexes a limited set of samples at a time. However there is no certainty in this, because the inner workings of the API are not well documented.

To extract the ground truth for the acquired samples, I used AVClass [13], an open-source malware labeling tool. AVClass takes as input the anti-virus scan report file(s), as well as aliases for detected malware and a generic token list for label stripping purposes. The tool outputs the malware family reported by the majority of anti-virus tools that detect the sample as malicious. In the alias file, I provided the tool my list shown in Table 4.1 with aliases specified in parenthesis to any given family. However, I needed to make changes to the tool's source code as initially, it could not provide a label for a number of samples. In order for AVClass to cast a majority vote, it needs at least 4 detections per sample. As some of my samples had lower detection rates, I removed this requirement.

Chapter 5

Filtering The Corpus

Before clustering the data set, I needed to filter packed and/or encrypted samples as TLSH differences for these type of samples are meaningless. I first used binary entropy calculation, which calculates the empirical entropy of a sample based on the bytes contained in the binary. There exist best practices on what values signal packed and/or encrypted executables. I also used YARA-rules, a widely used pattern matching approach in malware analysis, to statically look for leftover traces of packers.

5.1 Binary Entropy-based Filtering

Binary entropy calculation [7] detects packed and/or encrypted binaries by calculating the empirical entropy of their bytes. The algorithm operates with a 256-byte sliding window. For every window, the algorithm calculates the byte entropy using the Shannon formula ($H(x) = -\sum_{i=1}^{n} p(i) \log_2 p(i)$). Executables generally contain many blocks of zero-value data bytes in order to pad or align code sections. Thus, the algorithm does not take into consideration blocks with more than 128 zero bytes. Empirical best practices for entropy values are shown in Table 5.1.

The measured empirical entropy values of my data set is shown in Figure 5.1. Taking the previously discussed best practices into consideration, there's a clear cut between a set of

Data type	**Average entropy**
Plain text	4.347
Native executable	5.099
Packed executable	6.801
Encrypted executable	7.175

Table 5.1: Empirical entropy values

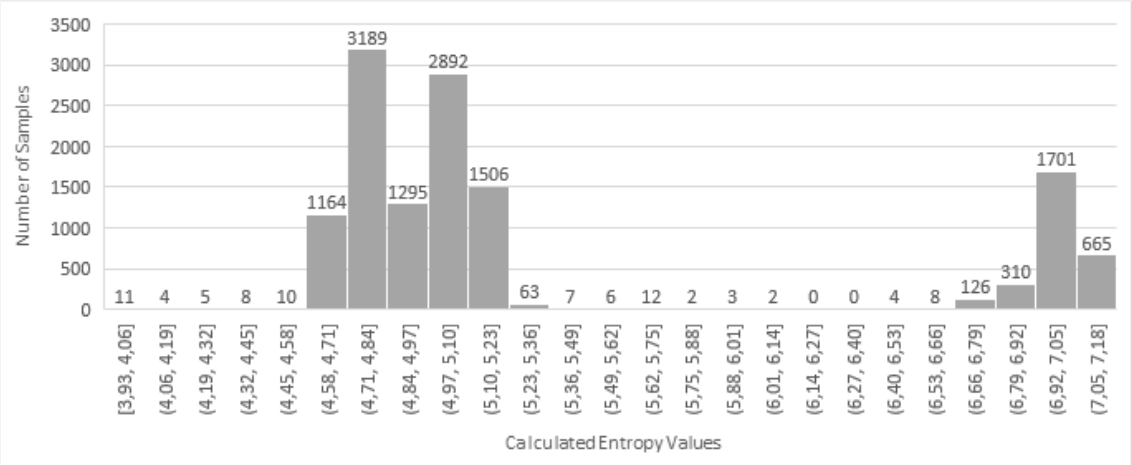

Figure 5.1: Entropy distribution of the dataset

native executables and a set of packed and/or encrypted samples. As a result, I excluded 2817 samples from the data set.

5.2 YARA-rules

Binary entropy calculation has one major limitation, namely, that large sections of low-entropy bytes can lower the calculated overall entropy. In order to remove this limitation from the filtering process, I also used YARA-rules. YARA is a tool mainly used in malware research to identify and classify malware samples[1] based on patterns. Packers can leave traces in the binary, e.g. specific strings and/or byte sequences unique to the packer. Byte sequences can be coded into YARA-rules and detected statically.

A YARA-rule has three main sections: metadata, strings and conditions. The meta section can be used to store key-value pairs, metadata and other sorts of information. The strings section is used to define variables, strings, for which analysis should search for. Three types of strings can be used: ASCII-text, hexadecimal strings and regular expressions. In the third section, condition, boolean expressions can be defined using strings from the previous section. If the condition is satisfied by the analyzed sample, the sample is marked.

```
rule DummyRule{
  meta:
    name = "Dummy Rule"
    purpose = "Example"
  strings:
    $var1 = {1A 2B 3C}
    $var2 = "Random string"
  condition:
    $var1 in (0..100) and $var2 in (0..1024)
}
```

[1] https://yara.readthedocs.io/en/latest/ (visited: 25 Oct 2019)

Listing 5.1: YARA example code

In Listing 5.1, I demonstrate a very basic YARA-rule. I define two variables, `$var1` and `$var2`, a hexadecimal string, and a text string. The condition section specifies that if `$var1` is found in the offset range $[0, 100]$ and `$var2` is found in the offset range $[0, 1024]$, the input file must be marked.

I ran YARA-rules for UPX and other packers on the whole data set, looking for packed binaries. I found a total of 980 packed samples, all of which were packed with UPX. However, these samples have already been filtered using binary entropy calculation.

5.3 Filtered Corpus

6108	mirai
3711	gafgyt
163	dofloo
92	tsunami
63	ddostf
19	presenoker[2]
5	dnsamp[3]
2	oneeva
2	ditertag
1	zergrush
1	luabot[4]
1	lightaidra
1	cloxer
1	SINGLETON:a15990a6650a7290042356d40350acc2799ef3b42be84b25d739cf2662c568b3
1	SINGLETON:99337f0add529b4e9e433175cc05e03b62b079edb45e501842710cbde13466ac
5	unclassified

Table 5.2: Distribution of Malware Families in the Filtered Corpus

The distribution of malware families in the filtered data set is shown in Table 5.2. The malware family names are the outputs of the AVClass tool and include names which were not part of my original list. This is due to the process by which I collected samples: I required at least one engine to detect a given sample as member of a relevant malware family. By contrast, AVClass took into consideration all labels and cast a majority vote on the malware family name. SINGLETON denotes samples whose malware family could not be determined. There were 5 samples for which AVClass was unable to produce a malware family name.

Chapter 6

Clustering

My goal is to group similar malware samples based on their pair-wise TLSH difference in order to detect variants of malware families. As a first step, I need to determine the maximum TLSH difference such that two malware samples are of the same variant. I then apply two widespread clustering algorithms to the data set: k-medoid [5] and OPTICS [1]. The results of each clustering algorithm are detailed in Sections 6.1 and 6.2, respectively.

In a previous study regarding TLSH *Tamás*[15] suggested a threshold of 70 to be used for few false positives. His suggestion was based on a relatively small set of 477 samples with most of the files being from two families. In order to define a globally applicable TLSH threshold for malware similarity, a measurement was made on a much bigger scale. It was found, the optimal threshold for detecting variants is 48.

6.1 K-Medoid

K-medoid is an algorithm with the goal of finding k representative objects among the objects of the data set [5]. There are several rationales behind choosing this particular algorithm. It is an unsupervised learning algorithm, meaning that there is no need to supply any additional data, only the similarity measurements between samples. Another reason is that the algorithm only selects existing data points as cluster heads. This is useful in my scenario, because analyzing cluster heads could give us information about samples in the same cluster. The disadvantage of this algorithms is that I have to specify the input k for the algorithm. Unfortunately, I did not know how many variants there were in the data set, therefore, I calculated cluster configurations for all potential k values. For my experiments, I used the k-medoid implementation from the Pyclustering library [8].

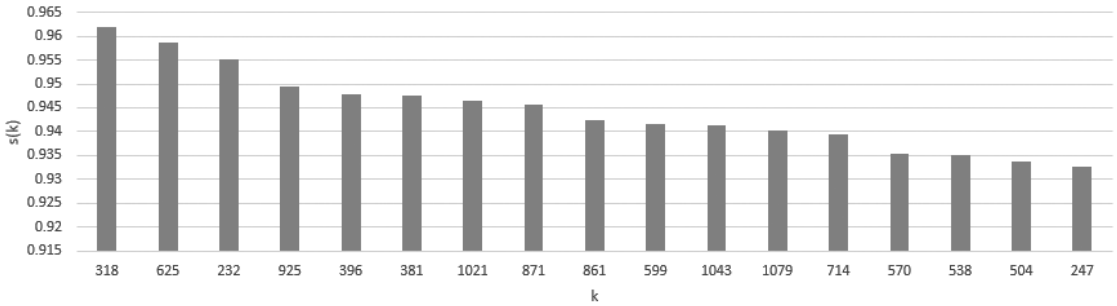

Figure 6.1: Best $s(k)$ values of the dataset

s(k)	0.4054158410234883
Number of clusters	17
Maximum diameter	1038
Minimum diameter	180
Mean diameter	467.823529411
Largest cluster size	1110
Smallest cluster size	35
Mean cluster size	573.294118

Table 6.1: Statistics for k = 17

The challenge I faced after calculating the cluster configurations was to choose between them. There exists a metric, $s(k)$[5], which can be used to select the best k value. The metric captures how well the clusters group objects by calculating how the diameters of clusters would change if objects were clustered differently. The closer the metric is to 1, the better the setup. I computed the $s(k)$ metric for all cluster configurations in order to rank my setups.

As shown in Figure 6.1, the best $s(k)$ values of the data set barely differ, however, the corresponding k values have a wide range. This makes it unclear which setup to choose. In order to overcome this challenge, I extended my requirements based on previously discovered TLSH threshold. Variants have a TLSH difference lower then 48, as a result, cluster heads of different clusters should have a TLSH difference score higher than 48. With this requirement in mind, I looked at the cluster configurations and found, that with TLSH thresholds ranging from 30 to 70, $k = 17$ has the achieves $s(k)$ value.

Statistics of about the cluster configuration $k = 17$ is shown in Table 6.1. In this configuration, the calculated cluster diameters range from 180 to 1038, the mean being 468. In my scenario, cluster diameter can be interpreted at the largest TLSH differences between any two samples in the same cluster. Cluster sizes in the setup range from 35 to 1110 samples.

There are a number of issues with the $k = 17$ setup. Firstly, the threshold for variants was determined to be 48. None of the clusters in this setup come close to this TLSH

difference, the lowest being 180. Thus, I can conclude that clusters should be split into smaller clusters. Secondly, I had families with only a small number of samples in the data set. By contrast, the setup did not have any singletons or small clusters. Therefore, I conclude, that the k-medoid algorithm does not perform well for the goal of clustering malware samples based on TLSH differences.

6.2 OPTICS

The second algorithm I tested is the density-based algorithm, OPTICS [1]. The algorithm is able to detect dense and sparse regions in the data set. This characteristic makes it favorable in my scenario, since I have families with only a few samples as well as families with thousands of samples, as shown in Table 4.1. The algorithm can also adapt in clustering size, i.e. small and large clusters can both be found.

As discussed before, the algorithm takes two additional parameters besides a precomputed distance matrix: $minPts$ and ϵ_{max}. I can specify an upper bound for ϵ_{max} as the maximum TLSH difference in the data set. However, selecting $minPts$ is a challenge without knowledge about the internal structure of the data set. To gain this knowledge, I ran OPTICS with different parameter setups: ϵ_{max} values were set to be 40, 50, 60 and 70, while $minPts$ was set to be 1, 2, 5, 10, 20, 40, 50, 70, 100, 150 and 200.

The resulting cluster configurations were again unsatisfactory. In all configurations, the number of unclassified samples was very high. The least number of unclassified samples was achieved when $minPts$ was set to low values. Different values of ϵ_{max} did not affect this trait: setting $minPts$ to 2, $\epsilon_{max} = 40$ yielded 1800 unclassified samples, while $\epsilon_{max} = 70$ resulted in 1721 unclassified samples. The more I increased $minPts$, the more unclassified samples were returned. The configuration $\epsilon_{max} = 70$, $minPts = 200$ resulted in 6934 unclassified samples, which is 68% of the data set.

Choosing lower $minPts$ values produced more clusters, with less members in each cluster and with a smaller diameter for each cluster. By contrast, higher $minPts$ values produced fewer clusters with more samples and higher diameters. For example, the setup with $\epsilon_{max} = 50$, $minPts = 50$ had 43 clusters. The maximum diameter was 191, the minimum was 6 and the mean was 83. The smallest cluster contained 51 samples in it, the largest 192. After increasing $minPts$ to 150, the number of clusters decreased to 12, the maximum diameter changed to 176, minimum being 2. However the mean diameter decreased to 49. The cluster sizes also increased, the smallest cluster containing 171 samples, while the largest had 440 samples in it. To summarize, clusters generated with different parameters were quite similar, however, all setups resulted in a large number of unclassified samples.

Chapter 7

New Clustering Algorithm

7.1 Clustering Lessons Learned

The results of both tested algorithms present issues for malware analysis. Firstly, the k-medoid algorithm produces clusters whose diameters are too large to represent variants of malware families. I determined that the threshold TLSH difference for variants is 48, however, k-medoid's diameters ranged between $[180, 1038]$. OPTICS's cluster diameters were more in line with my threshold value, however, as much as 68% of the samples were detected as outliers.

The main limitation of these algorithms when used with TLSH differences is their assumptions about the distance matrix. Namely, they assume values to be mathematical distances. TLSH differences, however, are not distances in the mathematical sense as the triangle inequality does not hold for computed values. Sample a being similar to sample b and sample b being similar to sample c does not imply that samples a and c are similar.

In addition, these algorithms were originally developed to cluster measurements which may be noisy. In order to remove noise, the data set must be cleaned and for optimal clustering results, it must also be balanced. A balanced data set in my case requires exclusion of samples from families with very high and very low sample counts. Such a step, however, is undesirable during malware analysis as potential outliers may represent previously unseen variants or even entire families.

In summary, my experiments have shown that existing clustering algorithms are inadequate to cluster malware samples based on their TLSH differences. What is more, their innate assumptions limit their use when the data set is unbalanced. As a result, new clustering algorithms are needed which take into consideration the previously discussed specialties.

7.2 New Clustering Approach

My goal is to create a new clustering algorithm which meets the following requirements. Firstly, it has to cluster samples based on their binary similarity expressed as TLSH differences. Secondly, it has to be able to find even the smallest clusters in a varying density data set. The input data set may contain singleton clusters, i.e. samples dissimilar to every other sample, however, these must not be treated as noise because these are the most interesting samples for malware analysis.

My developed algorithm is based on OPTICS, however, I replaced its default clustering algorithm, ξ. My algorithm can be divided into three major phases, as shown in Listing 7.1. In the first phase, I extract information about the structure of the data set. In the second phase, I generate a greedy, initial cluster configuration based on TLSH differences. In the last phase, I merge clusters in order to compensate for the greedy mechanism in the previous phase.

```
structureData = getDatasetStructureData(dataset)
clusters = calculateInitialClusters(structureData)
return joinClusters(clusters)
```

Listing 7.1: High-level Overview of Proposed Algorithm

In order to extract structural information from the data set, I reuse OPTICS with input parameters $minPts$ and ϵ_{max}. OPTICS can compute the reachability distribution of the data set which gives the ϵ values required to form a cluster around individual samples. Samples with low ϵ values represent dense regions while samples with high ϵ values represent sparse regions. I sort the resulting reachability distribution such that samples in dense regions come first in the list. The overview of this phase is shown in Listing 7.2.

```
reachabilityDistribution = list(OPTICS(minPts, ϵ))
structureData = list ( (sample_id, reachabilityDistribution[sample_id]) )
sort sturctureData based on reachability[sample_id]
```

Listing 7.2: Extraction of Structural Data based on OPTICS

My initial clustering is greedy in the sense that it organizes the densest regions into cluster first. I begin with no clusters and choose the first sample in the sorted reachability distribution. This sample represents the densest region of the data set and it becomes the first cluster head. I then put all samples into the selected head's cluster which are considered similar enough, captured by the parameter $maxIntraDissimilarity$. The selected head and the grouped samples are then removed from the reachability distribution. Subsequent cluster heads are selected such that they have the lowest corresponding ϵ value and they are significantly dissimilar to previously selected cluster heads. There may be cases when new cluster heads cannot be selected this way. In such cases, I select the sample with the

maximum dissimilarity to every other cluster head. I repeat this process until all samples are clustered. The pseudo code is shown in Listing 7.3.

```
selected_cluster_heads = [ ]
clusters = [ ]
cluster_head = structureData[0]
DO
  new_cluster = [ cluster_head ]
  # append every item to cluster, which are similar enough to the head
  for each item in structureData:
     new_cluster.append(item if difference to cluster_head < maxIntraDissimilarity)

  # append new cluster to array of clusters
  clusters.append(new_cluster)

  # exclude clustered samples
  for all sample in new_cluster:
    structureData.remove(sample)

  # select the head of the new cluster
  for all sample in structureData:
    # check if threshold dissimilarity rule can be used to select new head
    cluster_head = sample if dissimilarity to all other heads > headDissimilarity
    if sample not selected:
      # try maximum dissimilarity rule
    cluster_head = sample if dissimilarity to all other heads is maximum
    # retry with another sample if both rules fail
    if sample not selected:
      continue
    else:
      cluster_head = sample
      selected_cluster_heads.append(sample)

REPEAT until structureData is empty
```

Listing 7.3: Initial Greedy Clustering Approach

The data set may contain dense regions whose diameter is larger than the maximum allowed dissimilarity. In such cases, the initial clustering strategy faces a limitation as it groups the center of the region into one large cluster and generates several small clusters on its perimeter. In order to overcome this challenge, I try to detect such perimeter clusters and merge them with the center cluster. This process is shown in Listing 7.4. I combine two clusters if the combined cluster's diameter either remains under 48 (my empirical threshold for variants), or merging increases the diameter of the center cluster by a fixed parameter.

```
# function to decide if I can combine two clusters
function can_combine(c1, c2){
  # save the original diameter of the larger cluster
  originalDiameter = diameter(c1)
  if diameter(c2) > originalDiameter:
    originalDiameter = diameter(c2)
  # merge clusters
  cJoined = list(join c1 with c2)
  # return result of applying merge rules
```

```
  return diameter(cJoined) < 48 or originalDiameter * fixedValue
}

DO
  for each c1 in clusters:
  for each c2 in clusters:
    if c1 == c2:
      continue
    # if I can combine the two clusters, make them one, and delete the two originals
    if can_combine(c1, c2):
      new_cluster = c1 + c2
      clusters.remove(c1)
      clusters.remove(c2)
      clusters.append(new_cluster)
REPEAT until no clusters can be joined
```

Listing 7.4: Merging Clusters

7.3 Evaluation

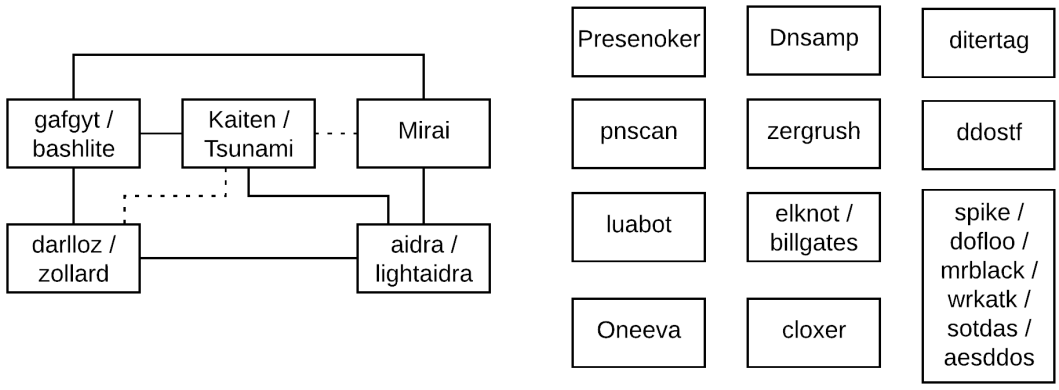

Figure 7.1: Relationships Between Families in My Dataset

In order to evaluate the efficiency of my proposed clustering algorithm, I compared it against the results of both k-medoid and OPTICS. During evaluation, I took into consideration cluster diameters, the number of singleton clusters generated and two new measures of goodness. The first measure of goodness shows how "pure" a cluster is in a strict sense, i.e. how many samples of the cluster is of the family with the most samples in that cluster. I note, that my experiments showed the untrustworthiness of malware family labels. However, at the time of writing, no better alternative is available as ground truth. I also need to take into consideration that malware families share and/or copy features from each other which results in shared features. In response, the relaxation of my measure of goodness considers not only the family with the most samples in a given cluster but also families with which it is known to share features. The feature relationships

	k-medoid	OPTICS	My algorithm
Number of clusters	17	13	745
Number of singletons	0	6058	353

Table 7.1: Comparison of the clustering methods

are shown in Figure 7.1. Straight lines represent immediate relations, dotted lines show indirect relationships.

The number of generated clusters and singleton clusters are shown in Table 7.1. K-medoid and OPTICS both generated considerably less clusters than my algorithm. Their numbers are more in line with the number of malware families my data set contains based on the output of AVClass. My algorithm generated 745 clusters of which 353 were singletons, a negligible amount compared to OPTICS's original clustering algorithm.

The diameters of cluster configurations from all three algorithms are shown in Figure 7.2. As my clustering relied on TLSH differences, cluster diameters can be interpreted as the maximum TLSH differences in each cluster. My experiments have shown that in order to detect variants of malware families, cluster diameters must be below 48. The figure shows that both k-medoid's and OPTICS's cluster diameters are too large to denote variants. The cluster configuration of my algorithm, however, are much closer to this threshold with 93.69% of my clusters having diameters below 50. As a result, my clusters are more likely to represent malware variants.

Figure 7.3 shows the algorithms' performance with respect to my strict measure of goodness: what the ratio of the most populated malware family in a cluster is to other families in the same cluster. This metric can only be computer for non-singleton clusters as clusters containing only 1 sample automatically achieve 100% ratio. As seen on the figure, cluster configurations of both k-medoid and OPTICS typically achieve ratios between 0.56 and 0.63. By contrast, of the 392 non-singleton cluster produced by my algorithm, 185 have ratios over 0.6 of which 103 outperform both OPTICS and k-medoid.

The relaxed measure of goodness ratios achieved by all cluster configurations are shown in Figure 7.4. In this setting, singleton clusters can be included as well, since non-singleton clusters have a chance of achieving 100% ratio due to the extensive relationships between malware families. The figure shows that both OPTICS and k-medoid achieve ratios above 0.95. However, my algorithms produce clusters, whose relaxed ratios are well below those achieved by k-medoid and OPTICS. While investigating this issue, I found another indication of poor anti-virus labels. Specifically, all clusters achieving ratios of 0.5 contained 2 samples and their malware family labels do not match. However, the diameters of these clusters were quite low, the mean diameter being 30.71. Checking the samples' VirusTotal pages, I also saw that there were only a few labels on their scan pages. As a result, the low ratios could be the result of misclassifications.

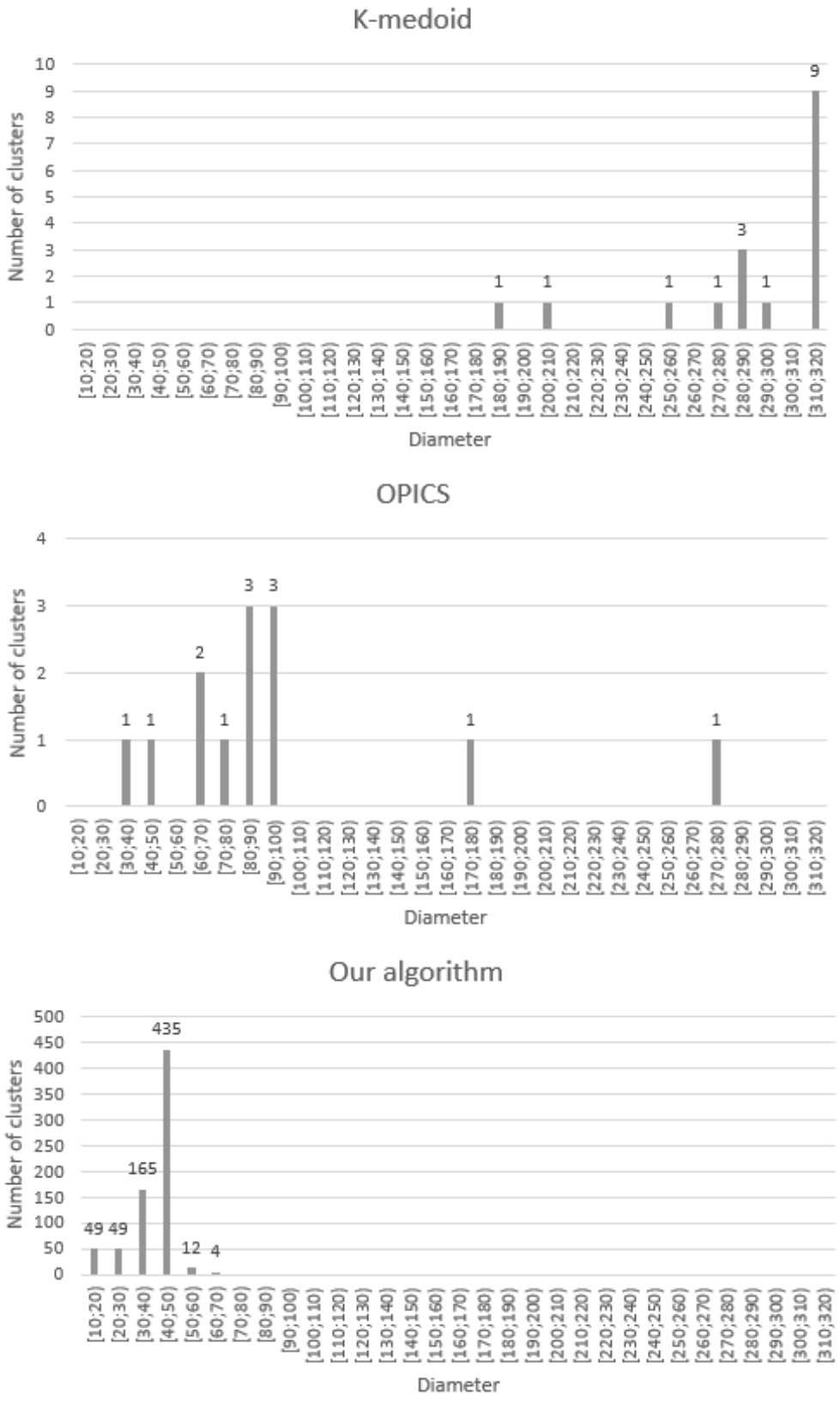

Figure 7.2: Diameters of All Cluster Configurations

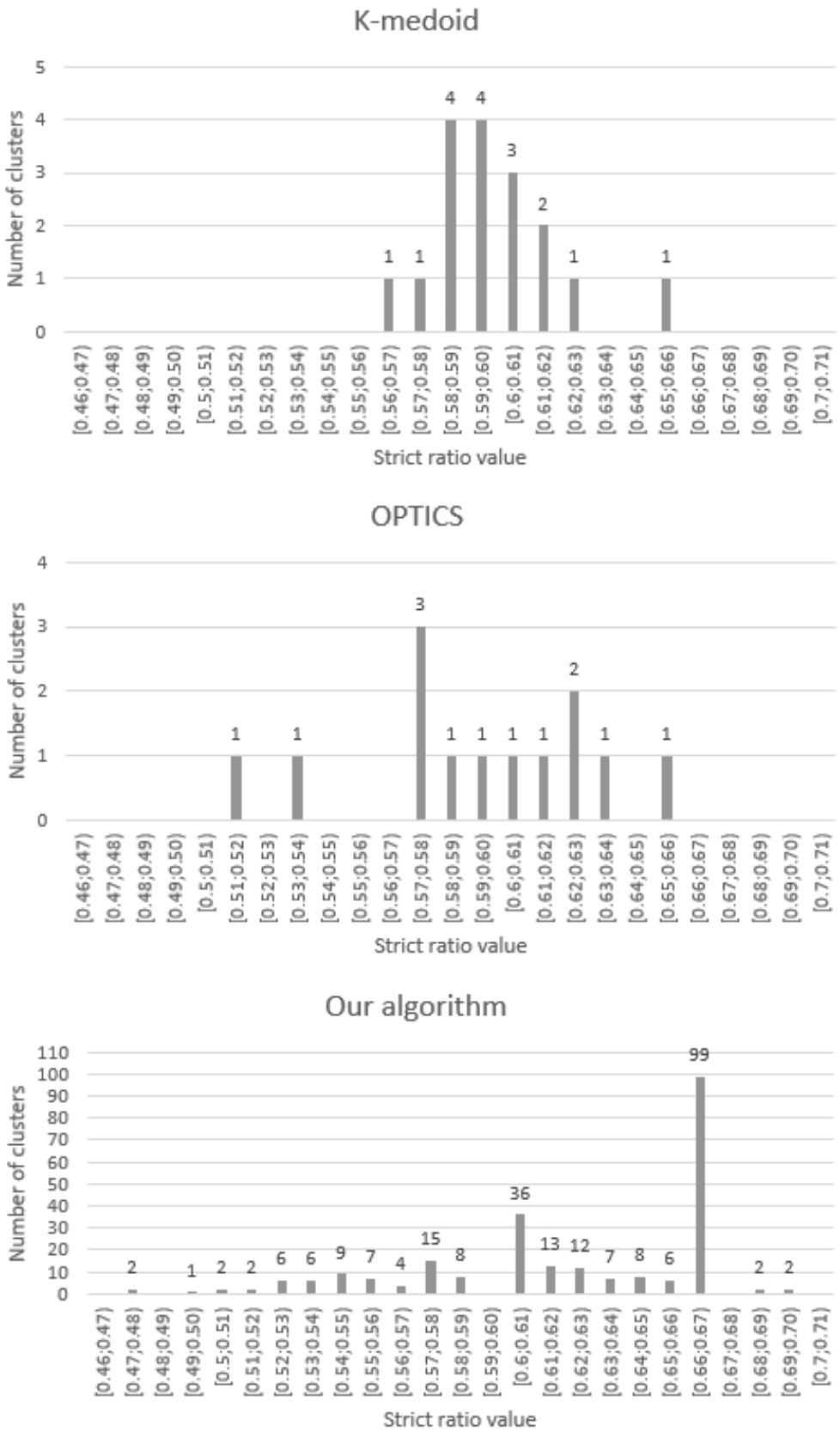

Figure 7.3: Strict Goodness Ratios of All Cluster Configurations

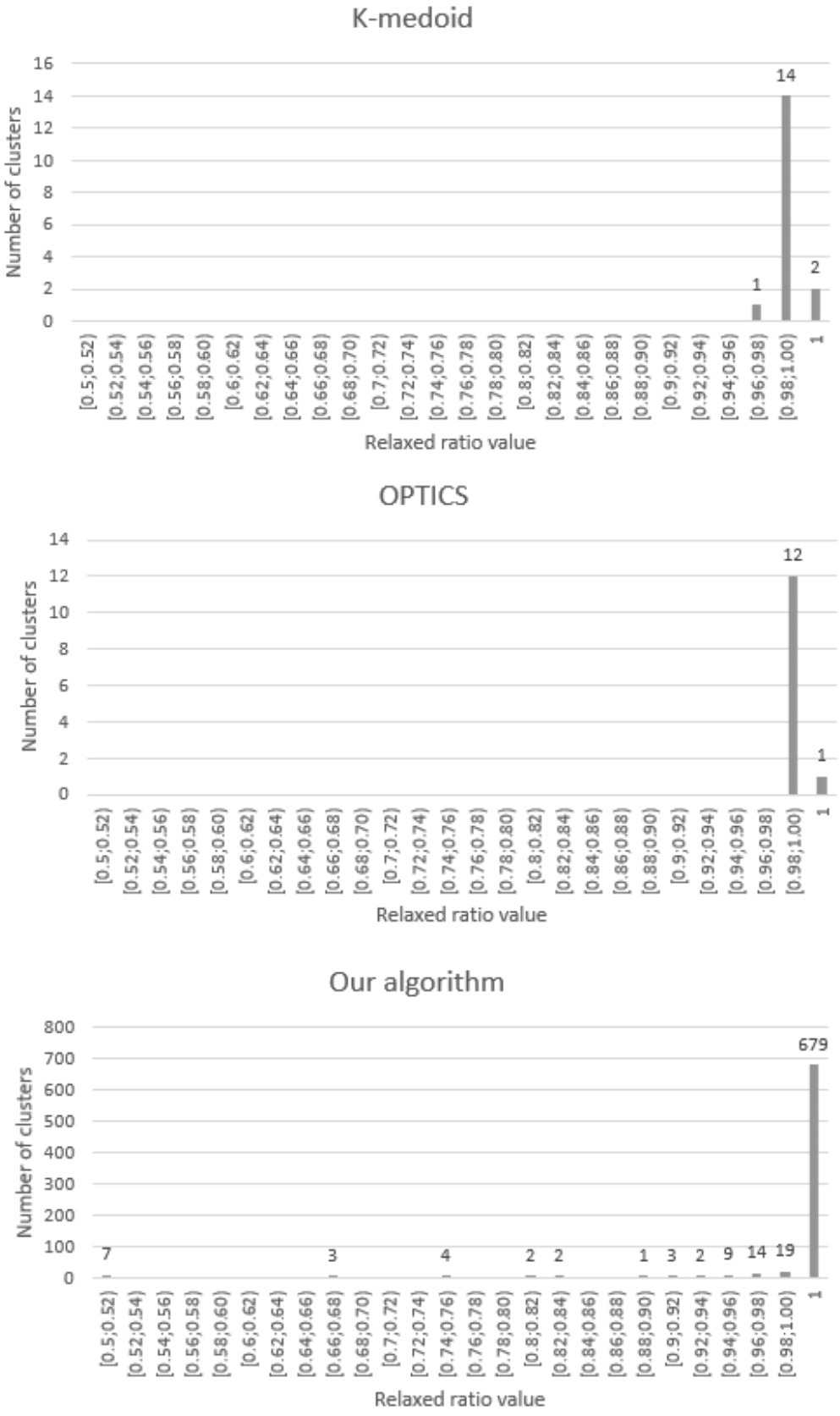

Figure 7.4: Relaxed Goodness Ratios of All Cluster Configurations

Chapter 8

Conclusion

In this document, I studied how binary similarity can be used to cluster malware samples and identify variants of malware families. I queried VirusTotal and obtained 12 993 malware samples belonging to malware families targeting the IoT ecosystem. I filtered the obtained data set using binary entropy calculation and YARA-rules in order to remove packed and/or encrypted samples. I applied two wide-spread clustering algorithms on the remaining 10 176 samples, k-medoid and OPTICS, using pair-wise TLSH differences as a distance matrix.

The results of the applied algorithms raised a number of issues. Firstly, the diameters of k-medoid's clusters were too big to represent members as variants of the sample family. I determined the threshold TLSH difference for variants to be 48 based on the EMBER data set. By contrast, the smallest diameter of k-medoid's cluster configuration was 180. Several OPTICS configurations suffered from large cluster diameters as well. In addition, its configurations detected many of my samples as outliers and did not include them in any of the calculated clusters. The standard approach to deal with outliers is to remove them from the data set, however, in my scenario, the outliers are the most interesting samples. They can signal previously unknown malware families and/or behaviors. As a result, such balancing is undesirable. Lastly, algorithms assume that values in the distance matrix are distances in the mathematical sense. However, such an assumption does not hold for TLSH which may explain the algorithms' poor performances.

In order to overcome the limitations encountered by k-medoid and OPTICS, I developed a new clustering algoritm which uses the data set structure information calculated by OPTICS. The main idea of my approach is to first identify dense regions in the data set and use the data points in the center of the dense regions as cluster heads. I include a data point in a given cluster if and only if its TLSH difference from the cluster head is within a threshold value. I also merge clusters together if they are found to be close to each other and the merged cluster does not exceed a pre-specified diameter threshold.

My experiments showed that my approach significantly outperforms k-medoid and OPTICS. Firstly, the cluster configuration calculated by my approach diameters much closer to 48, they range between 1 and 64. This is better then diameters produced by either OPTICS or k-medoid. Secondly, my approach detects a significantly lower number of samples as outliers then OPTICS. Thirdly, the distribution of malware families in clusters is purer than the distributions seen in k-medoid and OPTICS configurations.

I thus conclude, that TLSH can indeed be used for clustering malware samples into variants of malware families. However, new algorithms are required which take into consideration the specifics of TLSH differences. TLSH differences are not distances in the mathematical sense and therefore should only be used in a pair-wise fashion. What is more, TLSH differences have semantic meanings in the domain of malware analysis. New algorithms must respect and potentially rely on such semantic meaning in order to cluster samples more accurately.

Acknowledgements

The presented research has been partially supported by the SETIT Project (no. 2018-1.2.1-NKP-2018-00004), which has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme, and by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

I am grateful to Ukatemi Technologies for providing us with access to VirusTotal's private API and to Ukatemi's internal malware repository.

Bibliography

[1] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. In *ACM Sigmod record*, volume 28, pages 49–60. ACM, 1999.

[2] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, Vancouver, BC, August 2017. USENIX Association. ISBN 978-1-931971-40-9. URL `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis`.

[3] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding linux malware. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 161–175. IEEE, 2018.

[4] Michele De Donno, Nicola Dragoni, Alberto Giaretta, and Angelo Spognardi. Analysis of ddos-capable iot malwares. In *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 807–816. IEEE, 2017.

[5] Leonard Kaufman and Peter J Rousseeuw. *Finding groups in data: an introduction to cluster analysis*, volume 344. John Wiley & Sons, 2009.

[6] Ya Liu and Hui Wang. Tracking mirai variants. In *VirusBulletin*, 2018. URL `https://www.virusbulletin.com/virusbulletin/2018/12/vb2018-paper-tracking-mirai-variants/`.

[7] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–45, 2007.

[8] Andrei Novikov. PyClustering: Data mining library. *Journal of Open Source Software*, 4(36):1230, apr 2019. DOI: `10.21105/joss.01230`. URL `https://doi.org/10.21105/joss.01230`.

[9] Jonathan Oliver, Chun Cheng, and Yanggui Chen. Tlsh–a locality sensitive hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pages 7–13. IEEE, 2013.

[10] Fabio Pagani, Matteo Dell'Amico, and Davide Balzarotti. Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 354–365. ACM, 2018.

[11] Edward Raff and Charles Nicholas. Lempel-ziv jaccard distance, an effective alternative to ssdeep and sdhash. *Digital Investigation*, 24:34–49, 2018.

[12] V. Roussev. Data fingerprinting with similarity digests. *IFIP Advances in Information and Communication Technology*, 337 AICT:207–226, 2010. DOI: 10.1007/978-3-642-15506-2_15. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-78651093858&doi=10.1007%2f978-3-642-15506-2_15&partnerID=40&md5=d72d586c1e2186fdc9519c8ca35661f9. cited By 88.

[13] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquin Garcia-Alfaro, editors, *Research in Attacks, Intrusions, and Defenses*, pages 230–253, Cham, 2016. Springer International Publishing. ISBN 978-3-319-45719-2.

[14] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[15] Csongor Tamás and Boldizsár Bencsáth. Method for similarity searching in large malware repository. 2018.

[16] Benjamin Vignau, Raphaël Khoury, and Sylvain Hallé. 10 years of iot malware: a feature-based taxonomy. 2019.