February 28, 2019

CATCHING FLUX-NETWORKS IN THE OPEN

THESIS

R. Kokkelkoren
r.kokkelkoren@student.utwente.nl,
University of Twente,

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
DACS

Exam committee:
M. Jonker MSc
prof.dr.ir. A. Pras
dr. A. Sperotto

Documentnumber
— v1.0

UNIVERSITY OF TWENTE.

abstract The Domain Name System (DNS) protocol is one of the core protocols of the Internet which is used to map human-readable names into machine-readable IP addresses. The flexibility and broad implementation of the DNS protocol lead to alternative uses of the protocol such as provide load-balancing, high availability and performance services. Both malicious and benign networks, such as Content Delivery Networks, widely use these features to improve reliability and availability. The malicious variant of these networks are named flux-networks, and malicious actors use it for a wide range of malicious activities. These networks are known to use the DNS protocol properties to increase the difficulty in nullifying these malicious networks. Various studies exist in the literature that use detection methodologies to detect these types of networks.

In recent years a novel platform for active DNS measurements was established called *OpenINTEL*, this platform gathers DNS records of around 60% of the global DNS namespace and stores the records in a continuously updated unique large-scale data set. This data set has lead to novel insights for a varying range of topics such as the *insight into the use of cloud mail platforms* [1], *measuring exposure of DDoS protection services* [2], and more. Moreover, we want to study if it can also improve flux-network detection.

In this thesis, we present a methodology for identifying flux-networks that clusters the data records from *OpenINTEL* and uses a known malicious ground-truth for the identification of malicious networks. Our methodology is an adaptation of the work by Perdisci et al. [3] streamlined to work with *OpenINTEL* data. Using our detection application, we analyze every DNS record in *OpenINTEL* for the year 2017 for the Netherlands TLD.

Our results highlight that it is possible to implement a detection methodology for the *OpenINTEL* data set. This detection methodology did result in the identification of a total of 97.285 malicious networks. The dimensionality of *OpenINTEL* is significantly larger than previous studies, but the detection methodology did not result in the identification of actual flux-networks. We found that the lack of limiting the analysis to a single TLD or to the fact that *OpenINTEL* only gathers 2-level domain names may impede detection.

Our case study shows that the *guilty-by-association* techniques used to label networks as flux-networks can affect detection accuracy. This commonly used technique in flux-network detection may, therefore, have to be revisited to improve existing solutions.

Keywords - **aDNS, pDNS, OpenINTEL, flux-networks, domain-flux, IP-flux**

Acknowledgements First of all, I would like to thank the University of Twente in allowing me the opportunity to study, learn and improve myself. Mainly I would like to thank M. Jonker, my mentor in assisting me in finishing this thesis to graduate from my master studies. Although completing the thesis did take longer than expected, your advice and feedback has proven invaluable. Especially your effort and extra time you put into implementing a working Spark application at the University of Twente was much appreciated.

Secondly, I would like to thank SIDN for allowing me the opportunity of researching a part of my thesis at their organization. Their effort in making their resources readily available for students is a clear indication of their continued effort in improving the cyber security field. It was surprising how fast and easily I could make use of their data and the assistance I received from SIDN.

I also would like to thank my friend D. Planque which has provided clear feedback on the initial draft versions of my thesis. I can imagine that sifting through my initial versions of this paper and providing the appropriate feedback was much work. Your effort is highly appreciated and helped in getting the final version on the proper level.

Lastly, I would like to thank my most significant support, Doortje. I can say that without you I would not have finished this thesis. Your continued support, help, feedback, and love is the thing what has kept me going. I can only say that I'm looking forward to the next great adventure that we will undertake together and that I have no doubt that it will be a successful one.

For D&K

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Domain Name System (DNS) protocol is one of the core protocols of the Internet that is used to map human-readable names into machine-readable IP addresses and thus provides a crucial role in the continued operation of the Internet. The DNS protocol was initially proposed in 1983, but recent uses of the DNS protocol have long diverted from its initial goal. The broad implementation leads to the DNS protocol not being used solely to map domain names to IP addresses but also resulted in it being used to provide load-balancing, high availability and performance services. Benign systems such as Content Delivery Networks (CDNs) use this functionality to create resilient networks. These features for networks are achieved by rapidly changing the IP-addresses of the related domain names. The CDN then uses this functionality to assure reliable network connections for CDN users.

The same techniques are also used by malicious actors who use it to make their networks more resilient against takedown requests and to increase the overall availability and performance. We regard these agile malicious networks as CDNs used for malicious purposes that provide a wide range of malicious activities, such as phishing campaigns, distribution of malware and more. Security agencies around the world are in a continuous effort to remove these malicious distributed networks. A prominent approach for taking down these networks is to disable the systems that malicious actors use to control the specific network; these systems are in general referred to as *Command & Control* (C2) servers. Previously this was accomplished by analyzing the malware samples related to the network, thus the software applications that are used to propagate the network, to determine which domain names are in use by the C2 servers and then to blacklist those domains. Due to these actions by law enforcement agencies, malicious networks have begun to include additional defensive techniques called *IP-flux* and *domain-flux* to prevent these types of takedown actions. The implementation of these defensive techniques resulted in a significant increase in difficulty of taking down malicious networks; the networks that use these techniques are referred to as *flux-networks*.

Recent threat intelligence reports, such as those published by Symantec [4], show that the growth of new malware variants shows a steady increase and we, therefore, expect that the use of flux-networks will also show steady growth. Furthermore, phishing attacks, one of the malicious purposes of a flux-network, are still prevalent as stated by Symantec [4] and we, therefore, expect a continuous use of flux-networks.

The DNS protocol is a fundamental part of the agile properties of these networks and can easily be analyzed since the DNS protocol, by default, is unencrypted, and thus can be used to detect anomalous behavior. Several kinds of research [5, 6, 7, 8, 9, 10] have shown that analyzing DNS communications is an effective method against combating these malicious practices of flux-network. These studies have focused on using machine learning and clustering techniques to perform analysis on DNS communications between servers and clients to identify flux-networks.

At the time of writing, there are no studies related to flux-network detection focused on DNS records relevant to the Netherlands top-level-domain (TLD). Therefore, it is unknown whether components of flux-network have used domain names using the Netherlands TLD.

The lack of any DNS data set which is available for the detection of flux-networks relevant to the Netherlands TLD is probably the main reason why there are not any relevant case studies of flux-network detection for the Netherlands. However, recent collaboration between the University of Twente[1], Surfnet[2] and SIDN[3] have resulted in the development of a new active DNS (*aDNS*) measurement system called *OpenINTEL* [1] which is a new source for flux-network detection techniques and other IT security related researches.

OpenINTEL is used to perform large-scale *aDNS* measurements that generates a daily overview of the entire DNS namespace for numerous TLDs, such as *.com, .org and .net*. The *OpenINTEL* platform is, therefore, an interesting novel DNS data set containing current and historical DNS records that include at least 60% of the entire global DNS namespace. The number of TLDs supported by the *OpenINTEL* platform is still growing, and therefore the resulting data set is becoming increasingly a better representation of the entire global DNS namespace. Currently, the *OpenINTEL* platform collects DNS records for every 2-level domain name (2LD) within the available TLDs. Various DNS properties are recorded for each DNS record such as `A`, `AAAA`, `NS`, and `DNSKEY` records. The most interesting aspect of the *OpenINTEL* platform is that the gathered data is available for an extended period, meaning that the platform generates a complete historical data set of a large part of the global DNS namespace.

The recent development of the *OpenINTEL* platform and the characteristics of its data set increases its value as a data source for flux-network detection mechanisms. Initially, it is essential to determine whether it is possible to implement a known flux-network detection mechanism using the data from *OpenINTEL*. This implementation might be difficult to implement due to two reasons, initially due to the dimensionality of the data stored by *OpenINTEL*, which is significant. Secondly, since most current studies on flux-network detection are all based on passive DNS (*pDNS*) data sets and therefore might require different data structures than currently available in *OpenINTEL*. *Passive DNS* is a technique in which DNS communications within a network are monitored and stored for later research, meaning that data set only contains records from actively queried domains by users from the monitored networks. Due to this requirement, it is improbable that a *pDNS* data set contains every domain name available within the DNS namespace of the relevant TLD.

Also, it is impossible to determine the completeness, thus the percentage of the total DNS namespace for which there are records in the data set, of the *pDNS* data set by itself. Verification of the completeness of a *pDNS* data set can only be determined by using external data for verification, such as the zone files of the respective TLDs. This potential lack of completeness of the data set increases the difficulty of a proper analysis of flux-networks because the completeness of the data records cannot be guaranteed. Previous studies all used *pDNS* data set because until recently there had not been a large scale *aDNS* implementation available which systematically gathers every available domain name.

1.1 Research topic

Given the potential deficit of *pDNS* data sets and the novel *OpenINTEL* data set, it is prudent to determine the possibility of applying known flux-network detection mechanism to this new data. A case study in trying to detect any known flux-network in the Netherlands TLD is a perfect opportunity to verify the potential of such a flux-network detection mechanism and will also hopefully show any insight in flux-network activity within the Netherlands TLD.

This thesis will, therefore, consists of two goals. First, to determine the applicability of existing flux-network detection mechanisms to the *OpenINTEL* data set. Second, to investigate whether domains under the Netherlands TLD are abused for malicious purposes by flux-networks. To accomplish these goals, we define the following main research question:

[1] https://www.utwente.nl/
[2] https://www.surf.nl/en/about-surf/subsidiaries/surfnet
[3] https://www.sidnlabs.nl/

Can we use a novel active DNS measurement to identify flux-networks, and its components, in a case study for the Netherlands TLD?

We break this question down in the following subquestions:

RQ_1 Can previously researched detection methods be applied to the OpenINTEL DNS measurements. If not, are there other methods suited for identifying these networks?

RQ_2 Are the results of the flux-network detection system sufficiently reliable to get detailed characteristics of the identified flux-networks?

RQ_3 Are there any disadvantages of, or limitations to, using active DNS measurement data from the OpenINTEL platform to the end of fast-flux detection?

1.2 Background

This section contains a brief explanation of the core components used throughout this thesis.

1.2.1 IP-flux

IP-flux also referred to as Fast-flux, is a technique of continually changing the IP address associated with a Fully Qualified Domain Name (FQDN) [11]. This methodology uses the time-to-live (TTL) values of DNS resource records to ensure that DNS records for certain FQDNs can change in very short periods. The TTL values determine how long a particular domain name is cached in recursive DNS servers before being actively queried again. Setting this TTL value to a very low number is a method to ensure that the domain names are actively queried and thus allows for the possibility to change the associated IP-addresses rapidly by registering and removing the registration of the associated DNS records. This method, in turn, allows for the possibility to change the associated IP address continuously and reroute network traffic. This method is widely used by CDNs to provide load-balancing and other availability increasing capabilities. Legitimate applications usually use some round-robin technique to iterate over the available IP addresses; however, malicious actors also use it to protect the associated IP addresses related to a malicious FQDN. Using IP-flux increases the difficulty for organizations to pinpoint the associated systems related to a large botnet or flux-network. We show a visual representation of IP-flux in Figure 1.1.

1.2.2 Domain-flux

Domain-Flux is a similar technique to IP-flux, but instead of constantly changing the IP addresses associated with a domain, the domain name itself constantly changes but still refers to a common IP address. This method use algorithmically generated domain names to refer to systems used within the malicious networks such as C2 systems. This method of *domain fluxing* was first mentioned by Yadav et al. [12], which associated the application of this type of method to well-known botnets at the time such as *Conficker* and *Kraken*. The use of a domain generating algorithm (DGA) has since then been widely adopted in various malicious applications and has further increased the difficulty in identifying and stopping large botnets. A network which uses this domain fluxing referred to as a domain-flux network. In Figure 1.2 a visual representation of domain-flux is shown.

1.2.3 Domain generation algorithm

Domain generation algorithm (DGA) is a method for generating seemingly random domain names based on a particular input that could be a random seed or timestamp. Yadav et al. [12] performed one of the first studies into this field which resulted in the analysis of the properties of the DGA used by known malware such as *Conficker, Kraken* and *Torpig*. A DGA aims to prevent the disclosure of the relevant systems used in malicious networks by using random domain names that are not possible to predict without the knowledge of the inner workings of the algorithm used to generate these domain names. The administrator of the malicious

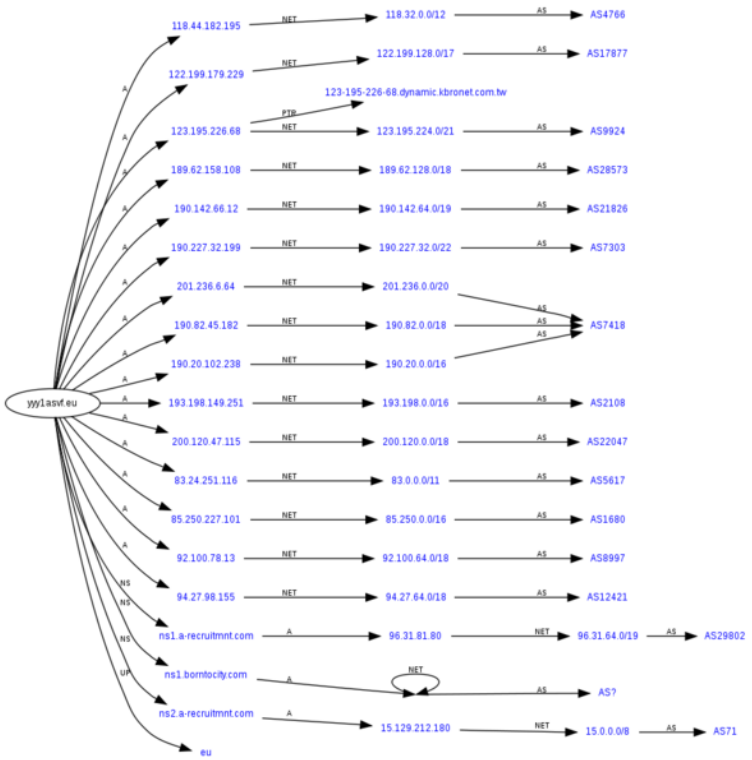


Figure 1.1: Visual representation of IP-flux

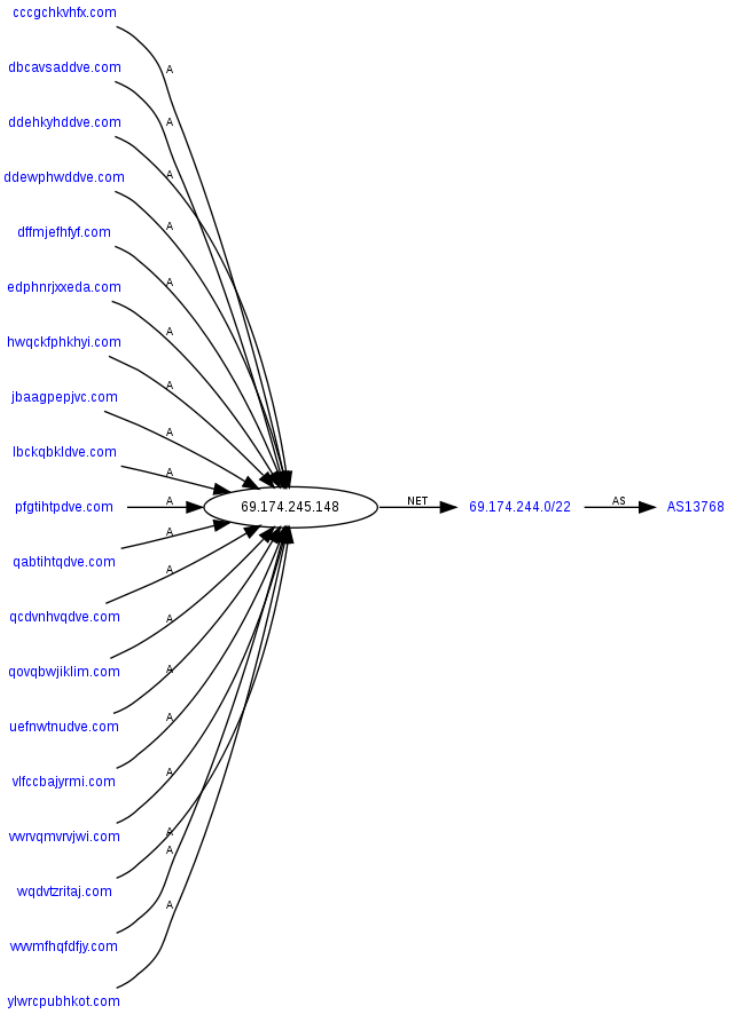


Figure 1.2: Visual representation of domain-flux

network is, of course, aware of the algorithm and ensures that he configures the generated domains for the specific period. The DGA has to be accessible by the components that use the network, so for example, in a botnet, the malware that distributes the botnet usually contains the DGA which is required to contact the appropriate domain name at the correct time. Due to this type of implementation, the DGA, if used is one of the primary targets during reverse engineering of the associated malware. We show an example of an implementation of a DGA in Listing 1.1, which is the DGA function used by the *Dyre/Dyreza* malware samples as documented by Chiu and Villegas [13]. Generally, domain-flux makes use of a DGA to generate and access the relevant domain names.

```
from datetime import date
from haslib import sha256

def dyre_dga(num, data_str=None):
    if None == data_str:
        data_str = '{0.year}-{0.month}-{0.day}'.format(date.today())

    tlds = ['.cc', '.ws', '.to', '.in', '.hk', '.cn', '.tk', '.so']
    hash = sha256('{0}{1}'.format(data_str, num)).hexdigest()[3:36]
    replace_char = chr(0xFF & ((num % 26) + 97))

    return '{0}{1}{2}:443'.format(replace_char, hash, tlds[num % len(tlds)])

todays_domains = [dyre_dga(i) for i in xrange(333)]
```

Listing 1.1: Example of a DGA algorithm as described by Chiu and Villegas [13]

1.2.4 Passive DNS

Passive DNS, *pDNS* or *passive DNS replication*, is a technique that has been initially described by Weimer [14] and consists of monitoring and storing the DNS packets on the network for later analysis. This process ensures that there is a database with up-to-date information regarding the DNS entries that are sent by the monitored network. These types of databases are used for security research, incident response or other relevant process.

Due to the implementation of *pDNS*, the database only contains actively queried DNS record from within the network. This deficit, in turn, leads to an incomplete and potentially biased data set because the *pDNS* only contains data which is *relevant* for the underlying network. Weimer describes this deficit as:

> **Weimer [14]**, Compared to the approach based on zone files; there is an important difference: we can never be sure that our data is complete. However, if passive DNS replication is used to support mostly local decision, this is not a significant problem in most cases; there is no customer interested anyway in records which are missing.

Although this setup is adequate for most cases because only DNS entries relevant to the specific network are required, this setup limits analysis that is not directly associated to the network but is used as a data source for other methods such as a pro-active detection techniques. Furthermore, this implementation also implies that most detection mechanisms that use *pDNS* require at least one victim who has accessed the malicious network before the DNS entry is recorded. This procedure ensures the registration of the DNS record within the database and then the detection mechanism would able to detect it.

1.2.5 Active DNS

The most significant difference between *aDNS* and active DNS (*aDNS*) is that *aDNS* actively queries FQNDs instead of passively monitoring a specific network. In general, there is not much difference between a system that generally queries FQDNs as part of its default operation or an *aDNS* system that uses DNS queries for security research except by the fact

that the number of queries is more significant for the *aDNS* system. Furthermore, the property of a *aDNS* data set is that its DNS records are not a good representation when compared to the DNS records from a live network. This deficit exists because the DNS records that are queried are specified beforehand. This *target* specification within *aDNS* is one of the reasons its implementation in security-related researches is limited because the specified target can notice an increase in the DNS queries for their respective domains and this, in turn, might alert certain malicious actors that their systems are under investigation.

1.2.6 OpenINTEL

The *OpenINTEL* platform developed by van Rijswijk-Deij et al. [1] is a high-performance scalable infrastructure for large-scale active DNS measurements. The *OpenINTEL* platform is unique because it gathers DNS records for, at the time of writing, at least 60% of the entire DNS namespace. It is possible to be this accurate because the *OpenINTEL* platform receives full DNS zone files for the available TLDs within *OpenINTEL* from the respective TLDs. This implementation means that the *OpenINTEL* platform functions on exact copies from a measured TLD and can, therefore, query every possible FQDN within the TLD. An overview of the *OpenINTEL* architecture is shown in Figure 1.3.

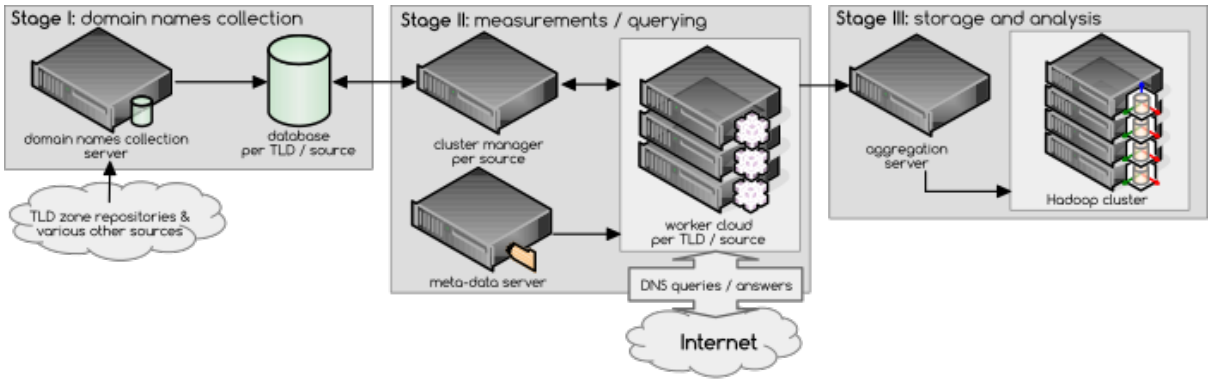


Figure 1.3: High level architecture of OpenINTEL [1]

Currently the *OpenINTEL* platform gathers records of several popular TLDs such as *.com, .org, .net*, and numerous country code top-level domain (ccTLD) such as *.ca, .fi, .nl, .se*. Due to the systematical requirements for gathering the high-dimensional data set and protocol structure of the DNS-protocol, the *OpenINTEL* platform only queries DNS records for every 2-level domain names within the available TLDs. The only exception is the *www* label which is also actively queried due to the wide usage of this label within DNS. Each FQDN queried results in a multitude of DNS resource records being stored including `DNSSEC`, `TXT`, and other relevant DNS resource records for a complete overview see Table 1.1.

Resource Record	Description
SOA	The Start of Authority record specifies key parameters for the DNS zone that reflect operational practices of the DNS operator.
A	Specifies the IPv4 address for a name, including *www* and *mail* labels.
AAAA	Specifies the IPv6 address for a name, including *www* and *mail* labels.
NS	Specifies the names of the authoritative name servers for a domain.
MX	Specifies the names of the hosts that handle email for a domain.
TXT	Contains arbitrary text strings
SPF	Specifies spam filtering information for a domain. Note that this record type was deprecated in 2014 (RFC 7208), we query it to study decline of an obsolete record type of time.
DS	The Delegation Signer record references a DNSKEY using a cryptographic hash. It is part of the delegation in a parent zone, together with the NS and established the chain of trust from parent to child DNS zones in DNSSEC.
DNSKEY	Specifies public keys for validating DNSSEC signatures in the DNS zone.
NSEC	Used in DNSSEC to provide authenticated denial-of-existence, i.e. to cryptographically prove that a queried name and record type do not exist.

Table 1.1: Recorded query types by *OpenINTEL* [1]

Chapter 2

Related Work

There already exist several mechanisms used for the detection of flux-networks. The specific implementation of these techniques varies widely and differs on whether it uses DNS responses, the DNS requests, it may use clustering or require a ground-truth of malicious domains and more. This chapter of related work consist of two sections. Initially, we describe the literary work of flux-network detection mechanisms that use solely the DNS responses in its detection methodology. Secondly, we describe the literary work where the detection mechanisms use both the DNS requests and responses.

2.1 Flux-network detection methodologies using DNS requests

One of the approaches to detect flux networks using DNS responses was reported by Perdisci et al. [3], which described a novel detection methodology called *FluxBuster*. This methodology uses *pDNS* responses from the Internet Systems Consortium's Security Information Exchange (ISC/SIE)[1] as its initial data set. The SIE project of ISC is a public benefits project which strives to enhance the cooperation of security companies. Especially by making *pDNS* data sets available for research. This *pDNS* data set was used by Perdisci et al. [3] to generate clusters of domain names and IP addresses which are related and could be a potential flux network. A classifier algorithm performs the actual verification of whether a cluster is deemed malicious or benign. By using this classifying approach, Perdisci et al. managed to get a $99,3\%$ true positive rate (TPR) and a 0.15% false positive rate (FPR) for the *FluxBuster* detection methodology. These results show that *FluxBuster* operates with high efficiency, but it still requires a relatable entry in the ISC/SIE data set before the system can detect any potential malicious network, so at least one victim should have accessed the flux-network before it can be detected. The *FluxBuster* detection mechanism uses a *single-linkage* hierarchical cluster algorithm (HCA) as described by Jain and Dubes [15] to cluster domain names. Although HCA also refers to hierarchical clustering analysis in this thesis, we will use it as an abbreviation for hierarchical clustering algorithm, which is a clustering algorithm that uses a similarity matrix containing the similarity weights of each set to cluster relevant record. A single-linkage bottom-up HCA algorithm, as used by Perdisci et al. [3], defines each domain as an individual cluster and combines the two nearest clusters given the similarity within the matrix. The *Jaccard similarity* with a *sigmoidal* weight is used to determine the similarity between two domain names based upon the resolved IP sets. Using the HCA algorithm a *dendrogram* is then created which consists of all domain names clustered together. By defining a certain height and cutting the *dendrogram* at that specific level, the *dendrogram* results in the clusters which are potential flux networks. Although the use of this clustering algorithm is one of the reasons why the *FluxBuster* can function under such high efficiency, it may also be the cause of the long processing time that is required by *FluxBuster* to analyze the results. The C4.5 decision-tree algorithm is used to determine whether a network is deemed malicious or benign. This algorithm decides the maliciousness of a domain name based upon a set of predefined features, such as IP and domain diversity, DNS TTL and growth ratio. It is interesting to notice

[1] https://sie.isc.org

that *FluxBuster* has resulted in such a high efficiency without any lexical analysis on the domain names. Thus, *FluxBuster* does not analyze the domain names in the cluster to determine whether a domain name is benign or created by a DGA.

There also exists techniques based on active machine learning algorithms to detect malicious domain names without focusing on a potential malicious network related to the domain name. An example of such is *Exposure* which was developed by Bilge et al. [5], *Exposure* uses a data set similar to *FluxBuster* and was based upon *pDNS* data from ISC/SIE. Although the data set was similar, the methodology of *Exposure* and *FluxBuster* differs greatly, mainly that *Exposure* only detects malicious domain names and does not attempt to cluster malicious domain names together to identify a potential flux network. The *Exposure* application uses a wide range of features related to the domain name following the C4.5 decision tree algorithm to identify malicious domain names. The features used to categorize the domain names are time-based features, DNS answer-based features, TTL value-based features, and domain name based features. The specific features that were chosen to identify malicious domain names were determined using a genetic algorithm that showed the most efficient feature set which results in the highest TPR and lowest FPR. Using feature sets determined by this genetic algorithm, the *Exposure* application functioned with a 99.5% detection rate and 0.3% FPR. Although these results indicate that the *Exposure* application can function with high efficiency, the lack of any clustering of malicious domain names might result in some networks not being detected. Especially flux networks that use IP-flux may have related IPs that change too quickly and thereby evade the detection algorithm. However, this study does indicate that it is possible to achieve a high detection rate by focusing on a single domain name and the related DNS responses. Both the *Exposure* and *FluxBuster* detection methodology has a high TPR and low FPR as shown in Table 2.1 even though both detection mechanisms implement very different approaches. These results lead to the impression that a combination of both methodologies might further improve the detection efficiency.

The previously mentioned detection techniques all use public or commercial blacklists and whitelists, either as ground truth or as training data for machine learning classifiers. However, Stevanovic et al. [16] points out that the use of public or commercial blacklists and whitelists as input for the learning algorithm impacts the overall detection efficiency. Stevanovic et al. argues that these public or commercial available blacklists and whitelists are inaccurate which might lead to an increase of false positives and true negatives. They consider some of the used blacklists and whitelists as inaccurate because there generated without sufficient verification which might lead to false entries within these lists. For example, as argued by Stevanovic et al., some public list are based on entries submitted or categorized by the general public which all have different technical backgrounds and perspectives. As stated by Stevanovic et al., this implementation decreases the overall quality of these lists due to potential false positives within these blacklists. To analyze these inadequacies of the blacklists and whitelists, Stevanovic et al. developed a DNS labeling technique for detecting agile DNS traffic. The methodology uses an application called *DNSMap*[8], that is used to generate graph components which resemble agile networks which might be benign or malicious. Using the K-means clustering algorithm distributes the networks in malicious or benign clusters depending on the characteristics of the network graph. Although most of the characteristics for detecting flux networks are similar as previous studies, Stevanovic et al. chose to use a blacklist of FQDNs as a characteristic, instead of using it as the ground truth or as a training data set for the machine learning algorithms. This implementation resulted in a remarkably low TPR of 73% and an FPR of 13%, indicating that the overall setup of this specific implementation was not efficient.

2.2 Flux-network detection methodologies using DNS responses

Besides detection mechanism that use only DNS responses, there are also detection methodologies that take the actual DNS request into account. One such methodology is called *Segugio* which is described by Rahbarinia et al. [10]. This methodology uses client behavior in addition to the DNS responses to generate a graph containing clients and FQDNs as nodes. Connections are established between the nodes whenever a client requests a certain FQDN.

Segugio uses a labeling process of identifying both the clients and FQDNs as either benign or malicious. They state that they identify clients as compromised when they connect to malicious domain names. The DNS requests from these compromised systems are then analyzed to detect new malicious domain names. By performing this analysis on the entire graph, it is possible to map and identify both compromised clients as well as malicious domain names. The benefit of this method is that compromised clients can also be easily detected and more quickly quarantined. Although this methodology shows a 94% TPR and a 0.1% FPR, it does require access to every DNS request made by each client, meaning that it has a severe impact on the privacy of the clients. Therefore this implementation might be difficult to realize in certain situations. Furthermore, this mechanism relies heavily on public or commercial blacklists to provide the first ground truth of compromised clients and domain names meaning that the detection mechanism cannot function without a reliable third-party further reducing the overall applicability of this method.

Another research which takes the client DNS request into account is the detection mechanism called *Graph-based Malware Activity Detection (GMAD)* which was described by Lee and Lee [9]. The study focuses on the *sequential correlation* of DNS traffic, this consists of the correlation of the specific sequence in which users query two different domain names. Lee and Lee determine the *sequential correlation* between two domain names by using the client sharing ratio (CSR), which they calculate by using the *Jaccard similarity* of the source IP addresses between the two domain names. The detection mechanism consists of three steps, initially, the generation of the graph containing the domain names and the corresponding CSR. Secondly, they cluster the graph into multiple graph components resembling related domain names and, finally, the malware detection. The clustering algorithm uses the CSR, the number of clients and the number of queries, the algorithm is then applied to the graph with increasing thresholds to ensure the components are reduced iteratively in size. The result of this algorithm is the dissected graph in numerous graph components which either resemble benign or malicious domain names that are related to each other. By looking up the domain names in a known blacklist, they verify the maliciousness of the actual domain name. This methodology results in the mechanism only being able to detect malicious domain names that are already detected by other detection mechanisms and is therefore reliant on the validity of third-party blacklists. The benefit of this mechanism is the possibility to detect malicious domain names that are related to known malicious domain names. It is interesting to note that the results of this research are based solely on 8 hours worth of DNS traces. This data set is minimal when compared to the months worth of DNS traces other studies have used. The mechanism itself ensures a $89,8\%$ TPR and a 0.13% FPR. However, the specific precision for the initial four data sets differs significantly; this might be an indication that the precision of the mechanism is dependent on the initial data set.

2.3 Related work conclusion

Table 2.1 summarizes the techniques that we found in related work. In this overview, it is easy to see that there exist many variations in the exact implementation of the detection methodologies. This variation in implementation also results in significant differences in the TPR and FPR of the various methodologies.

	Data source	DNS Response / Request	Clustering	Require ground truth	TPR / FPR
FluxBuster [3]	ISC/SIE *pDNS*	✓/ ×	*Jaccard similarity*	×	99.3% / 0.15%
Ground Truth [16]	ISPs *pDNS*	✓/ ×	*DNSMap* [8]	✓	73.0% / 13.0%
Exposure [5]	ISC/SIE *pDNS*	✓/ ×	No clustering	×	99.5% / 0.30%
GMAD [9]	ISPs *pDNS*	✓/ ✓	*Jaccard similarity*	✓	$89,8\%$ / $0,13\%$
Segugio [10]	ISPs *pDNS*	✓/ ✓	No clustering	✓	94.0% / 0.10%

Table 2.1: Overview of characteristics of flux-network detection methodologies

Chapter 3

Research Method

The primary goal of this study is to determine the possibility of implementing known flux-network detection mechanisms using the *OpenINTEL* data set. As described in Chapter 2, many related studies already exists in the literature. The applicability of these detection methods on the *OpenINTEL* data set, however, is uncertain and because of the differences in the dimensionality of the data, the implementation might not be trivial. Furthermore, although *OpenINTEL* measures numerous DNS records as shown in Table 1.1, it does not store all the properties from the DNS responses such as the TTL values. Some of these DNS properties might be required and thus increase the difficulty of implementing the specific detection method.

In Table 2.1 we show an overview of relevant detection methods. This overview shows that some of the detection methods require analysis of both the DNS request as well as the DNS response to detect potential flux-networks. There also exist detection methods that take a compromised client into account and analyze the DNS request sent by those clients. Since *OpenINTEL* is a single *aDNS* system and not a network of clients, and because it does not store the DNS request, it is not possible to apply these methods to *OpenINTEL* data set. The detection methods named *Exposure* by Bilge et al. [5], *FluxBuster* by Perdisci et al. [3] and *Ground Truth* by Stevanovic et al. [16] only require DNS responses and are therefore the most likely applicable methods for the *OpenINTEL* data set.

Table 3.1: Overview of feature sets used to detect flux-networks

Feature Category	Feature Set	*Exposure* [5]	*FluxBuster* [3]	*Ground Truth*[16]
Time-Based	Short life	✓	×	×
	Daily Similarity	✓	×	×
	Repeating Patterns	✓	×	×
	IP growth ratio	×	✓	×
DNS Answer	No. distinct IPs	✓	✓	✓
	No. distinct domain names	×	✓	✓
	No. distinct Countries	✓	✓	✓
	Reverse DNS	✓	×	×
TTL	No. distinct TTLs	✓	✓	×
	No. TTL change	✓	×	×
	No. scattered TTL	✓	×	×
Domain Name	% numerical char	✓	×	✓
	No. English words	×	×	✓
	Length of LMS	✓	×	✓
Network	IP diversity	×	✓	✓
	No. domain names	×	✓	✓

We show an overview of the required data for the specific detection method in the Table: 3.1. We note that both *FluxBuster* by Perdisci et al. [3] and *Ground Truth* by Stevanovic et al. [16] implement some form of clustering, as can be seen in Table: 2.1, but *Exposure* by Bilge et al. [5] does not use any clustering. The *Exposure* detection method requires multiple TTL features

from the DNS responses that are not available by *OpenINTEL*, which means that implementing this specific method might be difficult.

In general both the *FluxBuster* detection method by Perdisci et al. [3] as well as the *Ground Truth* detection method by Stevanovic et al. [16], are suitable for the *OpenINTEL* data set. The *Ground Truth* detection mechanism is the most likely candidate because it does not require TTL feature set and might, therefore, be the easiest to implement. The TPR/FPR (73.0% / 13.0%) for the *Ground Truth* detection method is, however, remarkably lower than the other methods. As shown in Table 2.1, this detection method is the only method with a TPR lower than 89% and a FPR higher than 0.30%. Since the TPR and FPR of the *Ground Truth* detection method are remarkably lower the potential results from this method are more unreliable. Therefore, we choose to implement the *FluxBuster* detection method on *OpenINTEL*. The TPR/FPR of *FluxBuster* are one of the highest in comparison with the other methods, and the required feature set are largely compatible with the record DNS records in *OpenINTEL*.

The *FluxBuster* detection algorithm roughly consists of several procedures to identify flux-network clusters. The algorithm implements both a clustering algorithm to cluster relevant records and a classifying algorithm that is used to identify the flux-networks. Delving into the specifics of the classifying method of *FluxBuster*, we reveal that implementing an exact copy of the methodology for the *OpenINTEL* data set is going to take extensive time and effort. This increased effort means that, given the practical limitations of this thesis, there is not going to be sufficient time available to verify and analyze the actual results of the identification methodology. Given this fact, we have decided that we are going to implement a simpler classifying algorithm based on a known malicious ground-truth for the detection of flux-networks so that we have ample time available for adequately analyzing the actual results.

So we have to implement a system that contains the following procedures that are significantly based on the methods of *FluxBuster* detection methodology to analyze the *OpenINTEL* data set. In the following sections, we describe the implementation of both the clustering, identifying and validating processes.

1. Clustering relevant FQDNs and IPs based on *Jaccard similarity*.
2. Identifying malicious flux-networks based on a known malicious ground-truth.
3. Validating the detection application results for flux-networks.

3.1 Clustering using HCA

The high dimensional data set of *OpenINTEL* makes it very difficult to implement a flux-network detection mechanism without using some form of clustering. When compared to previous studies the amount of data that is available in *OpenINTEL* is exceedingly higher. To handle large data sets and to implement a known detection method, we use a similar clustering algorithm to the algorithm used by *FluxBuster*.

Using a clustering algorithm for the *OpenINTEL* data set might be very beneficial for detecting every component of the flux-network. The advantage of the *OpenINTEL* data set is not the number of data points for each domain, but the near-complete coverage of all 2LDs for the queried TLD at a specific period. However, *OpenINTEL* records the various domains as individual records, and so the *OpenINTEL* data set does not contain any information with regards to underlying relations between those records, such as records matching to the same IP. It is possible to identify relatable records by analyzing the commonalities in the data of the records itself. Table 3.2 shows an example of *OpenINTEL* records having a commonality with each other based on the associated IP addresses. So to get an overview of all the components in a flux-network, it is essential to group relevant domains so that it is possible to link the DNS records of `www.example.com` with `example.com`, given that these records share a commonality based on IP-address.

The *FluxBuster* detection method uses a clustering process before classifying a cluster as malicious, to get a proper overview of all the components in the flux-network. The clustering

index	FQDN	IPv4/IPv6 address	day	month	year
0	inglesmundial.com.	104.25.94.7	15	03	2017
1	inglesmundial.com.	104.25.95.7	15	03	2017
2	likenhanh.net.	103.28.38.229	15	03	2017
3	www.likenhanh.net.	103.28.38.229	15	03	2017

Table 3.2: Example of the data format used in the OpenINTEL project

algorithm consists of clustering *similar* DNS records to assign relations between various DNS records. The resulting clusters can then be used for further analyses whether it be to identify malicious flux-network or other identifications of potential malicious behavior. Besides *FluxBuster*, other flux-network detection methods, such as those by Stevanovic et al. [16] and Lee and Lee [9], use clustering algorithms with a specific similarity indicator to group relevant records. The resulting clusters are then analyzed to identify potential malicious flux-networks from benign systems or CDNs.

3.1.1 Use of hierarchical clustering algorithm

The detection method described by Perdisci et al. use a *single-linkage* hierarchical clustering algorithm (HCA) by Jain and Dubes [15]. This algorithm calculates clusters of relevant domain names depending on the *similarity* of those domains. To be able to cluster these domains it is required to state what the actual similarity is between 2 domains. A popular choice for this similarity, also called the similarity index, is the *Jaccard similarity* also called the *Jaccard-index*. This similarity index is used to determine the similarity between two subjects by calculating the overlap in relevant information associated with those two subjects. In the case of clustering DNS records, the subjects are the domain names, and the relevant information are the IP addresses contained in `A` and `AAAA` DNS resource records. As shown in Equation 3.1, the *Jaccard similarity* in DNS records clustering is determined by the overlap of the IP addresses R_α and R_β associated with two domain α and β. Using this similarity-index, we calculate the exact similarity between two domains for which a resulting 1.0 indicates an exact match and a 0.0 indicates no overlap. The *FluxBuster* detection method uses the *Jaccard similarity* as their similarity index in their clustering algorithm.

$$sim(\alpha, \beta) = \frac{|R_\alpha \cap R_\beta|}{|R_\alpha \cup R_\beta|} \tag{3.1}$$

HCA implementations require a similarity matrix that contains the similarity index of every possible combination of domain names within a given set. More specifically, the similarity matrix $P = \{s_{ij}\}_{i,j=1\ldots n}$ consists of similarities $s_{ij} = sim(d_i, d_j)$ for each pair of domain names (d_i, d_j). The HCA configuration determines the exact process of clustering records depending on the similarity matrix. A single-linkage bottom-up HCA algorithm, for example, defines each domain as an individual cluster and combines the two nearest clusters given the *Jaccard similarities* within the similarity matrix. This algorithm results in the creation of a tree-like data structure containing nested clusters which we can visualize using a *dendrogram*. The *dendrogram* itself does not represent the actual partitioning of the clusters but rather the relevance between the clusters. By cutting the dendrogram at a specific relevance level h, we obtain the actual clusters.

3.1.2 Implementation of the HCA

An important factor of the correct implementation and use of the HCA is defining a proper cutting level h of the *dendrogram*. Perdisci et al. use a cutting level which they determined by analyzing the number of resulting clusters for various dendrogram cutting levels and analyzing the results of certain plateau regions within the graph. They describe this procedure as:

Perdisci et al. [3], In practice, we plot a graph that shows how the number of clusters varies by choosing different values of h, and we look for *plateau* (i.e., flat) regions in the graph that are an indication of "stability" or *natural clustering*. Plateau regions correspond to those steps of the algorithm where the two nearest clusters that have to be merged exhibit a quite low measure of similarity.

We use an approach similar to Perdisci et al. [3], and determine the cutting level of the HCA by plotting the number of clusters for a specific dendrogram cutting levels and verifying if there exist stable regions within the graph. We use a data set of 10.000 records randomly selected from *OpenINTEL* to verify this cutting level. We show the results of this analysis in Figure 3.1, the graph consists of roughly two major stable regions indicating some form of *natural clustering*. The largest of the two stable regions revolves around the cut threshold of 0.1 at the very start of the graph. This value indicates that there is only a 10% similarity required between the domain to form a cluster; this similarity is too low to be of practical use, and therefore this flat region is discarded.

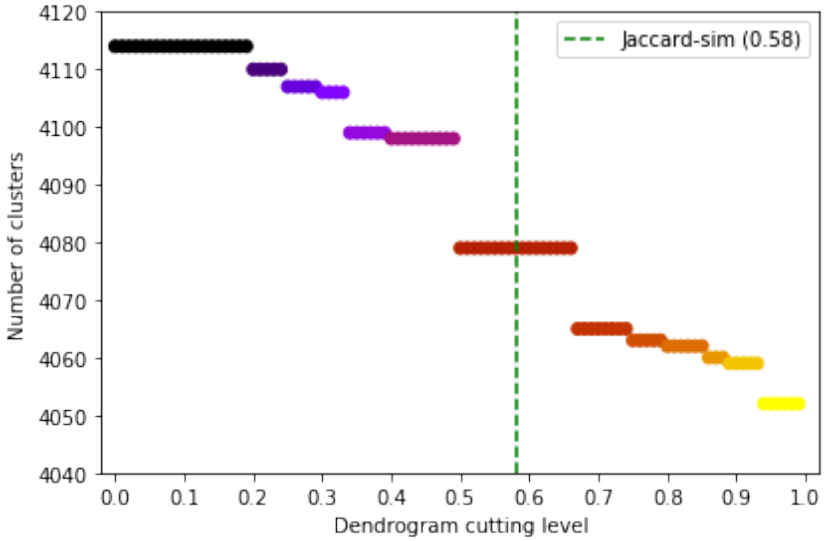


Figure 3.1: Number of clusters from HCA for given dendrogram cutting level h

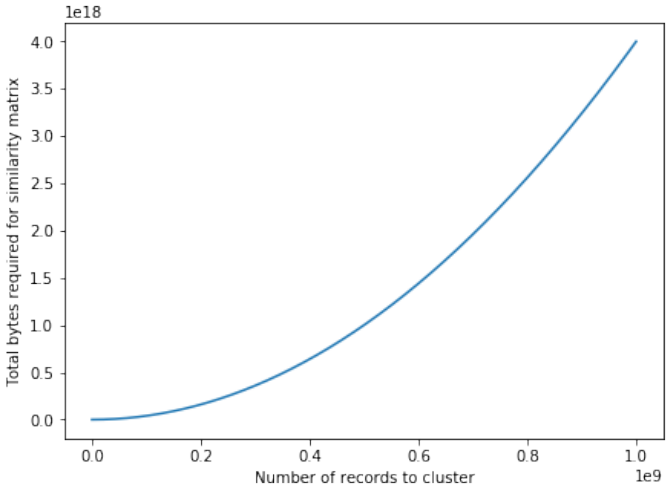
The second flat region revolves around the cutting threshold of 0.58 within the graph. Although this cutting level is lower than the threshold discussed by Perdisci et al., it is the second largest flat region in the graph indicating some form of *natural clustering* of the data set. This dissimilarity between the results of the cutting level h in this case study and the results described by Perdisci et al. [3] might be related to the difference in the characteristics of the data set used. The data used by Perdisci et al. consists of DNS records gathered using *pDNS*. As previously elaborated in Chapter 1.2, there consist many fundamental differences in an *aDNS* or *pDNS* data set. We argue that this difference of a *pDNS* data set and an *aDNS* data set caused the variation in the threshold value that we determined and the value specified by Perdisci et al.. The difference in the overall characteristics of these data set is likely the cause in the variation in the resulting thresholds. Given the results from the example data and graph, we determine the cutting threshold of 0.58 to use in the clustering algorithm for this case study.

3.1.3 Impracticality of HCA for high dimensional data

Although *FluxBuster* use HCA, the implementation of this algorithm does contain significant drawbacks. Especially the necessity of the similarity matrix required by HCA impose some severe practical restrictions. These restrictions exist because the similarity matrix is required to contain the similarity index of every possible combination of the input subject, this results in the actual memory size of the similarity matrix growing exponentially for each added subject. HCA is, therefore, a viable method for clustering smaller sets of data, but becomes impractical for bigger data sets due to the size of the matrix.

$$b = \frac{r \times (r-1)}{2} \times 8 \tag{3.2}$$

Using Equation 3.2, it is possible to determine the memory size in bytes b required for the similarity matrix for records r to cluster. It states the number of bytes required for storing a condensed similarity matrix, when using an 8 byte float variable for containing the similarity index between two records, for the r number or records to use in the clustering algorithm. Figure 3.2 indicates the memory requirements for the number of records r ranging from $1e4$ till $1e9$. The data indicates that when the number of records exceeds $1.000.000$, there are going to be practical difficulties in implementing and executing this algorithm based on the currently available hardware of modern computer systems. Also, because the requirement of a single similarity matrix exists, it is difficult to distribute the calculations of this algorithm across several computing nodes.



Num. records	Num. bytes
10.000	381.43 MB
100.000	37.25 GB
1.000.000	3.64 TB
10.000.000	363.8 TB
100.000.000	35.53 PB

Figure 3.2: Overview of bytes required for storing the HCA similarity matrix given a number of records r to cluster

This requirement for a similarity matrix for the HCA algorithm makes it an impractical algorithm to use with high dimensional data sets. It also indicates that the studies which have used the HCA algorithm were limited to significantly smaller data sets than those available by the *OpenINTEL* platform. The dimensionality of the data set of *OpenINTEL* does result in the unattainable goal of implementing the same clustering algorithm of the *FluxBuster* detection method for clustering relevant domain names. The size of the resulting similarity matrix becomes too large for it to be of practical use. We note that it can be argued that by gathering a data set from *OpenINTEL* that is similar in size to the data set used by Perdisci et al. [3], we can make a comparison while still using HCA. However, the study revolves around implementing a detection method on the novel DNS data set of *OpenINTEL*, not using all the available data within *OpenINTEL* influences the result of that study. Therefore, we choose to implement a different clustering algorithm that results in similar clusters as HCA, but that does not contain a data segment that grows exponentially, and thus allows for the possibility to use it for the *OpenINTEL* data set.

3.2 Clustering algorithm for high dimensional data

We determine that the HCA clustering algorithm that is used by *FluxBuster* cannot be applied to the *OpenINTEL* data set due to practical limitations caused by the memory size requirements of the similarity matrix. Therefore an alternative clustering algorithm has to be chosen that results in similar clusters as the HCA algorithm but which we can apply to high dimensional data sets. One of the requirements of this new clustering algorithm is that, if required by the computational specifications, it should be able to distribute the algorithm across a cluster of processing systems. The University of Twente has an Apache Spark cluster available that

we can use to execute these types of algorithms. Apache Spark[1] is a processing engine for large-scale data processing, and that uses other large-scale processing applications such as Hadoop, Mesos, HBase, and HDFS. It is possible to develop applications for Apache Spark using various programming languages such as Scala, Python, and R. Given the availability of this processing cluster, the clustering algorithm, and the subsequent detection method should be able to be deployed on this Apache Spark cluster of the University of Twente. When choosing a replacement clustering algorithm, we take into account whether there exists a readily available implementation for Apache Spark that has already proven itself in academic use. In general, the new clustering algorithm should fulfill the following requirements:

R_1 Similarity determined by Jaccard similarity

R_2 Ready to use implementation for Apache Spark

R_3 Resulting in similar clusters as the HCA clustering algorithm used by *FluxBuster*

Given these requirements, we identified multiple algorithms that we can use to cluster high dimensional data sets. Requirements R_1 and R_2 could be verified by performing an online search. The results of this verification are available in Table 3.3. We verify the R_3 requirement once a given clustering algorithm fulfills R_1 and R_2 and when we can implement it on the *OpenINTEL* test data set.

Name algorithm	R_1	R_2
DIMSUM[17]	✓	×
Latent Dirichlet allocation	×	✓
Locality Sensitive Hashing[15]	✓	✓
KMeans	×	✓

Table 3.3: Overview of available clustering algorithms

Given our requirements and the possible clustering algorithm as shown in Table 3.3, the Locality Sensitive Hashing algorithm is the only algorithm that fulfilled the requirements R_1 and R_2. For this reason, we further investigate Locality Sensitive Hashing to determine whether or not its result are similar to the HCA algorithm as defined by R_3 given the same Jaccard similarity threshold of 0.58.

3.2.1 Use of Locality Sensitive Hashing algorithm

The Locality Sensitive Hashing (LSH) algorithm is a clustering algorithm that reduces the dimensionality of high dimensional data and determines relevance between data sets using a hashing function. The hashing algorithm that LSH uses determines how the similarity between records is defined and so which item is related to other items. In contrast with cryptographic hashing algorithms, the hashing algorithms used by LSH are developed to result in *collisions* of similar items. LSH reduces the dimensionality by using an appropriate hashing algorithm to hash the input items into various buckets. The resulting buckets are an estimation of the potential clusters within the data set.

For the LSH algorithm to generate results similar to HCA, it requires a hashing algorithm that can determine the similarity of data items by calculating the Jaccard similarity of the records, in our case the related IP set corresponding to the FQDNs. A hashing algorithm suited for this task is the *MinHash* algorithm which is a technique to estimate the similarity of two data sets. The *minhash* function is a replacement for the Jaccard similarity because the probability distribution of the *minhash* function for two data sets equals the Jaccard similarity for those sets; this stated by:

> **Jain and Dubes [15]**, The probability that the minhash function for a random permutation of rows produces the same value for two sets equals the Jaccard similarity of those sets.

[1] https://spark.apache.org/

The LSH algorithm with the use of the minhash function is, in theory, a suitable replacement for the HCA based algorithm used by Perdisci et al. [3]. So that it should be able to provide similar clustering results, but it also should handle high dimensional data sets. This feature of the LSH algorithm is also described by Koga et al. [18], which developed a variation on the LSH algorithm for which the results have shown that the use of LSH resulted in similar clusters as those obtained by HCA and that it has run faster for more sizable data sets. Because LSH has the property of reducing high-dimensional data sets into smaller sets, we further analyze LSH to determine the suitability for the analysis of the *OpenINTEL* data set.

3.2.2 Implementation of Locality Sensitive Hashing algorithm

We analyze the LSH algorithm, with a combination of the *minhash* function, using an open-source implementation of the algorithm that is publicly available on Github[2]. This implementation of the algorithm is based on the description of the algorithm by Jain and Dubes [15] and is suitable for the Spark cluster of the University of Twente. This implementation of the algorithm allows for configurational changes to alter the functionality of the LSH algorithm as is shown in Equation 3.3. In which Z is the list of initial data vectors, (p, M, r, b, F) are the configuration options and $C = \{C_i\}_{i=1...l}$ is the resulting set of clusters.

$$LSH_{(Z,p,M,r,b,F)} = C \tag{3.3}$$

The configuration options of the LSH algorithm significantly influence the results; listed below is an elaboration of the options of the algorithm. It is paramount to find the correct values for options M, r, b to generate clustering results relevant to the HCA algorithm.

- p a prime number greater than the largest vector index.
- M the number of ”bins” to hash data into.
- r the total number of times to minhash a vector.
- b how many times to chop r. Each band has r/b hash signatures.
- F a post-processing filter function that excludes clusters below a threshold.

To be matched as a candidate pair, the signatures of two records should match in all the rows of at least one band. The r and b parameters of the LSH algorithm influence the probability of this happening. The actual probability Pr for a specific *minhash* threshold s is determined by Equation 3.4. We use this equation to determine the probability Pr for two candidate records to be paired for a given Jaccard similarity threshold s for using the LSH parameters r and b. Using the LSH S-Curve Equation 3.4 it is possible to determine the probability Pr of two candidate pairs with Jaccard similarity s of becoming a candidate pair.

$$Pr = 1 - (1 - s^r)^b \tag{3.4}$$

Using Equation 3.5 it is possible to determine the threshold value for the specific r and b values. The threshold is the value of similarity s where the chance of becoming a candidate pair is 50%. Records with a similarity greater than the threshold have a higher chance of becoming candidate pairs, while records with lower similarity are unlikely to become pairs.

$$t = (\frac{1}{b})^{\frac{1}{r}} \tag{3.5}$$

3.2.3 Validation of Locality Sensitive Hashing algorithm

To determine whether LSH is a suitable replacement clustering algorithm for the HCA clustering algorithm used by *FluxBuster*, we should verify the results of both algorithms to determine

[2] https://github.com/mrsqueeze/spark-hash

if they are equivalent. To validate requirement R_3, we run the HCA and LSH algorithm on two small subsets of the *OpenINTEL* data set for which we then verify the similarity of the resulting clusters. The two subsets consist of a data set containing 1.000 records and a data set containing 10.000 records from *OpenINTEL* respectively. We use both the HCA algorithm and the LSH algorithm to cluster the records in the data set for various thresholds. We then use the results to determine the coverage of the LSH algorithm for the results of the HCA algorithm, basically, how much percent of the clusters generated by the HCA algorithm is identical to the clusters generated by the LSH algorithm. We consider clusters from both algorithms equal if the FQDNs listed by the clusters from both cluster algorithms are an exact match.

We show the results of the comparison in Table 3.4. The results indicate that the resulting LSH clusters have very high coverage of the HCA clusters; this means that the results of both the LSH and HCA algorithm are almost identical. However, we note that the LSH algorithm always generates more clusters than the HCA algorithm. Due to time constraints, we did not identify the cause for these outliers. In general, we found that since the cluster coverage is 99.01% or higher, the LSH algorithm fulfills requirement R_3, and thus we regard it as a valid replacement for the HCA algorithm. So the LSH algorithm is used in this case study to cluster the high dimensional data set of *OpenINTEL*.

	1K subset			10K subset		
Threshold t	0.25	0.50	0.75	0.25	0.50	0.75
HCA	411	409	407	4107	4079	4063
LSH	413	412	411	4137	4116	4110
Coverage	100.0%	99.75%	99.01%	99.70%	99.46%	99.08%

Table 3.4: Results of HCA vs LSH cluster comparison

3.2.4 Threshold for Locality Sensitive Hashing algorithm

As determined, the LSH algorithm is a viable replacement for the HCA algorithm. However, before we can fully implement the LSH algorithm, we should also determine the threshold for the LSH. Using the r and b parameters of the LSH algorithm with the values $r = 10$ and $b = 210$ respectively, Equation 3.5 shows that the threshold for the LSH algorithm is $t = 0.585$. For these values, it is possible to plot the probability that two candidates match for a specific Jaccard similarity s. We show in Figure 3.3 the corresponding LSH S-curve using the previously defined Equation 3.4. The figure shows the overall probability of two records being candidate pairs for a specific Jaccard similarity; the vertical line shows the specified threshold value of 0.58. The graph also indicates the areas that resemble both the false-positive (FP) and false-negative (FN) rate for the specific Jaccard similarity. We configure the algorithm so that the FP-rate is larger than the FN-rate. We choose this configuration because we calculate the exact Jaccard similarity for the specific clusters after the clustering of the LSH algorithm, so any clusters with a lower similarity than 0.58 are detected and discarded. So minimizing the FN-rate is more important than preventing FPs; therefore we deem the parameters as $r = 10$ and $b = 210$ as sufficient.

The algorithm also requires configuration option p, which we automatically calculate depending on the size of the input data set. We determine the p parameter as the smallest prime number larger than the input size. The post process filter parameter F determines the required minimum size of resulting clusters; thus the algorithm discards any cluster with a size smaller than F. There does not exist a flux-network consisting of a single system; therefore, we decide to use a minimum size requirement of $F = 2$ in the LSH clustering algorithm.

The M parameter determines how many *bins* are used by the LSH algorithm. In accordance with the r parameter, this determines the maximum number of clusters that can be generated given $Clusters_{max} = M \times r$. As experiments have shown, an M parameter with smaller value results in a limited number of clusters, none of which have a Jaccard similarity that is greater or equal to 0.58. We determine that this is because the number of clusters formed by *natural clustering* within the data set, are higher than the maximum number of clusters that

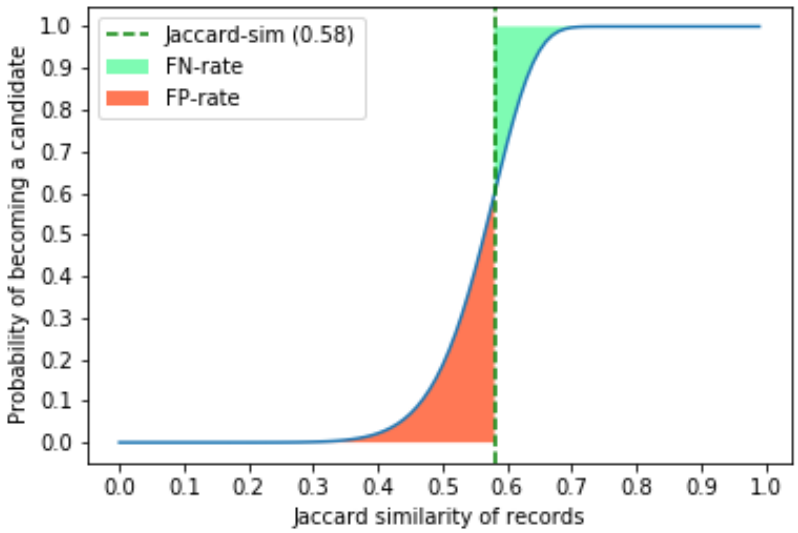


Figure 3.3: LSH S-curve, $N_r = 10$ & $N_b = 210$ using Equation 3.4

can be generated by the LSH algorithm. This behavior, in turn, results in unrelated pairs being processed into the same cluster, resulting in a very low Jaccard similarity. We decide to use a value of $M = 5.000.000$ for the LSH clustering algorithm, and results show that this value is extensive enough to accommodate the appropriate number of clusters within the data set. We further substantiate this by the fact that the number of resulting clusters is less than the maximum number of clusters and because sufficient clusters have an actual Jaccard similarity that is greater than 0.58.

3.3 Defining the ground-truth

The classification process that is described by *FluxBuster* defines for each cluster whether it is benign or malicious depending on several characteristics of the DNS responses in the *pDNS* data set. In Table 3.1 an overview of the characteristics that Perdisci et al. used are shown. As we previously described, implementing the exact classification methodology of *FluxBuster* takes too much effort which will limit us in properly analyzing and verifying the actual results of the detection application. We, therefore, decided to implement a simpler detection methodology which we base on a known malicious ground-truth of malicious domain names related to the Netherlands TLD. Implementing a detection methodology based on this method results in that we have ample time to analyze the available data to answer the research questions mentioned in the introduction of this document.

3.3.1 Ground truth

Previous studies [3, 5, 11] have already used known ground-truths in the context of flux-network detection, either as input for the classifying algorithm, as an additional verification of potential malicious domains or as a validation of the eventual results. An overview of the sources for the ground-truth used by the various flux-network studies, as defined by Stevanovic et al., is shown in Table 3.5. We note that the sources for the ground-truth in the various studies do not contain any specific information regarding observed flux-networks. The sources that the researchers used contain information regarding domains and IP addresses that have shown some form of malicious behavior; for example, a domain that is mentioned in a phishing mail, a known C2-server or a domain which has spread malicious software. The reason that there are no known flux-network ground-truths is because implementing an agile network using DNS is not malicious per se; it is the purpose of a flux-network that makes it malicious. We, therefore, argue that not every domain name or IP address listed within the ground-truth are actually provisioned or maintained by a flux-network and that therefore it should be expected that these

types of publicly available ground-truths likely result in FP for the detection of flux-networks. However, the focus of this research is not to develop a novel detection mechanism which requires a specific TP rate but rather an analysis of the detection characteristics of using a novel *aDNS* data set. Therefore, we do not require that the ground-truths should be utterly related to known flux-networks. Also, we argue that due to increased law-enforcement activities on taking down large malicious networks, flux-networks have become increasingly used by malicious actors. This behavior, in turn, can be used as an argument that at least a subset of the ground-truths is related to flux-networks in some form.

Study	Training/Evaluation	Blacklist
Perdisci et al. [3]	Evaluation	`abuse.ch` (FQDN) - 75 flux 2LDs 12 public blacklists (two of them stated): `malwaredomains.com` (FQDN) `malwarepatrol.com` (FQDN)
Bilge et al. [5]	Training and Evaluation	`domains.com` (FQDN) `zeustracker.abuse.ch` (FQDN and IP) `malwaredomainlist.com` (FQDN) `wepawet.cs.ucsb.edu` (FQDN) A set of Anubius reports (FQDN) `phishtank.com` (FQDN) `siteadvisor.com` (FQDN) `safeweb.norton.com` (FQDN)
Choi and Lee [6]	Evaluation	`kisarbl.or.kr` (FQDN) `malwaredomains.com` (FQDN) `cyber-ta.org` (FQDN) `siteadvisor.com` (FQDN) `mywot.com` (FQDN) `domaincrawler.com` (FQDN) `spamhaus.org` (FQDN and IP)

Table 3.5: A subset of the overview of the labeling practices used by some of the most well regarded contemporary DNS-based detection methods, as described by Stevanovic et al. [16]

3.3.2 Used ground truth

As shown in Table 3.5, numerous studies use a wide range of public data sets for the matter of identification of malicious domain names. This case study uses only DNS data sets that are related to the Netherlands ccTLD and which have been detected in 2017 and are related to second-level domain names such as `example.com`. Due to these requirements, we only use domain names from the ground-truth that have data available that comply with these restrictions. We take the sources described by Stevanovic et al. [16], shown in Table 3.6, as a starting point to create our ground-truth. The ground-truth that we gathered consists of 15265 entries, each of the entries had been identified in the year 2017, are all related to the Netherlands ccTLD (.nl) and only contain 2-level FQDNs except for the *www.* 3LD.

We use the generated ground-truth for the identification of malicious flux-network clusters in the *OpenINTEL* data set. We perform the identification by verifying whether the domain name of the cluster exists in the ground-truth. During the identification process, domains names are compared to the malicious domains in ground-truth as is; we make no changes to either the domain name of the cluster or of the ground-truth. We consider a cluster malicious if at least two domain names in the cluster are listed in the ground-truth. By definition, a flux-network has multiple domain names associated with it to decrease overall detection as stated by Nazario and Holz [19]. Because a multitude of domain names are required, we use a minimum hit count of two for detecting flux-networks. Because we store the ground-truth hits and domain names in the results, it is possible to filter out any false positives in the analysis eventually.

Source	Category	Number
malwaredomains.com	FQDNs	41
zeustracker.abuse.ch	FQDN	1
ransomwaretracker.abuse.ch	FQDNs	155
sslbl.abuse.ch	FQDNs	2
www.malwarepatrol.net	FQDNs	40
NetCraft	FQDNs	15026
Total	FQDNs	15265

Table 3.6: List of sources used for the ground-truth

3.4 Detection of flux-networks

In this section, we describe the general steps of the flux-network detection mechanism used in this case study. We show a general graphical overview of this process of clustering and identification in Figure 3.4.

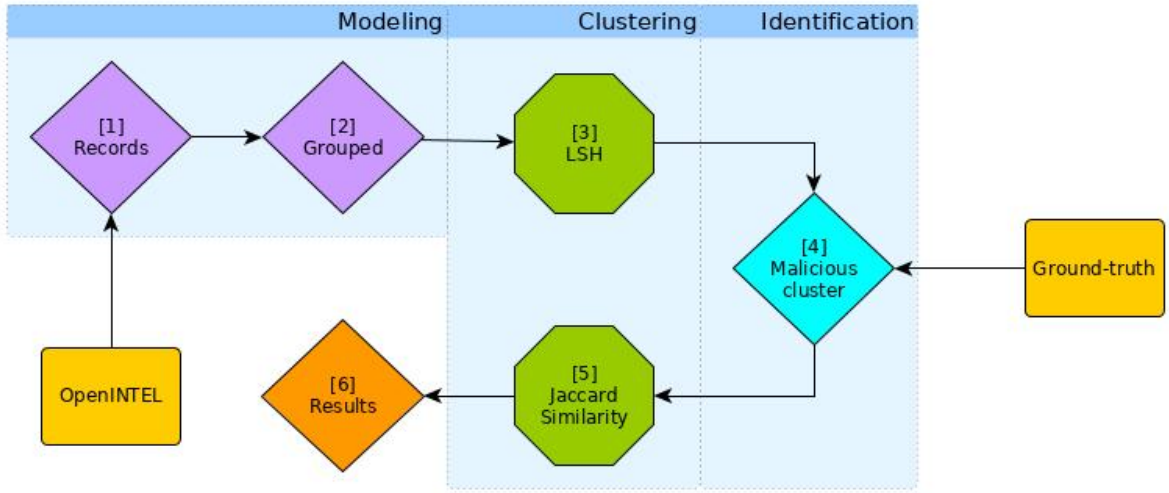


Figure 3.4: Flux-network identification process

The initial step of the process is gathering records from the *OpenINTEL* data set for a specific period. We perform the analysis for this case study on the Netherlands ccTLD (.nl) for the year 2017. The analysis starts by segmenting the data into one week time periods for the entire year 2017 because it is impractical to analyze the entire year in one run. This segmentation means that for a single week data records are gathered from *OpenINTEL* and the flux-network application is used to identify malicious flux-networks in that particular *OpenINTEL* data segment. An example of the data record[3] gathered from *OpenINTEL* is shown in Table 3.7.

FQDN	IPv4	date
inglesmundial.com.	104.25.94.7	15/03/2017
inglesmundial.com.	104.25.94.8	16/03/2017
likenhanh.net.	103.28.38.229	15/03/2017
likenhanh.net.	103.28.38.229	16/03/2017
www.likenhanh.net.	103.28.38.229	15/03/2017
www.likenhanh.net.	103.28.38.229	16/03/2017

Table 3.7: *OpenINTEL* records

[3]Due to agreements between *OpenINTEL* and SIDN, the organization responsible for the Dutch TLD, we do not mention Dutch domain names in this paper. Therefore examples from other TLDs are used.

The records gathered from *OpenINTEL* are not usable by the LSH clustering algorithm. So to use the records by the LSH algorithm, the second step of the process consists of modeling the data into tuples containing the FQDN and the set of related IP addresses. This tuple is then used to cluster relevant FQDNs using the LSH clustering algorithm; we show an example of the data format in Table 3.8. A set is used to process IP addresses because the LSH algorithm does not contain functionality to determine the significance of an IP address if it is used multiple times by the same domain name. The clustering algorithm can only determine the relevance of two FQDNs when the set of two IP addresses related to the two FQDNs are similar. The upside of this requirement is that duplicate data is discarded resulting in less data that we need to process.

FQDN	IP set
inglesmundial.com.	{104.25.94.7, 104.25.94.8}
likenhanh.net.	{103.28.38.229}
www.likenhanh.net.	{103.28.38.229}

Table 3.8: *OpenINTEL* records grouped

The third step of the process is to cluster relevant FQDNs together based on the Jaccard similarity of the related IP addresses. The LSH algorithm is used to cluster the FQDNs which have a high probability of having similarity equal to or greater than 0.58. We show an example of the resulting clusters in Table 3.9.

Cluster ID	Entries
1	{inglesmundial.com. : {104.25.94.7, 104.25.94.8}}
2	{likenhanh.net. : {103.28.38.229}, www.likenhanh.net. : {103.28.38.229}}

Table 3.9: *OpenINTEL* records clustered

Once we identify the clusters, we perform the categorization of malicious clusters in the fourth step by validating the FQDN in the clusters with the known malicious FQDN in the ground-truth. We mark every cluster that matches at least two times FQDN to the known ground-truth as malicious. Because we only require flux-networks for this case study, we discard any clusters that we do not mark as malicious. The result of the fourth step is a significantly smaller subset containing only malicious clusters with a high probability of being similar.

After the identification, the fifth process step is to validate the exact similarity of the malicious clusters. This procedure is performed to ensure that every resulting cluster has at least a 0.58 similarity. Until this process step, we cannot guarantee this because the results of the LSH clustering algorithm is an estimation of the available clusters within the data set. Therefore, the chances also exist that two unrelatable records end up in the same cluster. So for every cluster that is marked malicious, we also calculate the definite Jaccard similarity to ensure that we discard any cluster with low similarity. This verification step is the most processing-intensive task of the entire mechanism and is therefore only performed after the identification of malicious clusters to ensure that application executes this task on an as small as possible set of data. Experience has shown that determining the exact Jaccard similarity before the identification, and therefore resulting in the calculation of the similarity for much more clusters, increases the overall running time significantly. Using Equation 3.6, we calculate the similarity of the entire cluster C by diving the intersect and union of all combined FQDNs n within the cluster. We discard any cluster that does not meet the requirement of at least a 0.58 similarity. This verification process is the final step of the detection application; the results of this application are then made available on the HDFS platform in JSON format.

$$sim(C) = \forall n \in C \frac{|n \cap n+1|}{|n \cup n+1|} \tag{3.6}$$

3.5 Verification of flux-networks

The result of the detection application is a list of clusters that have a similarity higher or equal to 0.58 and which have at least two domain names listed in the ground-truth. As previously specified, there is no guarantee that the entries in the ground-truth are directly related to actual flux-networks; the ground-truth may also contain references to other malicious behavior not facilitated by flux-networks. Due to this fact it is apparent that we need to verify the results of the detection application to determine if the results are either IP-flux or domain-flux networks. In the following sections, we elaborate on the methods that we use to determine whether the resulting clusters are actual IP-flux or domain-flux clusters.

3.5.1 Identifying networks

One of the characteristics of both IP-flux and domain-flux is the fact that the properties related to those networks, such as domain names and IP-addresses, frequently change to avoid detection. So when we identify IP-flux or domain-flux clusters, it is essential that we can verify whether the properties of those clusters frequently change over an extended period. This type of behavior, including other specific characteristics elaborated in the following sections, is then used to identify flux-networks correctly. So, the purpose of this process is to combine several separately detected clusters into the same network; a network is thus a combination of clusters from different periods that show some form of similarity.

Unfortunately, we have not seen this type of approach in previous researches. The reasoning for this is unclear since other studies have also used specific time segments in which they detected flux-networks. So, they would also need to link several clusters together to get a better overview of the characteristics of the entire network from start to finish. Due to the lack of previous research, we can use no prior implementation to resolve this particular issue. The general approach which we use to determine these networks is to ascertain a commonality that could be used to identify clusters from the same network and then use an algorithm to combine the most similar clusters into the same network.

We expect to base this commonality on the individual characteristics of the clusters; potentially we can base this on the IP-addresses, domain names or hits in the ground-truth associated with the cluster. The exact commonality is determined once we analyze the actual results and we can determine which property of the cluster is suited for this comparison.

The difficulty in identifying these networks is that it is unknown when a specific network starts or ends, what the relevant size is and which clusters are part of this network. However, using graph theory, it is possible to identify the appropriate networks by using the clusters as vertices and weighted edges as the commonality between those clusters. It is then possible to use shortest path algorithms, such as Dijkstra's algorithm, to combine clusters into the appropriate networks. Such an implementation has been described by Khalil et al. [20] which used a conforming implementation of graph theory with Jaccard similarity weights for edges.

So a directed weighted graph $DG(C, E)$ where C are known malicious clusters and the edges $e = \{c_1, c_2\} \in E$ are created. The weight of a certain edge e is denoted as $w(c_1, c_2)$ and it reflects the commonality between both clusters. Because we use Dijkstra's algorithm to create a network of clusters, the commonality values are inverted to be compatible with Dijkstra's algorithm. Dijkstra's algorithm generates networks with the shortest path; thus the path with the lowest weights, so an edge that indicates two completely overlapping clusters should consist of the weight 0.0 instead of the default Jaccard similarity of 1.0. The use of Dijkstra's algorithm results in networks that always consist of clusters with the greatest similarity possible. For the algorithm to give the proper results, it is required that there does not exist an edge between very vertices in the graph because if this were the case, it would become possible for completely dissimilar cluster to grouped into the same network simply because a path would exist. So when we determine the commonality between the clusters, we also need to determine a minimum similarity that is required before we can create the weighted edges. We can derive this value from analyzing the actual results from the detection application.

The result of Dijkstra's algorithm is a list of networks containing clusters that we have detected in various periods, which share a minimum required commonality so that we determine that the clusters are part of the same network. These networks are then used to verify whether the clusters changes overtime as to verify whether the clusters are actual flux-networks.

3.5.2 IP-flux detection

Based on the behavior and properties of the cluster, we are going to classify a detected cluster as an IP-flux, domain-flux or non-flux network. In this section, we are describing the process that we are using to identify IP-flux networks. Different studies related to flux-network detection have used various detection methods for identifying flux-networks that show IP-flux behavior. Although the exact detection method differs between the researches, in general, the method is quite similar in which most detection methods are looking for the variation of IP-addresses associated with the cluster.

For example, the *FluXOR* detection mechanism by Passerini et al. [7] looks into the number of distinct networks, autonomous systems (AS), assigned network names and organizations related to a potential flux-network candidate to determine whether the network is benign or malicious. Passerini et al. actually show an example in which popular benign domain such as *hp.com* and *www.avast.com* are easily identifiable as benign because every IP-address points to the same AS and organization; whilst another example, given as *www.factvillage.com* resolves to three IP-addresses that are hosted by three separate networks and organization thereby identifying this malicious behavior.

The researches of the *FluxBuster*[3] detection mechanism use a similar approach with the same principle that a great variety of used networks used by a single domain name is a sign of IP-flux behavior. However, in contrast with the approach of the *FluXOR* detection method which does require additional resources in gathering the AS, network and organization names related to the IP-address, Perdisci et al. performed this distinction of networks related to the IP-addresses by just analyzing the /16 prefix of the IP-addresses. They did specify that mapping the IP-address to the appropriate AS or BPG-prefix increases the overall accuracy but that using the /16 prefix was sufficient in their study and that using this method removed the computational burden of receiving the additional AS information. This distinction of the IP-addresses based on the /16 prefixes has also been done by other studies to determine whether there was a significant *distance* between the IP-addresses, such as the study by Nazario and Holz [19]. Perdisci et al. specified the analysis of the /16 prefix as follows:

> **Perdisci et al. [3]**, The rationale behind these features is that, unlike in the case of CDNs or other legitimate services, flux agents are often scattered across many networks located in many different countries, thus increasing the number of IP addresses that do not share a common /16 prefix.

Both studies have shown the importance of determining the variation of used networks associated with the cluster to identify IP-flux behavior. In general, the higher the variety of used networks the more likely it is that the detected network is an IP-flux network. In this case study, we are using the approach of Perdisci et al. to filter out clusters that have multiple IP-addresses associated to it but which use the same /16 prefix, indicating that they still point to the same network. This behavior of using IP-addresses which all point to the same network is contradictory to actual IP-flux behavior, so we are not going to categorize clusters that show this type of behavior as IP-flux candidates. The clusters that use numerous different networks are still IP-flux candidates, so we analyze these clusters to determine further whether the cluster does show any other IP-flux characteristics. For these clusters the same methodology is used as in *FluXOR* detection mechanism, so we are using manual verification to determine the number of distinct networks, AS, network names, organization names for that specific cluster.

If we can determine that several hosting providers, of the IP-addresses associated with the cluster, are located in various countries, we use this as another indication of whether a cluster is an actual IP-flux network. We know that malicious networks spread their infrastructure among several countries to increase the difficulty for law enforcement agencies to start an investigation

and to take appropriate actions. This behavior is mainly due to the legislation which limits a law enforcement organization to investigations directly related to their appropriate country. A malicious network hosted by a provider located in multiple countries requires cooperation between the law enforcement agencies of all the affected countries and thus dramatically increases the difficulty in starting an actual investigation. The study by Cooke et al. [21] already mentions this type of behavior which increases the overall difficulty in taking down malicious networks.

This expected behavior of IP-flux networks is further enforced by the fact that hosting content in various countries is currently no more difficult than hosting it in a single country. Due to this reasoning, we determine which countries are related to the IP-addresses of the clusters. We use the GeoIP databases of Maxmind [22] to verify in which countries the IP-addresses of the cluster are hosted. GeoIP is currently the defacto standard for this type of geolocation based lookup of an IP and is widely used in previous relevant researches [23, 24]. As mentioned by *Maxmind*, the accuracy of looking up the city level address is currently insufficient; however, determining which country hosts the IP should be accurate enough for this case study. We note that during the analysis the used *Maxmind* database is more recent than the data that we verify using these databases; in general, this difference in time is approximately one year. Online resources state that the current *Maxmind* databases should still be sufficient to verify the countries associated to the IP-addresses since the lack of free IPv4 IP-ranges results in very few ISP still trading their assigned prefixes, which means that information from the *Maxmind* databases is sufficient to verify the current results.

To further identify a potential IP-flux network, we compare the clusters to the other clusters within the same network according to the methodology described in Section 3.4. This action aims to verify the change in IP-addresses associated to the overall network. As previously stated by Nazario and Holz [19], an IP-flux network rapidly changes its associated IP-addresses. Therefore we can assume that an actual IP-flux would not use the same set of IP-addresses over an extended period. By comparing the IP-addresses associated to the cluster, which shows IP-flux behavior, against the IP-addresses of clusters related to the same network, we determine whether the network uses the IP-addresses for a long or short period. Any network that uses the same set of IP-addresses over an extended period is not categorized as an IP-flux network simply because this type of behavior is contradictory to IP-flux behavior. We perform this validation by determining whether the IP set of the clusters in the same network is a subset of the IP-addresses of the cluster with IP-flux behavior. If we identify more than 2 clusters with IP-addresses that are subsets of the IP-flux of the cluster, the cluster is no longer regarded as IP-flux because this identification concludes that throughout a period of three weeks a similar set of IP-addresses was used. This behavior is not related to IP-flux behavior, and therefore we regard such a cluster non-IP-flux network. We regard any cluster that has not been filtered out during this final process as an actual IP-flux network.

3.5.3 Domain-flux detection

In a similar manner on how we identify IP-flux networks by analyzing the characteristics of the cluster, we are using the same methodology for domain-flux networks. There are several criteria that we use to determine whether a cluster is domain-flux or a non-flux network which we are basing on relevant studies.

Domain names used by domain-flux networks are generally generated by DGAs, which we describe in Section 1.2.3. The domain names generated by a DGA usually have different attributes than benign domain names, previous studies [16, 5] used this variation in attributes to identify domains generated by DGA. In general domain names generated by DGAs consist of more random characters than common domains because these domains do not suffer from the deficit that a regular user should be able to use them; thus these malicious domain names do not have to be *logical to use*. Because of this type of behavior, previous researches have all focused on the lexical analysis of domain names to identify malicious domain names generated by DGAs.

There have been numerous studies that focus on the analysis of this type of malicious domains

to detect flux-networks. Researches that implemented some form of lexical analysis were studies by Stevanovic et al. [16, 25] and by Bilge et al. [5]. Interestingly enough, the detection method of *FluxBuster* did not focus on the lexical analysis of the domain names whatsoever and thus was unable to detect malicious domain names used by DGA's. Therefore it is required to look into the other researches for the proper method for detecting these types of domain names.

As expected the main focus of the identification process for these type of domains revolves around the lexical analysis of the domains itself to distinguish between randomly generated domains and legitimate ones. There have been other researches [6, 10] that show other methods in detecting these type of malicious domains. However, the detection methods that they have implemented, for example, analysis of client-side behavior, are incompatible with the current case study and are therefore not included. Overall, there is some form of consensus in the various studies on which attributes of the domain names to focus to identify randomly generated domains. Mainly the focus resolves around verifying the number of vowels, consonants, and digits in a domain and retrieving the overall length and getting the number of dictionary words in the domains. The consensus is that domains generated by a DGA contain overall more digits in a domain, have greater length but contains less actual dictionary words, Stevanovic et al. [16] describes these specifications as:

> **Stevanovic et al. [16]**, Pseudo-random domains are characterized by smaller number of words within them. ... Pseudo-random domains are characterized by higher number of numerical characters.

The *FluxBuster* detection algorithm does not contain any detection method which we can mimic to detect domain-flux networks. However, another researched detection method *FluxOR* [7] does contain a method which we can apply to this case study. This method revolves around gathering specific statistics of the analyzed features, which we show in the following list for this case study, for both benign and malicious domains and then compare the results. This verification can both be performed manually or by using a classifier algorithm. This method does require a set of statistics from benign domain names which we use to validate the statistical results of the potential malicious domain names. The goal of this analysis is to identify clusters that show outlier behavior of the average statistics of the domain names in the cluster when compared to the benign domain names. The statistical properties that we gather for both benign and malicious domain names for the benefit of domain-flux detection are:

- D_1 Number of vowels in domain name
- D_2 Number of consonants in domain name
- D_3 Number of digits in domain name
- D_4 Length of domain name
- D_5 Number of words in domain name

Determining the number of words in a domain name Although we can gather the majority of properties of the domain names easily by analyzing the specific domain name, the specification of how many dictionary words are within the domain is more difficult. For the proper detection of domain names generated by DGAs, it is vital that we determine the correct value for D_5 for each domain name in the clusters. Domain names generated by DGAs are randomly generated that it will contain very few to none dictionary words within them. For example, a long domain name which contains many vowels and consonants can either be a domain name generated by a DGA or just a large domain name if it also contains a high number of dictionary words.

Determining this property is also difficult because there is no guarantee that the domain names that we analyze only consist of Dutch words. It is a common fact that many Dutch people are bilingual and are proficient in both Dutch and English; therefore, it is not uncommon of domain names located in the Netherlands TLD to contain English words. Due to this reasoning, the

analysis with regards to the number of actual dictionary words located in a domain name focuses on both the Dutch and English language.

The problem with this type of analysis is that domain names do not contain a proper segmentation of words because this type of behavior is prohibited by the DNS protocol, so domain names do not contain the usual segmentation of spaces or special character except the special character dash. The analysis should, therefore, be able to determine the number of words based on a single unsegmented string. A practical example of this challenge is transforming the single unsegmented string such as *thegreatestadventureever* into multiple dictionary words: *the*, *greatest*, *adventure*, *ever*.

There are software modules available that do provide this type of functionality of segmenting an unsegmented string into multiple words. The used software module should, however, be able to use both an English corpus for segmenting the string as well as being able to use a corpus of an additional language. Given these requirements and the authors experience with the Python programming language, the software module used in this case-study is *WordSegment*[4]. This module allows for the segmentation of the domain names into several words to calculate the D_5 statistic. We must note that this software module only supports analyzing alphabetical character, so the software does not take the dash character in domain names into account. This module comes with a ready to use English corpus required to perform the actual segmentation of the English language; however, using Dutch segmentation is not possible by default. *WordSegment* requires a corpus for a specific language to be able to segment a string into multiple words properly. A corpus has to consist of both a dictionary of *unigrams*, a list of words with the number of occurrences of the specific word, and a dictionary of *bigrams* which is a list of combination of two words and again the number of occurrences of the specific combination.

We gather a large data set containing Dutch written text, that was made available by *Instituut voor de Nederlandse Taal*, from a Dutch newspaper [26]. This specific data set contains numerous news articles spanning multiple years of journalism and which describe various topics. Since the articles are describing a variety of topics and the data set contains data from years of publications, it is a suitable data source to generate the Dutch Corpus. Thus we use this data set to generate a Dutch corpus for the *WordSegment* module. A sample of the resulting *unigram* and *bigram* values from the Dutch corpus is shown in Listing 3.1.

```
[('de', 126104), ('in', 55417), ('het', 50542), ('van', 47899), ('een', 47688)]
[('in de', 12981), ('van de', 12764), ('in het', 5441), ('voor de', 4981)]
```

Listing 3.1: Small sample of the unigram and bigram generated to form the Dutch Corpus

This word segmentation module with both the provided English corpus and the generated Dutch corpus is used to segment domain names in the appropriate words. This word segmentation module segments the initial input into multiple words but it does not perform any filtering on actual dictionary words, this means that if we combine two actual words with an unknown word the segmentation module will results a list of three words. So to verify if the results of the segmentation module are actual dictionary words, we also look up every word in actual dictionaries. Therefore, we also create a data set of both a Dutch dictionary from *OpenTaal* [27] containing 164.313 entries and an English dictionary containing 370.098 entries. Only words that exist in either the Dutch or English dictionary are regarded as actual words and are used to determine the average number of words in each domain for a specific cluster.

Defining the baseline To be able to detect outliers in the generated statistics for the domain names in the clusters, there needs to be a baseline for the statistics of benign domain names. In a similar fashion as was done by Passerini et al. [7] in the *FluxOR* study, the baseline consists of the statistics $(D_1, D_2, D_3, D_4, D_5)$ generated from known benign domain names. The list of known benign domain names consists of the top 10.000 most popular domain names

[4] http://www.grantjenks.com/docs/wordsegment/

as was published by *OpenDNS*, a popular public DNS server. For each type of characteristic of the domain name, we calculate both an average and standard deviation, these statistics we then calculate for all 10.000 domain names available within the baseline. Both the average and standard deviation of the statistics of the benign domain names are then used to verify the resulting statistical data of the malicious clusters and are used to identify any outliers.

We perform actual classification of a cluster whether it is benign or a domain-flux cluster if the cluster shows outlying behavior compared to the benign baseline over an extended period. So once we identify a cluster that shows outlying behavior for the gathered statistics, the general statistics of the network to which the cluster is related is also analyzed. So only if it revealed that over an extended period the network shows extraneous behavior when compared to the baseline, the network and the corresponding clusters are identified as domain flux-network.

Chapter 4

Results

Using the described methodology and configurational options, we developed a Spark application for the detection of flux-networks that uses the 2017 data set of *OpenINTEL*. We created the specific implementation of this detection mechanism in the Scala programming language, specifically for the Spark framework; the source code of this application is shown in Chapter B.1. We sequentially execute the detection mechanism for specific time periods. These periods consists of 1-week segments starting at *01/01/2017* and ending at *31/12/2017*, thus resulting in 53 segments of which only the last segment does not contain seven days. We choose these segments to both minimize the processing time required for the Spark application and to increase the data points in a single execution.

The analysis of the entire data set of *OpenINTEL* for the year 2017 took roughly 1.5 weeks to finish. In that time, the entire *.nl* TLD namespace stored in *OpenINTEL* was analyzed meaning that a total of $3.951.904.173$ records were analyzed, clustered and identified. We show the exact number of analyzed records, IP-addresses, and domains in Table 4.1. The application resulted in the detection of $7.969.946$ clusters, of which 322.164 were identified as malicious because at least two domains of the specified cluster were listed in the ground-truth. Finally, of those clusters, 97.285 also satisfied the requirement that all components of the clusters have a Jaccard similarity of at least 0.58. These clusters are the final result of the Spark application, and these resulting clusters are the data set which we analyze further in this case study.

The characteristics gathered from the detection mechanism are shown in Figure 4.1. We show in the top left graph the overall number of unique FQDNs and IPs. Both the overall record count as well as the IP addresses and domain names stored in the *OpenINTEL* data set show a steady increase. We expect his pattern as part of the continuing increase of registered domain names and used IPs on the Internet. The exciting aspect of the number of records available in *OpenINTEL* is the fact that there is a sudden dip in the overall number of domain names in *OpenINTEL*. It is unknown what the cause is of this decrease; however, due to time constraints, we did not investigate further.

The graph in the top right of Figure 4.1 indicates the number of clusters that we detected. We both show the total number of clusters, the number of detected malicious clusters, and the number of detected malicious clusters with Jaccard similarity of at least 0.58. The strange dip in the number of unique FQDNs shown in the top left graph did not result in a corresponding

Description	Number
Num. *OpenINTEL* records	$3.951.904.173$
Num. grouped IP's	$599.239.358$
Num. domains	$557.278.830$
Num. clusters	$7.969.946$
Num. malicious clusters $sim >= 0.00$	322.164
Num. malicious clusters $sim >= 0.58$	97.285

Table 4.1: General characteristics of Spark job analysis

dip in the number of detected clusters for every category. The last graph in the bottom left corner is the total number of records that are available in the *OpenINTEL* data set. This graph shows similar behavior to the graph depicting the total number of unique FQDNs and IPs in the data set, thus showing an overall steady increase.

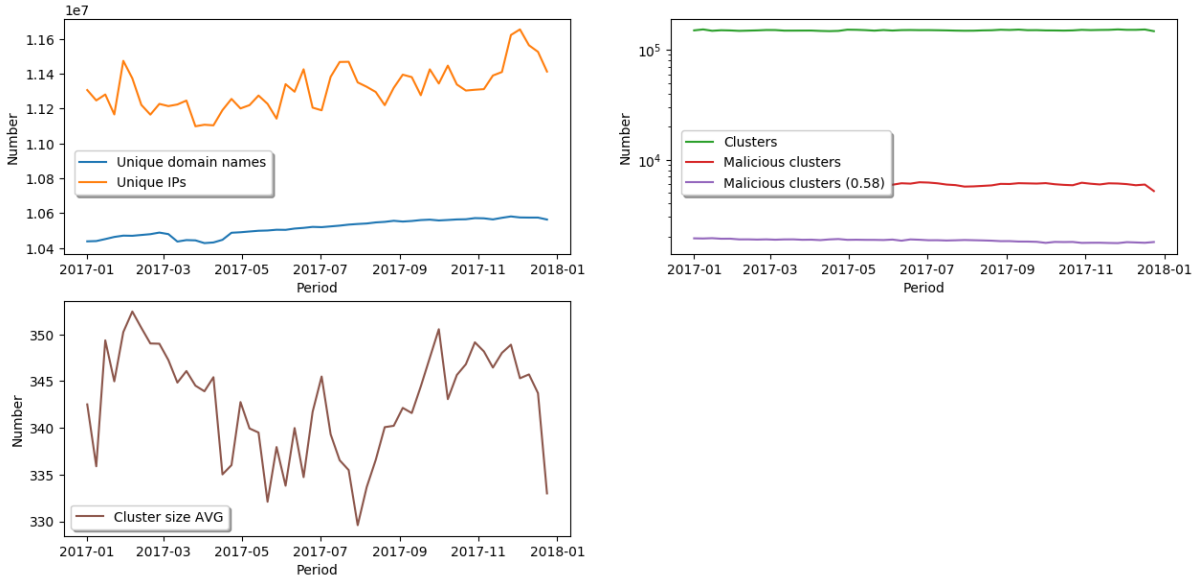


Figure 4.1: Statistics from the Spark jobs

The overall number of clusters that we identified and the average Jaccard similarity score of each segment are shown in Figure 4.2. Given the number of identified clusters, we indicate that there exists a steady decline of detected malicious clusters. Only at the end of the year, there is again an increase in the overall detected clusters. It is unknown what the exact origin is of this sudden change in the number of detected clusters. Another detail shown is the high similarity of the identified clusters, which is rather remarkable and unexpected when looking for flux-networks. The lowest Jaccard similarity average for a weekly segment is 0.9992, indicating that even in this segment the majority fo clusters have a similarity score of 1.0. Such a high score indicates that the vast majority of FQDNs associated with the cluster all pointed to the same IP-address. There are also periods in which the average Jaccard similarity is 1.0, indicating that there has not been a single cluster that did not have a complete overlap of IP-addresses with the domain names. Investigating the results does indicate that some clusters have Jaccard similarity close to the actual minimum requirement of 0.58, but still, the majority of the clusters have a 1.0 similarity score. We attribute this behavior to the fact that the vast majority of clusters only have 1 IP-address associated with it which automatically results in the Jaccard similarity score of 1.0.

4.1 General characteristics of clusters

We need to determine whether we identified flux-networks properly; therefore, we create a general overview of the global characteristics of the various clusters. The two main attributes of the clusters are the list of domain names and IP-addresses associated with the cluster. The overall sizes of the domains and IP-addresses related to the clusters are shown below with additionally the deviation, min and max values:

This overview shows that there is an unexpected significant difference between the domain name and IP-address attributes. Mainly that the vast majority of the clusters only have 1 IP-address associated with it, which is unexpected behavior when the clusters are suspected of being a flux-network. As mentioned, the small variation in the number of IP-address does clarify the reason for the overall high Jaccard similarity score, as shown in graph 4.2. Also, it is noteworthy that the number of domain names associated with the clusters differs widely. As an example, there are clusters with only 1 IP addresses associated with it while the domain

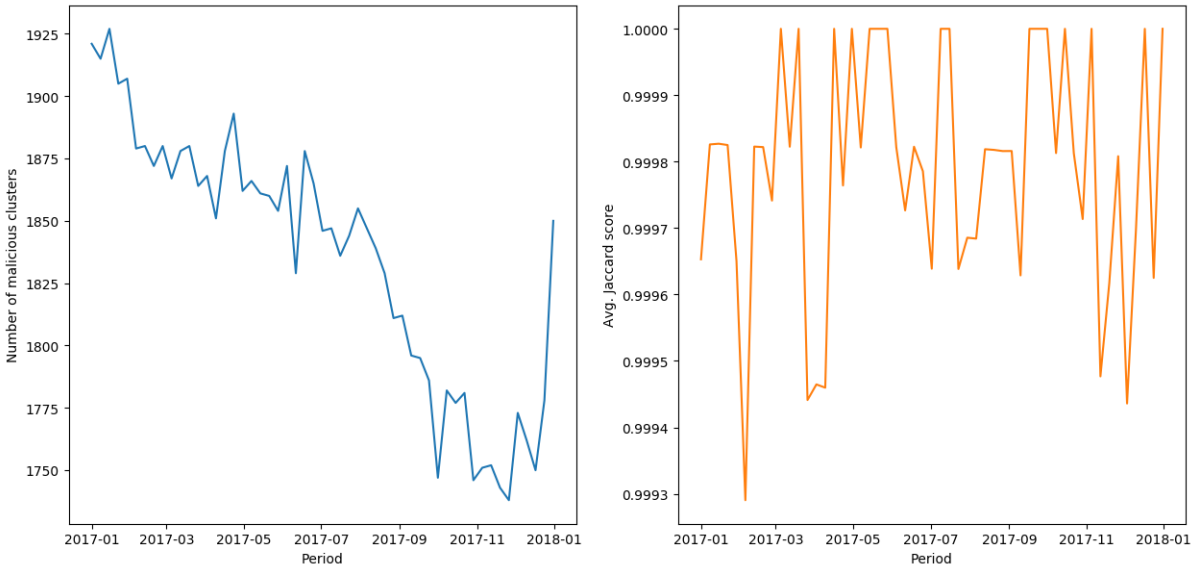


Figure 4.2: General characteristics of the identified malicious clusters

Domain names	
mean	1090.203392
std	4030.448724
min	3
max	515,816

IP addresses	
mean	1.011060
std	0.173785
min	1
max	24

Table 4.2: General characteristics of domain names and IP-addresses of all identified clusters

names related to those clusters can be either 3 or $515,816$. The significant differences within the overall characteristics of those clusters increase the difficulty in analyzing the results.

We generate an overview of the domain names characteristics for the different IP-addresses sizes; we show the results in the graph of Figure 4.3. This graph shows several histograms of the domain-list sizes categorized by the IP-list sizes ranger from the smallest detect IP set size of 1 to the largest 25 in increments of 5. We choose these increments as they show an appropriate segmentation of the IP-addresses list sizes of the clusters in the graph. So each graph shows on the x-axis the size of the domain-list and the y-axis the number of clusters. Each graph shows the appropriate clusters with an IP-list size range shown above the graph. This figure also indicates that the majority of clusters have very few IP-addresses associated with it. There are very few clusters with more than 5 IP-addresses associated with the cluster. Also, the graphs show that currently there is no distinguishable divergence in the number of domain names related to the cluster depending on the size of the IP-list.

It becomes clear that with the great division of the attributes of the malicious clusters, it becomes impossible to identify any flux-networks quickly. We, therefore, deem it required to perform some form of classification of the different types of clusters to identify a cluster with characteristics more commonly associated with flux-network. As a practical example, a malicious cluster with only 1 IP-address and only 3 domain names associated with it is not a flux-network, and we can discard it.

4.2 Cluster categorization

Besides the overall variation of the cluster attributes, a substantial number of clusters have outlying properties when compared to the majority of the clusters. These clusters further increase the difficulty in identifying potential flux-networks. We attribute these outlier clusters mainly to either tiny or huge clusters. The smaller clusters only have a small number of either IPs or FQDNs thus making it very unlikely that they are flux-networks. While the large

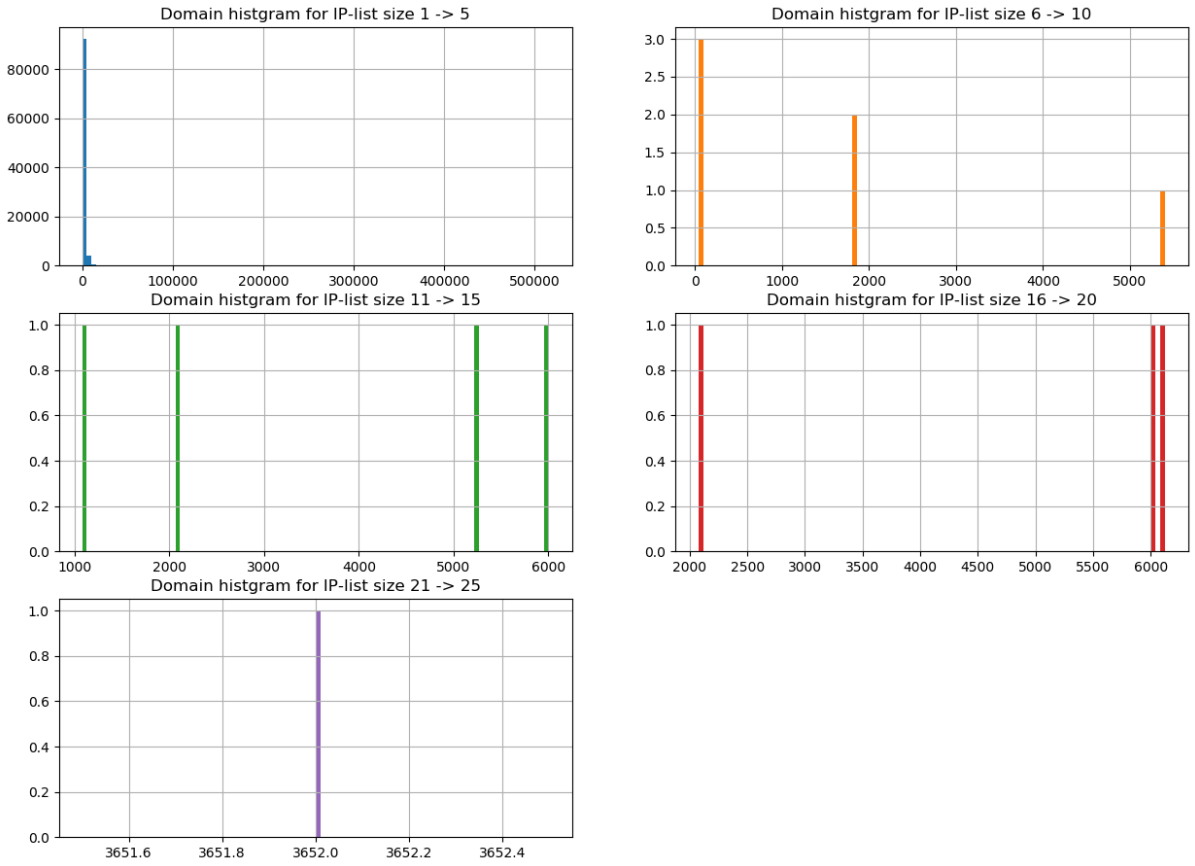


Figure 4.3: Overview of domains list size histograms in various IP size categories

clusters are likely to be part of some form of shared hosting that results in many thousands of FQDNs associated to the same cluster with only a tiny percentage of the FQDNs are listed in the ground-truth. The overall variation of the size of these clusters dramatically increases the difficulty of efficiently analyzing these clusters mainly due to processing and memory requirements. To increase the efficiency of the analysis some form of classification process should be implemented to classify these outliers and categorize them accordingly. In general, we use this classification process to identify clusters with similar properties and classify them in the same category. We expect the result of this process to be a separation of the clusters divided by the general characteristics of the various categories. This process is primarily aimed to categorize the overall cluster results which we then use to increase the efficiency of the actual analysis. We ensure that every cluster is still analyzed thoroughly in this case study to determine whether flux-networks exists in the various categories.

Because the number of clusters with more than 1 IP-addresses associated with it, is limited, the main properties used to classify the clusters is the domain name set size, hit-size, and hit-coverage. The hit-size is the number of domain names listed in the ground-truth and hit-coverage is the percentage of the number of domain names that are found in the ground-truth compared to the total number of domain names associated to the cluster. These properties show the most distinct differences in the various categories. As an example of the overall fluctuation of the properties of the outlying clusters, we show an overview of these properties in Table 4.3, which shows both relatively large and small clusters with the corresponding properties.

id	domain-size	IP-size	hit-size	hit-coverage	score
9	1,074	1	2	0.18622	1.0
20175	3	1	2	66.666667	1.0
71475	515,816	1	13	0.00252	1.0

Table 4.3: Clusters with outlying properties

4.2.1 Categories of clusters

We are going to classify the clusters into four distinct classes, three of the classes we are using to classify outlying clusters that have a smaller chance of being flux-networks. We are going to use the fourth category to classify all clusters which we have not identified as an outlier cluster meaning the clusters that have the most significant chance of being a flux-network. We primarily focus on this category because in general, it has the highest potential of actually containing flux-networks. We are going to thoroughly analyze every cluster within the various categories to detect for flux-networks, so this classification procedure is not going to be used to filter out any outlying cluster.

The initial focus of the classification process is to categorize relevant clusters with similar outlying properties into the same category. Examples of outlying clusters are clusters with either a relatively high or low number of associated domain names when compared to the overall majority. These types of clusters are going to be classified into several separate categories, mainly because the category of the clusters with large domain sets has such a high computational impact that separating the categories are going to benefit the overall efficiency of the analysis. Furthermore, it is important to note that because we are going to analyze every cluster within the categories, we will not fully statistically substantiate the overall criteria for each category. The criteria of the various properties being either the size of the domain set or the hit-coverage where chosen for the categories based on manual inspection of the entire data set. We use the following clusters categories to classify the current data set:

Category: 1 **Small sized clusters**
This category consists of clusters with very few FQDN records associated with it. Thus mainly clusters that only have a low number of hits in the ground-truth and an overall minimal number domain names associated with it. Due to the small number of domains associated with the cluster, these clusters usually have a relatively high hit-coverage.

Category: 2 **Normal clusters**
This category is used to identify all clusters that we did not identify as either abnormally large or small. These clusters are therefore most likely to consist of flux-networks.

Category: 3 **Normal clusters with low hit-coverage**
This classification is a more specific classification of the clusters which we categorized as category 2. So it contains all the clusters which we did not identify as either abnormally large or small, but clusters that still have a distinguishable low hit-coverage. We base this classification in a separate category on the assumption that the hit-coverage of the cluster is so considerably low that it is difficult to categorize the entire cluster as malicious. Manual inspection indicates that this property exists in normally sized clusters that only have the minimum required hits in the ground-truth resulting in very low hit-coverage. As a delimiter for this particular category, we use a limitation of less than 0.5% hit coverage. This specific value was selected because it showed a clear segmentation of the different clusters during manual verification and because we determine that a hit-coverage of only 0.5% is thus drastically low that any conclusion that the entire cluster is malicious is going to be difficult to substantiate.

Category: 4 **Large sized clusters**
We use this classification for clusters with an abnormally large number of domain names associated with it. Generally, we label clusters that consist of several tens of thousands of domain names as category 4. Due to the large numbers of domain names associated with these clusters, they also contain a very small hit-coverage.

4.2.2 Selection of classifier algorithm

We are using a classifier algorithm for the labeling of the clusters into the cluster categories. To use a classifier algorithm, we are going to supply a subset of the data with an appropriate classification. This data set is then used to both train and verify the classification algorithm. The actual ratio of the division of the data set between training and testing differ depending on

the application. In most cases, multiple ratios are used ranging from $90\%, 80\%, 70\%$ for training the algorithm and the remaining subset used to verify the results. Most studies begin with either an 80/20 or 70/30 ratio to start with; we decide that we are initially using 80% of the data set to train the algorithm and the remaining 20% of the data to verify the results.

To use a classifier algorithm we need to make a labeled subset of the data available, which requires a manual classification process to categorize the clusters accordingly. For this particular case study, we use 1% sample of the total data to train and verify the classifier, thus creating a subset with a total of 973 clusters. We randomly select this subset from the total data set, and manual verification did indicate that it contains clusters that we can classify as average clusters or cluster outliers that contain either very small or large domain sets. We manually categorize every cluster within this sample to one of the categories previously discussed. We performed this inspection by verifying the number of domains and IP associated with it, validating the overall score, the number of hits in the ground-truth and the overall hit coverage.

Similar to the selection of the training and testing data set ratio, the best classifier algorithm to use also depends on the exact application of the algorithm. The most suited classifier algorithm for a specific case is determined similarly to the ideal ratio of the data set, mainly by executing various algorithms with various ratios and validating the results. For each execution of the algorithm, we verify the actual performance and validate the results given the test data set. The initial classifier algorithm which we test is the *RandomForest* [28] classification algorithm with an 80/20 ratio of training and testing data set. It is noteworthy to mention that the segmentation of the entries in the subset, so the entries that are going to be either the training and testing data set are randomly chosen during every test. After running multiple tests verifying the classifying results of the algorithm, we noticed that the *RandomForest* algorithm classified all clusters 100% correctly during every test. We argue that these results are related to the fact that the different categorization between the clusters is pretty distinct and which is based on only a small subset of properties. This behavior is further substantiated by the importance of each cluster property as indicated by the algorithm as shown in Table 4.4. The table indicates that the main property used for classification are both the size of the domains-list and the hit-coverage; in lesser terms, the classifier also uses the number of hits in the ground-truth. The table also indicates that both the IP set size and score property are not taken into account by the classifier. We expect this behavior since there is not much variation in these cluster properties. We decide not to test any other classifying algorithm, or different data set ratios since our application of the *RandomForest* algorithm with the 80/20 ratio results in the most optimal outcome possible; namely that the algorithm has identified 100% of the testing data set correctly.

Property	Classifier importance
ip_size	0.0006893600545678496
domain_size	0.4485818385211231
score	0.0
hit_coverage	0.47836247697287887
hit_size	0.07236632445143013

Table 4.4: Table indicating the importance of the various cluster properties for the classifier algorithm

4.2.3 Classifying results

Given the cluster categories, the classifier algorithm and the subset of already categorized clusters, we can identify the clusters appropriately. We show the classification of the data used to train and test the algorithm in Figure 4.4. This figure indicates in the leftmost graph the overall clusters within the data set with the number of domain names related to the clusters on the y-plane and the number of hits in the ground-truth in the x-plane. The middle graph shows the same data but colorized depending on the classification. This graph indicates the

large clusters in the top of the graph in purple and the smallest clusters in yellow in the lower left corner. We clearly show the difference between the standard clusters and the standard clusters with very low hit coverage in the right graph of Figure 4.4. This graph still uses the same y-plane but uses the hit-coverage as the x-plane. Please note that in contrast to the other graphs, the x-plane of this graph is an indication in percentage instead of the number of domain names. Albeit that the x-range is similar, maxing 50 in the amount in the left and middle graph and max 50 percentage in the right graph. The right graph indicates the difference between standard clusters in green and standard clusters with very low hit coverage in red. A clear distinction between the clusters in the various categories is shown in Table 4.5, that shows a small random sample of clusters related to that specific category and the properties of those clusters. This distinction shows that there are definitive differences in the cluster properties between the various categories.

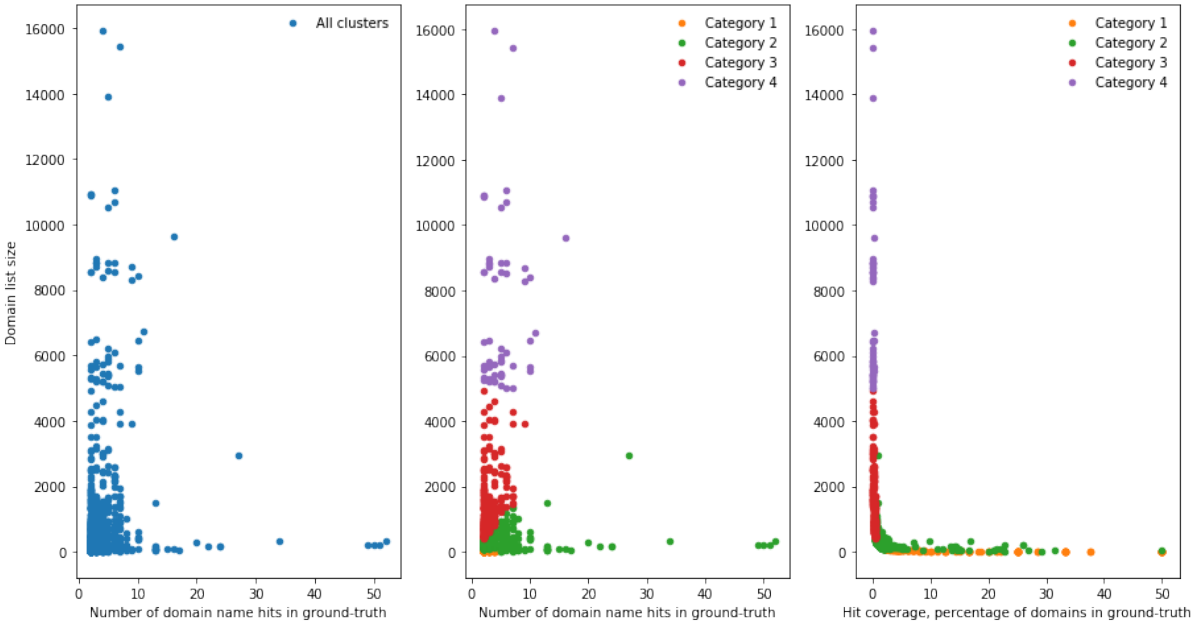


Figure 4.4: Visualisation of subset of data used for training classifier

The same training data set and the same algorithm is used to classify all 97.285 clusters within the entire data set. The classification algorithm classified 21.608 as category one, 40.880 as category two, 29.129 as category three and 5.668 as category four. We show the difference in the number of clusters related to the various categories in Figure 4.5. We do not use the classification results to filter out any category in the effort of trying to identify flux-networks; however, these results are used to prioritize which cluster is most likely to be a potential flux-network.

4.3 Identifying networks

As predicted, manual verification of the identified clusters and the results of the categorization procedure, described in the previous Chapter, does indicate that many clusters show similar or equal characteristics to other clusters in different periods. As we previously described, we expected this behavior since the detected clusters are likely part of a network that does not merely exist in a single week in which the *OpenINTEL* detection system has detected it, but we expect that the clusters exist for a far longer time. We attribute this behavior to all types of malicious and benign networks, so to make a proper characterization of clusters, all relevant clusters of the same network should be known.

Using the method described in Section 3.5.1, we are going to group relevant clusters into the same network to use these results for the proper identification of either IP-flux or domain-flux networks. The only aspect that was not defined yet is the commonality that we are using to determine to group cluster into similar networks. We elaborate on the property that we use for the commonality between the clusters in the following section.

	Category 1			***small sized clusters***		
id	domain_size	ip_size	hit_size	score	category	hit_coverage
73494	16	1	2	1.0	1	12.500000
46016	18	1	4	1.0	1	22.222222
15427	78	1	2	1.0	1	2.564103
	Category 2			***normal clusters***		
id	domain_size	ip_size	hit_size	score	category	hit_coverage
134786	780	1	4	1.0	2	0.512821
15260	392	1	10	1.0	2	2.551020
75671	281	1	20	1.0	2	7.117438
	Category 3			***normal sized clusters with low hit-coverage***		
id	domain_size	ip_size	hit_size	score	category	hit_coverage
112641	1112	1	4	1.0	3	0.359712
133697	738	1	2	1.0	3	0.271003
61592	453	1	2	1.0	3	0.441501
	Category 4			***large sized clusters***		
id	domain_size	ip_size	hit_size	score	category	hit_coverage
27550	8575	1	5	1.0	4	0.058309
131725	15447	1	7	1.0	4	0.045316
68059	6722	1	11	1.0	4	0.163642

Table 4.5: Sample of clusters of each category indicating the category properties

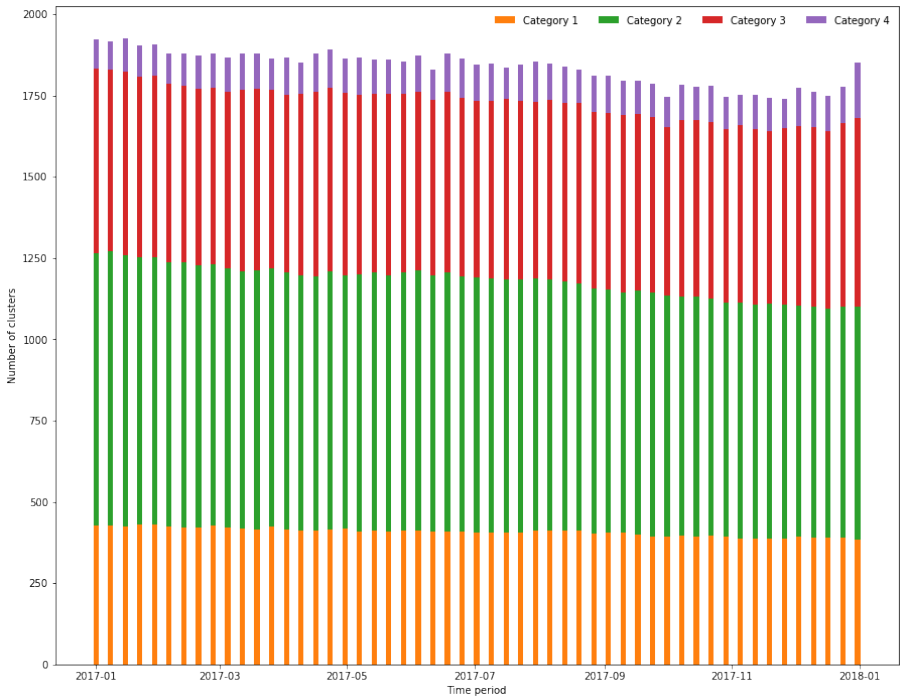


Figure 4.5: Overview of the number of clusters attributed to the categories

start	end	hit_list	ip_list	score
2017-03-26	2017-04-01	$domain_A$ $domain_B$	138.201.31.229	1.000000
2017-04-02	2017-04-08	$domain_A$ $domain_B$	138.201.31.229 104.31.68.31 104.31.69.31	0.666667
2017-04-09	2017-04-15	$domain_A$ $domain_B$	104.31.68.31 104.31.69.31	1.000000
2017-10-01	2017-10-07	$domain_A$ $domain_B$	104.31.68.31 104.31.69.31	1.000000
2017-10-08	2017-10-14	$domain_A$ $domain_B$	138.201.133.218 104.31.68.31 104.31.69.31	0.666667
2017-10-15	2017-10-21	$domain_A$ $domain_B$	138.201.133.218	1.000000

Table 4.6: Example of a network changing its properties over time

4.3.1 Defining network commonality

The most critical aspect of relating similar clusters is choosing the property of the clusters that we are using to determine the commonality between the various clusters. The properties of the clusters that would most likely be relevant to link clusters together would be either the IP-addresses or hits in the ground-truth. The IP-addresses associated to the clusters are not very diverse in most cases, as can be noticed in the fact that the vast majority of clusters only 1 IP-address associated, meaning that during the detection there was no other IP-address linked to the FQDNs in the *OpenINTEL* data set. The hits in the ground-truth is another property viable for linking consonant clusters together because these properties are expected to last longer than a specific period.

We regard both characteristics to be sufficient to identify relevant clusters; however, over time the exact properties of both the IPs and hit list can change. Manual verification of the data shows that over time a network can change the associated IP-addresses due to either benign or malicious actions. Given the situation, we made an initial setup that uses a combination of these two properties. The theory is that although the IP-addresses and hits in the ground-truth may change over time, it is improbable that a change occurs for both lists in precisely the same period in such a way that the change results in a completely different cluster which we cannot attribute to the network. We show a specific example in Table 4.6, this shows the change of the associated IP-addresses to a network that we recorded by the detection method. This example does show that although the properties of the cluster do change, the overlap ensures that this is still detectable even if we base it solely on the associated IP-addresses.

The actual similarity between the properties of two clusters is determined by using the *Jaccard similarity* for both IP and hit sets. To link consonant clusters, we generate a data set containing the maximum Jaccard similarity of both the IP set and hit set for each cluster within the data set. We determine these values by calculating the Jaccard similarities for each cluster compared to the clusters in the next period and deriving the maximum score as shown in Equations 4.1 and 4.2. These equations show how we determine the highest Jaccard similarity for relevant IPs and hit sets for a set of clusters C and for a given cluster c that consists in period p.

$$sim(c_{ips}) = max(\forall n \in C_{p+1} \frac{|c_{ips} \cap n_{ips}|}{|c_{ips} \cup n_{ips}|}) \tag{4.1}$$

$$sim(c_{hits}) = max(\forall n \in C_{p+1} \frac{|c_{hits} \cap n_{hits}|}{|c_{hits} \cup n_{hits}|}) \tag{4.2}$$

Using Equations 4.1 and 4.2, we determine a maximum similarity for both the IP set and hit

set for each cluster within the data. We show the categorization of these results in Figure 4.6, these statistics indicate the overall division of the Jaccard similarity for both the IP and hit sets. We show that the overall majority of the 89.103 clusters within the data set have identified an equal match, e.g., a Jaccard similarity of 1.0, to both the current cluster IP and hit set to a cluster in the next period. This behavior is the best case scenario meaning that we identified a cluster that has a cluster in the next period containing the same IP set and hit set thus demonstrating a commonality between the two clusters. The second largest group, although significantly smaller, is the number of clusters that have found no match with either the IP or hit sets, basically indicating a network that has ceased to exist. This group contains by definition every cluster detected in the last period since there do not exist any clusters to which we can compare it.

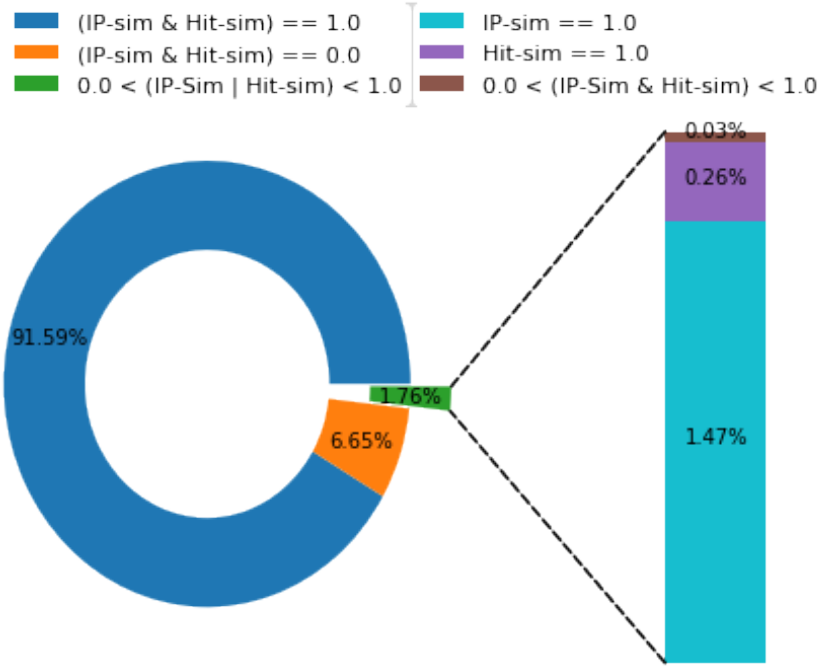


Figure 4.6: Overview of the IP and hit set properties for determining the similarity between clusters

The third group in the left donut chart is the clusters which have not found completely identical or dissimilar clusters for both data properties. This group is further elaborated in the right bar chart in Figure 4.6. This chart indicates clusters that did not identify a completely similar cluster in the next period for both properties, the majority of the clusters still have identified clusters with either the IP set or hit set having the same content. This behavior is expected behavior for example given in Table 4.6, in which the cluster changes the associated IP-addresses during a certain period, resulting in a Jaccard similarity being lower due to the change in IP-addresses but the similarity of the hit set remains the same. For this study, we determine that an exact match for either the IP set or hit set with another cluster is sufficient to relate both clusters to the same network.

The remaining small subset of clusters are clusters that have a partial match for both the IP set and hit set for another cluster. However, the number of clusters that fall in this category is substantially small; this group consists of only 29 clusters which consist of roughly 0.03% of the total number of detected clusters. Although this subgroup is rather insignificant, there should be some form of clarification for these clusters what the exact thresholds are for being related to an existing network.

Figure 4.7 indicates the division of both the IP and hit set properties of the 29 clusters using a violin plot. A violin plot is similar to a box plot as it shows numerical data using the quartiles, but it also indicates the kernel density similar to a histogram. Thus the external body is used to indicate the kernel density estimation, and the inner black bar and the white dot is used

to indicate the quartiles with the white dot indicating the median. Besides the IP and hit set properties, we also use this figure to indicate the *max* property division; this property consists of the highest recorded similarity which is either the IP set or hit set similarity. The figure also indicates that the majority of the IP set similarity is centered around 0.5, while it is visible that the similarity of the hit set is more distributed. We use the maximum value of either sets to create a more concentrated distribution; this new distribution revolves around 0.64 similarity which is a sufficient indication of similarity in this case study. Given the fact that the lowest value for the max property is 0.3333, there should be a distinction whether this similarity is sufficient enough to use as a commonality of clusters within the same network. In this case study, we already encountered a similar issue for which we had to determine the threshold for which two records are related using Jaccard similarity. For this particular issue, we deem the threshold of 0.58 threshold sufficient; using this same threshold, we state that a total of 8 clusters contain insufficient similarity to be related to any known network. This statement means that the overall majority of the clusters can be easily related to either known networks and that only 8 clusters, which are only 0.008% of the total data set, have insufficient similarity to be related to any network.

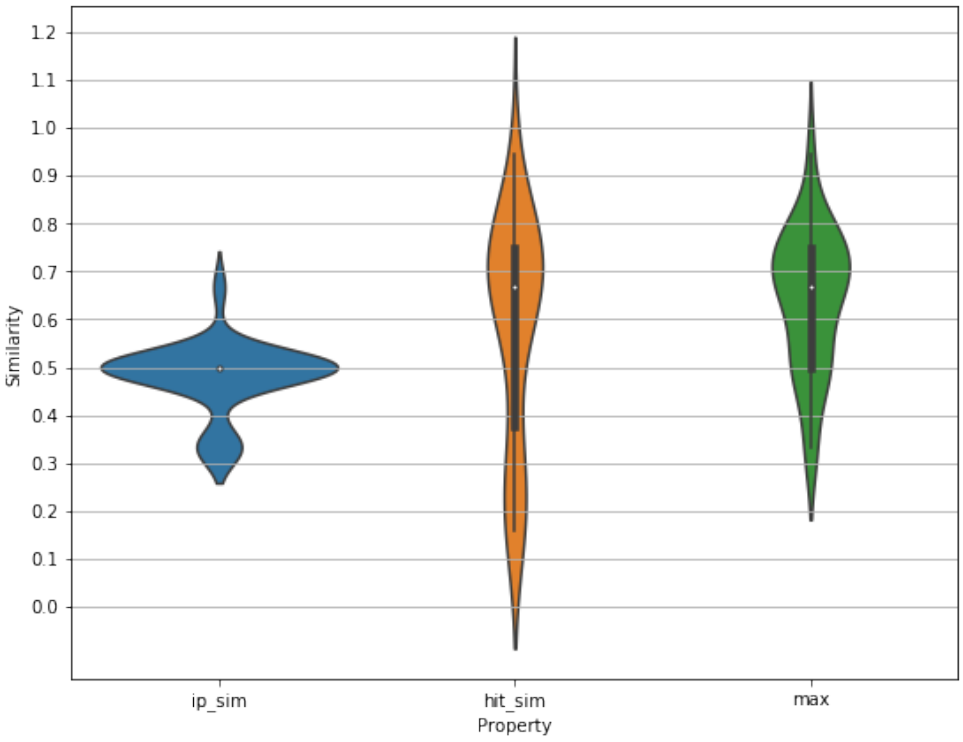


Figure 4.7: General characteristics of the malicious clusters identified

Validation of determining network commonality The method proposed to define the similarity of clusters based on either the hit-list or IP-list has not been previously documented and is, therefore, a novel method. To validate this particular method we use a different data set to determine whether it results in similar statistics. As a testing data set for this purpose, we reran the detection application on the *OpenINTEL* data set but this time in monthly increments instead of weekly increments. This change in the configuration should result in an overall decrease in the accuracy of the particular clusters, but given the described similarity methodology, it should also result in similar similarity distributions.

The application did result in the detection of 19.795 malicious clusters. Using the same methodology to detect similar clusters related to the same network as previously described, we generate an overview of how many clusters did identify an exact or partial match. The results of this overview are visible in Figure 4.8 and in Table 4.7 which show an elaboration of the exact statistics compared to the weekly segments. Distinct in this figure and table is the fact that there is an increase in the number of clusters that do not contain a match whatsoever or only a partial match; however, we expected this type of behavior. In general, the results are

Description	Weekly segments		Monthly segments	
	Number	Percentage	Number	Percentage
IP-sim & Hit-sim == 1.0	89.103	91.59%	15.221	76.89%
IP-sim & Hit-sim == 0.0	6.470	6.65%	3.660	18.49%
0.0 < (IP-Sim | Hit-sim) < 1.0	1.712	1.76%	914	4.62%
IP-sim == 1.0	1430	1.47%	722	3.62%
Hit-sim == 1.0	253	0.26%	165	0.85%
0.0 < (IP-Sim & Hit-sim) <1.0	29	0.03%	27	0.15%
Total	97.285		19.795	

Table 4.7: Table showing the difference of the weekly & monthly segments for determining similarity between clusters

similar to the original data set in which the vast majority of clusters have identified a complete match, that the second largest group have identified no match and that a relatively small group has only a partial match. Because the results of the validation data set are similar to the original data set, we determine that this particular methodology of linking multiple clusters to the same network is valid and we can use it in this case study.

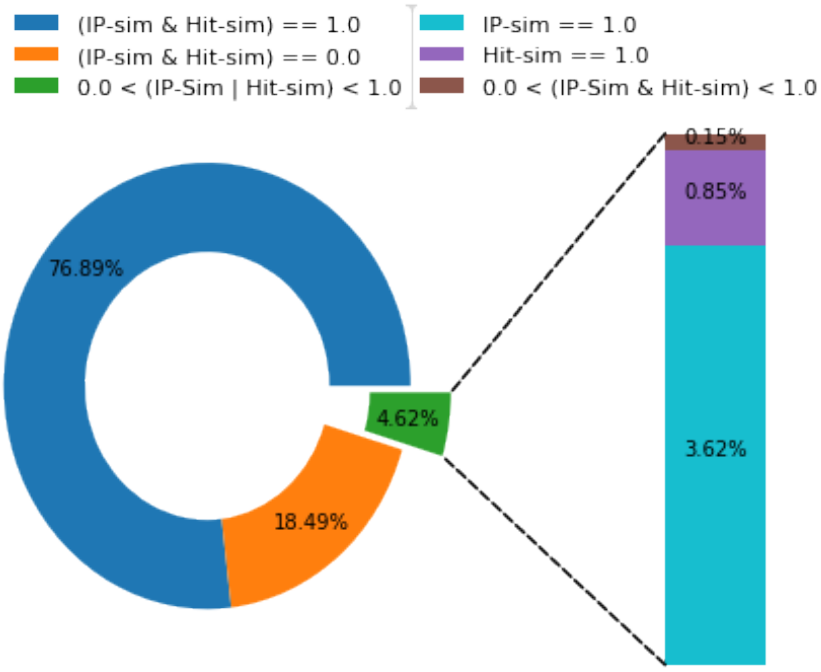


Figure 4.8: Overview of the IP and hit set properties for cluster chaining for the validation data set

4.3.2 Identifying components in networks

So clusters are related to the same network based on the maximum of either the IP set similarity or hit set similarity if the resulting similarity is at least higher than 0.58. The vast majority of the clusters passes this requirement and only clusters with no similarity whatsoever (1.712) or clusters with a similarity lower than 0.58 in both the hit set and IP set, are used to indicate a potential end of a network.

As previously specified in Chapter 3.5.1, we are using a directed weighted graph $DG(C, E)$ that will contain the clusters $c \in C$ and the edges in the form of $e = \{c_1, c_2\} \in E$. The weight of a certain edge e is denoted as $w(c_1, c_2)$ and it reflects the maximum Jaccard similarity that we

calculate between both clusters which are either the similarity of the IP or hit set. As elaborated, there should only exist an edge between two clusters if the similarity is higher than the required minimum. We define the minimum required similarity as 0.58, which we invert to the value 0.42 to be compatible with Dijkstra's algorithm. So if the similarity between two vertices is higher than 0.58 the edge is created otherwise no edge is added to the graph, the definition is shown in Equitation 4.3.

$$w(c_1, c_2) = \begin{cases} 1 - max(sim_{ips}(c_1, c_2), sim_{hits}(c_1, c_2)) & \text{if } w(c_1, c_2) <= 0.42 \\ \text{Inf.} \end{cases} \tag{4.3}$$

4.3.3 Identified networks

Using Dijkstra's algorithm, we discover 5004 networks within the data set. The majority of the networks are identified as either relatively small size or of the maximum size, as shown in Figure 4.9. The largest group of networks 904 have a size of 52 clusters indicating that the network has spanned the entire year for which the mechanism has identified malicious clusters. The second largest majority 756 of networks are relatively small, only consisting of 2 clusters. There does not exist networks with only 1 cluster associated to it, simply because a network has to contain a minimal of 2 cluster before we categorize it as a network. Interestingly enough, we did not generate any network that consists of exact 26 clusters; the reason for this is unclear.

Verifying the similarity of those networks does indicate that the majority of networks (4984) have an identical match to all components in the network thus resulting in an average similarity of 1.0. The resulting networks 20 which do not have a completely identical similarity still have an average 0.944444 similarity. This result indicates that on average there is still a substantial similarity between every component within these networks. Given these results and the fact that the similarity of the gathered networks is relatively high, the results are sufficient enough to use as indicators of larger networks within the data set.

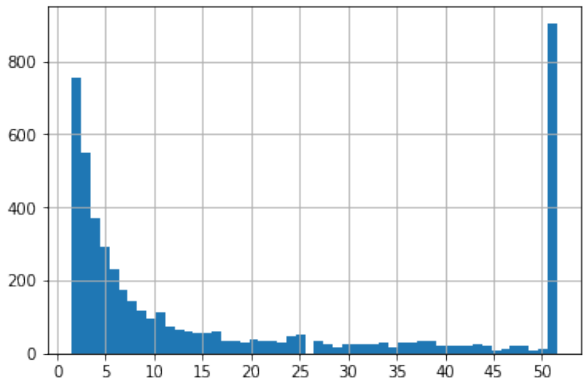


Figure 4.9: Histogram of network sizes

We show a visualization of a randomly selected subset of identified networks in Figure 4.10. The figure shows on the x-axis the entire year of 2017 and on the y-axis, the uniquely identified networks identified by a single row. This figure indicates the detected life span of some of the networks that we identified. In general, it is shown that some networks span the entire year, probably existing before and after the year 2017, but it also indicates that some networks have a clear origin and end in the allocated period.

4.4 Detection of IP-flux

Flux-networks could potentially use domain-flux or IP-flux or both as a defensive measure against the detection of malicious networks. So during this case study, we analyze both the characteristics of domain-flux and IP-flux to determine whether the detected clusters show any

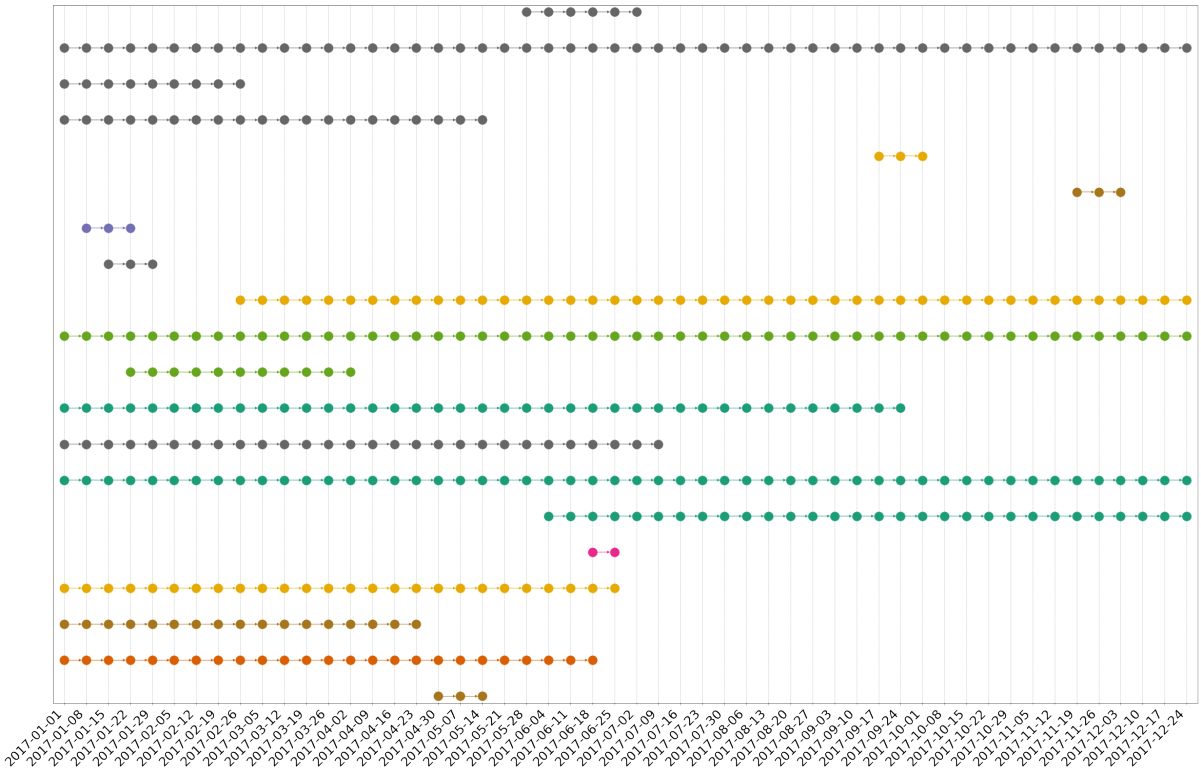


Figure 4.10: A visualization of the lifetime of a sample of 20 identified networks

appropriate behavior. We focus in this section on identifying clusters that show IP-flux type behavior.

One of the most notable results so far is that the general characteristics indicate that only a small fraction of the clusters have more than 1 IP-address associated with them. The data set only contains 858 clusters that have more than 1 IP-address associated, which is only 0.88% of the total detected clusters. Furthermore, the majority of this set of clusters only have a small number of IP-addresses related to the cluster, being either 2 or 3 IP-addresses. Given the weekly periods in which we generated the clusters, we expect that an IP-flux network shows up in the results as a cluster with numerous IP-addresses associated it. However, in the data set of the current case study, the maximum number of IP-addresses associated to a single cluster in one period is 24; which is not that excessive for an actual IP-flux network. A study by Nazario and Holz [19] further emphasized this fact that 24 IP-addresses is rather small for an IP-flux network. They state that based on a cumulative data set of 4 months, the average flux-network has $2,683$ distinct IP-addresses associated to the network. They base these findings on a data set containing records from the ATLAS system which is a data repository and globally deployed network of honeypots. So even dividing the number of distinct IP-addresses into weekly segments, we determine that on average the flux-networks used in their studies have an approximate of 167 distinct IP-address in a week. This result might indicate that the current data set does not contain any IP-flux network simply because we have not identified a single cluster with more than 24 associated with it. The overview of the number of clusters with more than 1 IP-address associated to the cluster is shown in Table 4.8, clearly indicates that the overall majority of clusters only contain a small number of IP-addresses.

We verify the cluster to determine further if they show any IP-flux behavior according to the methodology specified in Section 3.5.2. Although in general, it is improbable that the majority of detected clusters show IP-flux behavior, we still thoroughly analyze the clusters to determine whether any of the detected clusters show IP-flux behavior.

4.4.1 IP-flux results

Based on the previously described requirements, we analyze the clusters for the various categories to determine which cluster shows specific IP-flux behavior. We describe the

ip_size category	2	3	4	5	6	8	9	12	14	15	16	24
1 *small sized clusters*	660	42	1	0	0	2	1	0	0	0	0	0
2 *normal sized clusters*	53	27	4	0	0	0	0	0	0	0	0	0
3 *normal sized clusters / low hit-coverage*	42	11	1	1	1	0	0	1	0	1	0	1
4 *large sized clusters*	3	1	0	0	1	0	0	1	1	0	2	0

Table 4.8: Overview of number of clusters with more than 1 IP-address

analysis of identifying clusters with IP-flux behavior in the following sections apportioned by the cluster categories.

Cluster category 1 Starting with category 1, which incorporates the largest number of clusters that contain more than 1 IP-address. Initially, category 1 consists of 702 clusters with more than 1 associated IP-address; however, the greatest number of unique IP-addresses for any clusters identified as category 1, is a total of just 3. Given the characteristics, it is therefore unlikely that any of these clusters use some form of IP-flux functionality. Using the requirement that an IP-flux should contain IP-addresses that are not solely related to the same network reduced the number of viable clusters to 183. The verification that the clusters should also contain IP-addresses that are not all related to the same country further reduces the number of clusters to 34. Although there is a steep decline of viable IP-flux clusters, there is still a significant number of clusters that fulfill the mentioned requirements for an IP-flux network. The final verification is whether the viable clusters are part of a network that uses the IP-addresses for an extended period. Validating this requirement results in 0 clusters meeting all requirements and therefore none of the category 1 clusters have the potential of being an IP-flux network.

Cluster category 2 We identify a total of 84 clusters as both category 2 and has more than 1 associated IP-address, which is less than the number of clusters with more than 1 IP in category 1. Initial analysis of the /16 prefixes of those IP-addresses of the clusters shows that a subset of 58 clusters had more than 1 /16 prefix associated with it. Again using the *Maxmind* database, we determine the countries for each specific cluster in the data set. Given the previous explanation that a specific IP-flux network would not limit itself to a single country, we perform additional filtering of the potential IP-flux clusters by removing all clusters with only one country associated with it. This limitation resulted in only 31 clusters that could still possibly be an IP-flux cluster. Unfortunately, the final verification of the requirement that the cluster does not use the associated IP-addresses for an extended period in the same network results in 0 clusters being a viable IP-flux network.

Cluster category 3 The cluster category contains 59 clusters with more than 1 IP-address. Noticeable is that category 3 contains clusters with some of the highest number of associated IP-addresses. Even the cluster with 24 IP-addresses, which is the largest number in the entire data set, is classified as a category 3 cluster. Due to the IP-flux behavior, the more IP-addresses a cluster has associated with it, the greater the chance that it is an IP-flux network. Interesting enough, the first requirement for which an IP-flux should have IP-addresses associated to it from multiple hosting providers, only resulted in a single cluster not complying to this requirement. So a total of 58 clusters we still validate, in other categories this requirement results in a far greater number of clusters that we removed as potential IP-flux networks. The requirement that an IP-flux should not be hosted only in the Netherlands had a far bigger impact, lowering the number of potential IP-flux clusters to 9.

Based on the final requirement, that an IP-flux should not use similar IP-addresses for an extended period in the same network, further diminished the number of possible IP-flux networks to only 2 for category 3. The characteristics of these networks are shown in Appendix A for clusters with id 3104 and 59221. Interesting enough, there are some similarities between

these two clusters. First of all, although the IP-addresses are unique between the clusters, they do share some commonalities, for example when comparing the IP-addresses based on the /16 prefix they do have prefixes in common. Also, a subset of the hit-list of the cluster 3104 is also present in the hit-list of the cluster 59221. The fact that both clusters have various associated IP-addresses and the fact that both sets of IP-addresses completely changed in a matter of a couple of months is behavior which we could potentially attribute to an IP-flux network.

The only attribute that could be an indication that these clusters are not IP-flux clusters is the fact that the number of malicious domains attributed to the clusters compared to the benign domains is rather small. Both clusters have a hit-coverage of only 0.27% or lower. Delving into the hits of the ground-truth and verifying them using VirusTotal, indicates that only a single domain used by both clusters shows any sign of malicious behavior. VirusTotal is a useful tool to verify the domains because every input is verified VirusTotal by validating against 68 different Anti-Virus solutions of numerous companies. Since each Anti-Virus solution has its unique threat intel programs, it is possible to validate a single domain name against numerous intel program quickly. Previous researches[20] have already used VirusTotal to verify the maliciousness of the domain names.

Validating the other domains in the ground-truth of both clusters shows only benign behavior according to VirusTotal. Since only one domain in both clusters shows any malicious behavior, it is difficult to categorize the entire clusters as malicious. Furthermore, the malicious domain has only been positively identified as malicious by only five out the 68 Anti-Virus solutions; the other applications identified this domain as benign. The 5 Anti-Virus has classified this domain as a domain showing phishing behavior. Implementing a proper IP-flux mechanism takes substantial effort, and therefore it is highly unlikely that the entire defensive mechanism is only implemented to host a single phishing website. The *NetCraft* ground-truth source identified all the domains in the hit-list of both clusters; unfortunately, it seems impossible, by open-source intelligence, to determine the cause of the blacklisting in *NetCraft* blacklist. Since only one domain shows any malicious behavior was verified using open-source intelligence, and that for the other FQDNs entries in both clusters no malicious behavior was detected, both clusters are not labeled as malicious and are therefore not identified as IP-flux networks. Also, investigating into the prefix indicated that *CloudFlare* currently owns the prefix. This ownership of *CloudFlare* might be the reason why there is such a high change-over of used IP-addresses used by the domain names and is therefore not linked to IP-flux behavior.

Cluster category 4 Category 4 contains the least number of clusters with more than 1 associated IP-address, only a total of 9 clusters fulfill this requirement. Although we should mention, that even though category 4 has a small number of clusters with more than 1 IP-address, the majority of those clusters have far more than 2 IP-addresses related to it, making it more likely that they are IP-flux networks. The initial validation of whether the cluster used more than one network reduced the number of viable clusters to 6. We further validate the requirement that the clusters should be located in more than one country; this limits the number of viable clusters to 3. The final validation that the cluster should not use the same set of IP-addresses for an extended period in its network reduces the number of viable IP-flux networks to only 1.

The final viable cluster is again shown in Appendix A with id 14289. Interesting enough the last viable IP-flux cluster in category 4 show similarities with the previously described clusters in category 3 with id 3104 and 59221. Although this cluster has more domain names associated with it, a total of 6125 domains, compared to the other clusters, the clusters do have commonalities in the hit-list. Investigating the prefix indicates that *CloudFlare* hosts the networks, similar to the other networks. Using the same validation technique, we verified the ground-truth hit list of this cluster, and again only a single domain shows malicious behavior; the same domain name we encountered in for the category 3 clusters. Using the same argumentation, a single malicious domain name for a cluster containing 6125 domains is not reason enough to classify the entire cluster as malicious. So, therefore, this cluster is not categorized as an IP-flux network.

4.4.2 IP-flux detection conclusion

Using the previously described detection method, it was not possible to classify any cluster as an IP-flux network. We already predicted this outcome due to the lack of clusters with numerous associated IP-addresses, even the cluster with 24 IP-addresses is not that many when comparing the cluster actual know IP-flux networks. Although we encountered some clusters that initially fulfilled every requirement and which we could potentially label as an IP-flux network, the actual malicious indicators of those clusters were just too minuscule to label the entire cluster as malicious.

4.5 Detection of domain-flux

As stated in the description of the methodology for identifying domain-flux in Section 3.5.3, we are identifying domain-flux networks based on the characteristics of the domain names associated to the cluster by comparing them to benign domain names. An example of domain names generated by the DGA function shown in Listing 1.1, which was described by Chiu and Villegas [13], is shown in Listing 4.1. It is visible that these domain names are distinctly different from benign domain names such as `google.com` or `wikipedia.org`. By analyzing the characteristics of the domain names, we aim to identify these generated types of domain names used by domain-flux networks.

```
a542b857df2b9ad746ea85d9792e8f4c88.cc
b462c5daae400c715b12be13593ce7f9bf.ws
c35f84584f97cc2afd2d2c3d26a97e0b9e.to
d8455237a828234a2ea7ad175aa2db64a9.in
e199b1ab95141e9d953dd9f84a069dd9da.hk
```

Listing 4.1: Domain names generated by a DGA as described by Chiu and Villegas [13]

4.5.1 Domain-flux statistical results

Once we gather both the statistics of the detected clusters and the benign domain names we can make an effort to determine whether there exist any domain-flux clusters within the data set. As previously indicated, we use a similar approach as described by Passerini et al. [7] which uses statistics to determine any cluster which shows abnormal data when compared to benign domains. We show the statistics gathered for the 10.000 domain names in the benign data set in Table 4.9. In general, these results indicate that there are very few digits and few dictionary words used in the benign domain names. Also, the overall length of the domains names in the benign data set is somewhat limited. We attribute this behavior to the manner that popular websites choose to use relatively small domain names that consist of unique names because this makes it easier to remember. We use the statistics shown in Table 4.9 to verify the results gathered from the domain names of the malicious clusters analyzed in this case study.

To better understand the underlying relations between the various properties of the domain names and the possible relation with malicious clusters, we generate a multitude of graphs indicating the number of occurrences for each category of clusters for the various statistic property; the graphs are visible in Figure 4.11. The figure indicates on both the x and y-axis the averages of the attributes which we calculate, thus the number of words in a domain name, the number of digits, consonants and vowels and the overall length of the domain name. These scatter plots show the density of the clusters for specific statistics on the x-axis and y-axis, and so we use this to identify a possible relation between the various statistics. As such, we use these scatter plots to identify any partial relation between the properties gathered from the domain names in the clusters. Furthermore, it is possible to get an indication of which category contains the most outlier cluster because the various categories are independently colored. The diagonal graphs in the figure show a histogram for that specific property that we then use to determine the overall distribution of the categories for the specific statistic.

A practical example on how to read these plots, given as an example the scatter plot on the top row second from the left, is that the majority of clusters have an average of zero numerical characters in the domain names of the clusters. We make this conclusion by the fact that the majority of dots are located at the leftmost axis of the plot. Visible, however, are some outlier clusters for cluster category 1 and 2, that have an average of 2 or more numerical characters in the domain names, visible in the right-hand side of the plot. There is also a slight increase in the number of numerical characters in the domain names for the word-average of around the value 1 and 3. We use the same cluster categories in this description as we specified in Section 4.3.3. We use these findings to determine if there are any relations between the various properties of the domain names. Please note that the results in the scatter plots of the lower left corner are identical to the top right corner but that the graphs are inverted. In the following section, we reference the graphs in the figure by numbers ranging from 1 to 25; the graphs are indexed from left to right and from top to bottom.

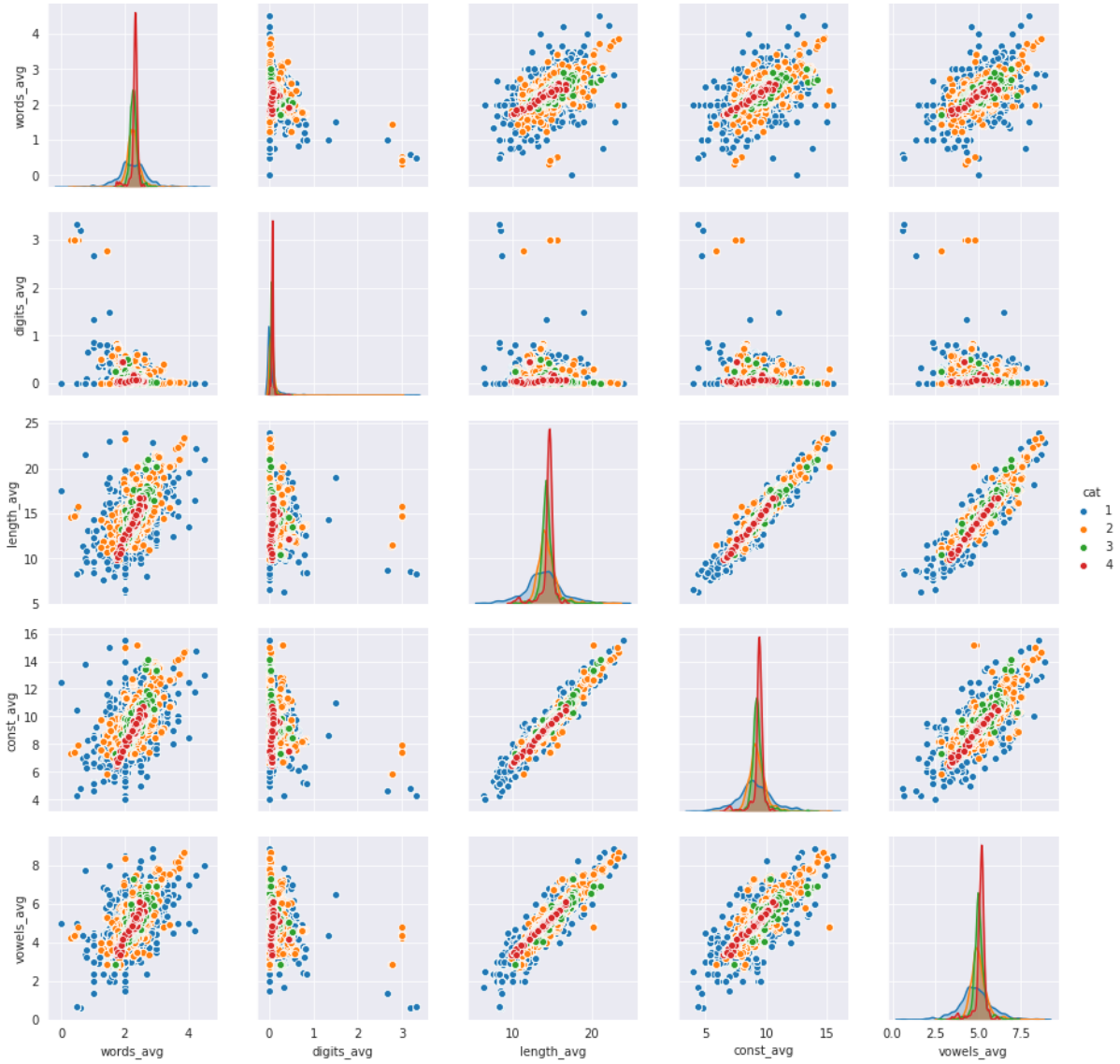


Figure 4.11: Overview of various statistics used in the detection of domain-flux and their underlying relations

Given the results we derive some interesting conclusions from the graphs of Figure 4.11. Namely that it is possible to conclude that there exists some form of relation between the number of vowels and consonants when compared to the overall length of the domain name. This relation can be seen in graph 18 and 23 by the very diagonal orientation of the scatter plot indicating a similar increase in vowels or constantans with an increase in domain name length. Although we expect this type of relation, it is interesting to verify this relation so clearly within the graph.

Another aspect that is visible within the figure is that the overall number of digits is quite low in the various categories of the clusters. The main bulk of the clusters have an average roughly higher than 0 as is shown in the histogram in the digits_avg column or row located in graph 6. Although the majority of the clusters have a small average of the number of digits in the domain names, there are outlier clusters that have significantly more digits, in some cases an overall average that exceeds 3 digits per domain.

There is no clear relation when comparing the number of words to the overall length of the domain name, especially when comparing to constantans and vowels with the overall length. As shown in graph 3 and 11, the scatter plots are still in general diagonal orientation, meaning some form of a simultaneous increase in the number of words while compared to the overall length. However, the diagonal orientation is not as clear when compared to the vowels and constantans concerning the overall length plus it has a far more circular layout. We conclude from this graph that the relation between the number of dictionary words and the domain name length is not that distinct. We make this same conclusion when comparing the number of dictionary words in the domain name against the number of vowels and consonants in the domain name. So the scatter plots of these comparisons all have a general diagonal orientation but that the overall graph is too circular to make any clear distinctions of the underlying relations.

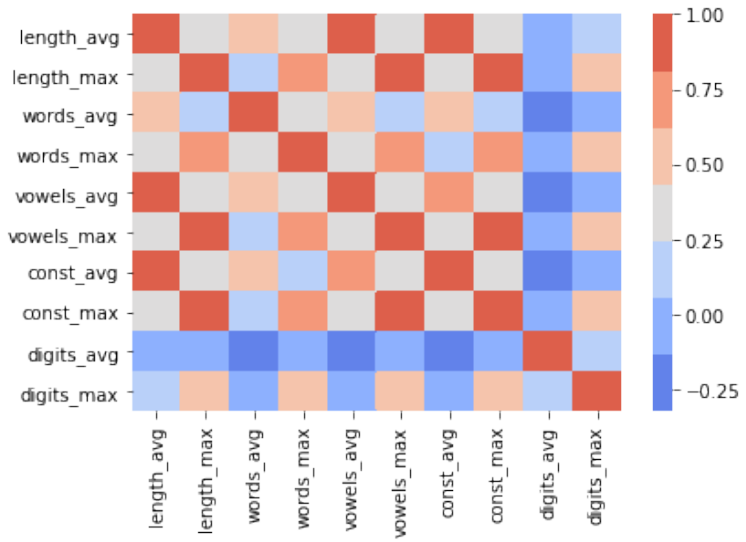


Figure 4.12: Heat map of the correlations of the properties calculated of the domain names in the malicious clusters

The graphs in Figure 4.11 show some interesting results with regards to the possible underlying relations between the overall statistics gathered from the clusters. When looking at the general figure, there is a clear distinction that we make on which category of clusters have more outliers. When looking into the graphs, it is clear that the majority of the categories that have outlying clusters compared to the overall results for the graph are either category 1 or category 2 clusters. It is even possible to roughly identify the categories of clusters within each graph as layers on top of each other. All the graphs show a center of category 4 clusters (red), then a small layer of category 3 clusters (green), then a layer of category 2 clusters (yellow) and finally a layer of category 1 clusters (blue). The histogram graphs, in the diagonal of the figure, further enforces this idea. These graphs, show that the larger sized cluster, either category 3 or 4, have a more clear spike in the graph meaning that the clusters centers around a specific average. While comparing the smaller or standard sized clusters of category 1 or 2, these categories have a far broader range of average values that are distributed more around the x-axis of the graphs. Due to this type of behavior, we expect that if domain-flux clusters are detected, it is more likely that they are either category 1 or 2 instead of category 3 or 4.

We further substantiate the underlying relations between the various properties in Figure 4.12. This figure depicts the correlation between the properties by using a heat map, the warmer the color, the higher correlation is between the properties and the cooler the color, the less

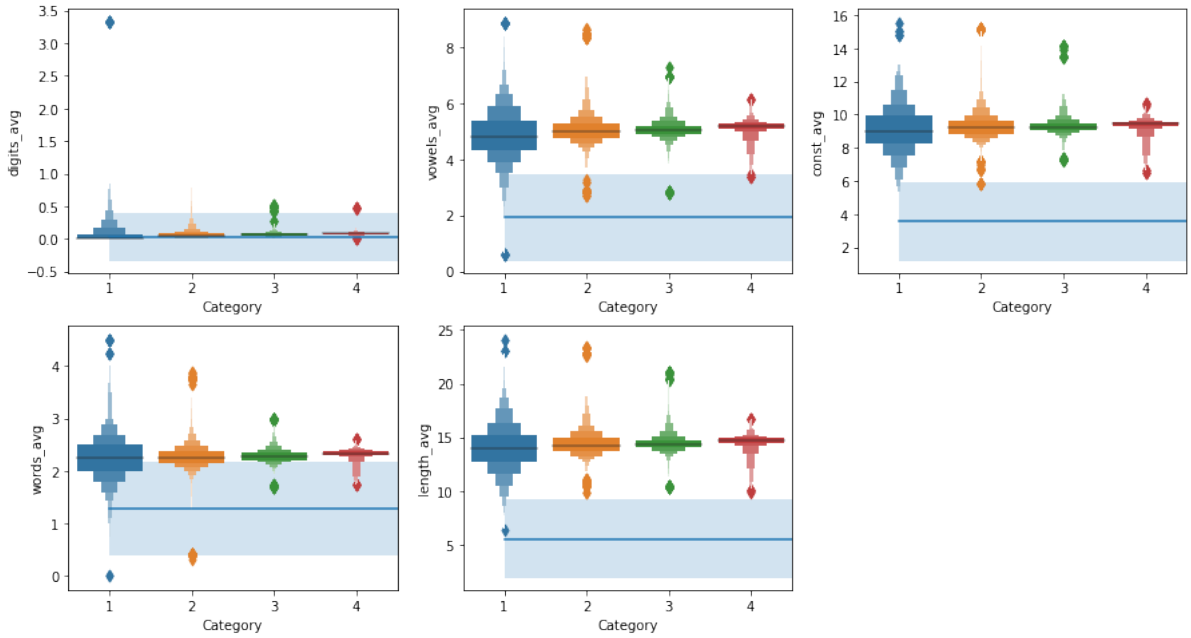


Figure 4.13: Graph displaying the distribution of values for the statistics compared to the benign averages

correlation there is. This figure shows both the averages and the max values that we gather from each cluster. In most cases, there is a correlation between the max value and the general average of each property as could be expected. Furthermore, the figure also shows a general high correlation between the average length of the domain names and the number of constantans and the vowels in the domain names as was previously described. The exciting aspect of this figure is the fact that the entire correlation of the average number of digits in the domain names is very low, this means that the average number of digits in the domain names is not related to another property, not even to the average length of the domain names.

The underlying relations of the different properties are made apparent with Figure 4.11 and 4.12. However, these figures do not show any relations between the statistics gathered from the malicious clusters and the statistics gathered from the benign domain names. This difference is shown in Figure 4.13, which shows multiple graphs for the various properties that we gathered from the malicious clusters compared to the average and the deviation of the benign domain names. These five graphs indicate the properties that we gather, and within each graph, the four box plots show the distribution of the averages of the specific property for the various cluster categories. The horizontal blue line is the average of the benign domain names with the standard deviation indicated by the blue area. The statistical results for the various properties are also shown in Table 4.9.

There are some interesting conclusions that we make based on Figure 4.13 and the statistical results in Table 4.9. First of all that most statistics gathered from the malicious clusters are far more significant when compared to the results of the benign domain names. Starting with the average length of the domain names, it is clear that the average length of domain names in the malicious clusters is far greater than the average length of the benign domain names. The main bulk of the distribution of the categories centers around an average length of 14 characters while the average length of the domain names centers around 6. Even when taking the standard deviation of the benign domain into account, the average length of the domain names in the clusters of the various categories are still far more substantial. Only some clusters in category 1 have an average length of domain names that are comparable with the lengths of the benign domain names; however, this is only a small subset of the category 1 clusters. Due to a higher average size of the domain names, it is not strange behavior that the average number of vowels and consonants is also higher. So simple outlier behavior for a larger than an average number of vowels and constantans is not sufficient to classify a cluster as a domain-flux network due to the larger average size of the domain names.

	Type	D_1	D_2	D_3	D_4	D_5
Benign	*Avg.*	1.92	3.57	0.03	5.53	1.28
	Dev.	1.54	2.38	0.36	3.64	0.89
Category 1	*Min.*	0.60	4.00	0.00	6.33	0.45
	Avg.	4.86	9.08	0.06	14.01	2.25
	Max.	8.85	15.50	3.33	24.00	4.50
Category 2	*Min.*	2.68	5.84	0.00	9.84	0.22
	Avg.	5.04	9.28	0.07	14.40	2.27
	Max.	8.68	15.15	3.00	23.37	3.88
Category 3	*Min.*	2.77	7.24	0.00	10.28	1.67
	Avg.	5.06	9.31	0.06	14.44	2.28
	Max.	7.30	14.13	0.51	21.07	3.00
Category 4	*Min.*	3.34	6.49	0.00	9.90	0.11
	Avg.	5.13	9.34	0.07	14.56	2.31
	Max.	6.14	10.70	0.48	16.74	2.60

Table 4.9: Statistics of the benign and malicious domains gathered for domain-flux detection

In the effort of trying to detect domain flux-networks, the focus is on clusters that show outlying behavior when compared to the benign domain statistics. Both the digits and words statistics are the focus of this analysis to identify domain flux-networks because as was previously stated by Stevanovic et al. [16], domain flux-networks are categorized by fewer words within the domain names and higher use of digits with a larger than average domain length. The focus is not on the vowel and constants statistics because they are related to the overall higher number of words within the domain names and the higher average length when compared to benign domains.

4.5.2 Domain-flux word statistics outliers

We characterize domain flux-networks by their use of randomly generated domain names containing a random sequence of alpha-numerical characters. Therefore these domain names usually contain fewer dictionary words than a benign domain name. As we show in Figure 4.13, there do exist some clusters that we identify as outliers because they contain properties that are lesser than the overall standard deviation of the benign domain names. Due to the characteristics of domain flux-networks, we analyze only clusters with an average of fewer words than the benign domains. This limitation results in only three clusters from which two clusters have an average of zero words and the third cluster have an average of 0.30719 words per domain name.

The two clusters with an average of zero domain names are related to the same network, which we easily asses by analyzing the statistics. These clusters are shown in Appendix A with id 17198 and 42105. Both of these clusters are very small in size with only four domain names associated with them. Because the two clusters are identical given the gathered statistics, except for in which period they were detected, we only analyze one cluster to determine whether it shows domain flux behavior, especially because both clusters contain the same hits in the ground-truth. The hit list of the analyzed cluster only contains two entries both of which are related to the business of a handyman. Both these domain names are associated with malware distribution by three anti-virus solutions according to VirusTotal. Given the manual inspection, it is clear that some Dutch words are contained within the domain names however there were not detected because we did not train the Dutch corpus, used to identify the words in the domain names, for this exact combination. So the fact that this outlying property of these domain names contains 0 words is a false positive. Therefore both these clusters cannot be categorized as a domain flux cluster.

The final cluster that contains a word-average of 0.30719 is analyzed to determine whether

it does show domain flux behavior; again the specific statistics of this cluster is shown in Appendix A with id 22931. When compared to the previous cluster, this cluster does contain more entries in both the hit-list as well as the overall domain-list making it more likely that it could show domain flux behavior. A subset of the hit-list of this cluster contains domain names of a specific format. Namely, the emergency number of the Netherlands *112* with appended the name of a Dutch town or city. All the domain names in both the hit-list as well as the domain-list follow this format. The use of names of Dutch towns and city is the reason why the average number of words in the domain name is so low because these names are not listed in a Dutch dictionary. Only a small fraction 2.3% of the total number of domains is associated by a small number of anti-virus solutions as malicious, being a PayPal phishing site. Because of this reasoning, we expect that some of these domains have been compromised and were reconfigured to host a phishing site but that in general the entire clusters cannot be categorized as a domain flux network.

4.5.3 Domain-flux digits statistics outliers

In the same manner, as described for identifying outlier clusters for the word statistics, outlier clusters are analyzed for the digit statistic. However, in contrast with the word statistics, the outlier cluster for the digit statistic is performed based on clusters that have a higher average of the number of digits within the domain names compared to the benign cluster. We previously elaborated on this decision in Section 3.5.3, it revolves around the fact that we characterize domain-flux by containing more numerical characters when compared to benign clusters. Thus clusters that have an average number of digits higher than the average with a standard deviation of 0.404127 compared to the benign domain names are further analyzed. In contrast to the word statistics analysis, this requirement resulted in 1.276 clusters being a potential domain-flux network based on the fact that having, on average, more digits within the domain names of the cluster when compared to the benign domain names. The number of clusters makes it impossible to manually verify every cluster to determine whether they are domain-flux networks. Using the networks that we previously determined in Section 4.3.3, it is possible to relate the clusters to a total of 61 networks. These networks sizes range from the size of only 2 clusters to the size of containing a total of 52 clusters and thus spanning the entire year.

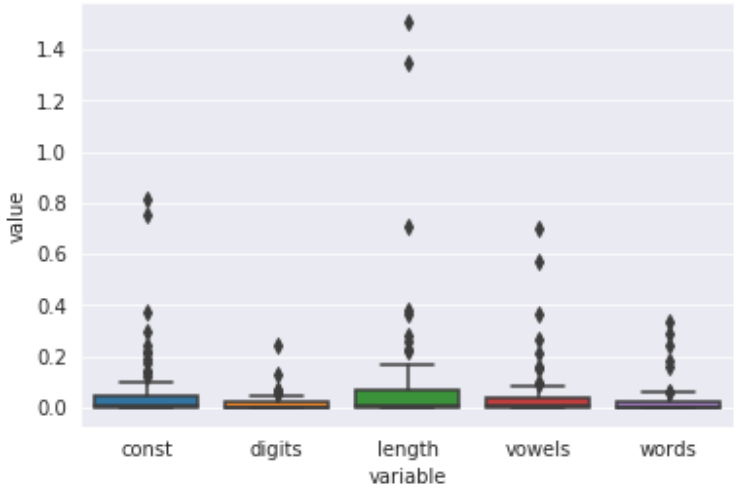


Figure 4.14: Graph displaying the deviation of the various statistics of the clusters within the networks

Delving into these networks, we show that all 61 networks have a similarity of 1.0, indicating that every cluster has an exact match with all other clusters within the same network. Although this distinction is sufficient to analyze the networks of the clusters further, we should note that although the similarity of the comparing properties complete, there is some small deviation in the properties of the domain names of the clusters in the networks. The graph in Figure 4.14 shows the deviations of each of the collected statistics for each network. Although for

the majority of networks the deviation for each statistics is rather small, there are again some outlier networks that show a higher than the average deviation of the property for the clusters in the network. Besides these deviations, the fact that every network has a similarity of 1.0 for each cluster within the networks makes it sufficient to analyze a single cluster of each network to determine whether it is an actual domain-flux network instead of analyzing all clusters within every network. This procedure reduces the number of clusters that we have to analyze to only 61 clusters.

Results cluster category 1 and 2 Analyzing the hit-list of the category 1 and 2 clusters of this data set, we indicate that although the overall clusters show a higher than the average number of digits in the domain names, this is not immediately related to the hit-list of those clusters. It is even more interesting that the overall majority of the hit-list of category 1 and 2 clusters do not contain numerical characters at all. Verifying the maliciousness of some of the domain names in the ground-truth does indicate that some clusters contain domain names that are related to malicious activities. Such an example is the domain name belonging to a marathon event website which domain name revolves around the Dutch dumplings that are served commonly on new year's eve, that we associate with the distribution of Trojan Horse malware. However, it is again interesting to note that this particular domain does not contain any digits whatsoever.

The previously mentioned cluster with id 22931 in Section 4.5.2, is a category 2 cluster. However, due to the formatting of the domain names associated with the Dutch emergency number it is clear that this cluster is not a domain-flux network, although it has a high average of numerical characters in the domain names. Furthermore, the most interesting cluster of the category 1 clusters is a cluster with the highest average recorded for the number of numerical characters in the domain name. We show this cluster in Appendix A with id 23330 and which is a relatively small cluster with only 2 known hits in the ground-truth. However, verifying the maliciousness of the domain names related to this cluster reveals that VirusTotal has no records of any potential malicious behavior on both domains from the known malicious ground-truth. The strange aspect remains that this particular cluster has almost an average of 3 numerical characters for each domain in the cluster. Analyzing the domains associated with the cluster indicates that the domain names all follow a specific pattern of containing the word *crm* appended with a year ranging from 2014 towards 2018. Due to the small size of this cluster, the overall average of the number of digits in each domain name is higher than in larger clusters. So since no external source could verify the maliciousness of the domains and because the domains follow a specific pattern in defining the domain names, this cluster is again not categorized as a domain-flux network.

Results cluster category 3 and 4 The clusters that we categorize as either category 3 or 4 contain domain names found in the ground-truth that do not contain any numerical characters. This result is interesting because the domain-lists of these clusters do contain a slightly higher average of digits in the domain names than the benign domain names. The category 3 cluster with the highest average of digits in the domain names associated to the cluster, is shown in Appendix A with id 134839, only contains two ground-truth hits that both do not contain any numerical characters. Using VirusTotal both these domain names are associated with phishing attacks by only a small number of anti-virus solutions. The rest of the domain names of the cluster, that are not listed in the ground-truth, we mainly associate with pornographic content. The higher than the average number of numerical characters in the domain names we associate to entries in the domain-list of this cluster which contain three or four digits and which do not contain a clear indication of the purpose of these domain names. However, these entries are rather short when compared to actual domain names used by domain-flux networks. Furthermore, there is no sign of actual malicious behavior by these domain names. Therefore this cluster is not regarded as a domain-flux network.

The category 4 cluster with the highest average of digits in the domain names shows similar behavior to the previously discussed category 3 cluster. Mainly, it only contains a very small hit-list of only two entries, and it contains in the domain-list entries that consist solely of numerical

characters. Some of these domain names with many numerical characters have been *parked* by *Sedo Domain parking*[1] an organization which buys, sells and resells domain names. None of these domain names show sign of malicious behavior; we show the specifications of this cluster in Appendix A with cluster id 25219. The overall coverage of this cluster is so low that it is difficult to mark the entire cluster as malicious due to only two hits in the ground-truth. The hits associated to this cluster are related to only two domain names both of which do not contain any numerical characters and based on visual inspection of the domain names are unlikely to be domain names used by a domain-flux network. Only one anti-virus solution associated one of the domains with phishing attacks; however, we associate the second domain name with increased malicious behavior. We conclude that besides hosting phishing sites it also briefly distributed a trojan virus that could impact a system. So although this is malicious behavior, the current properties of the cluster and the properties of the domain names in the hit-list show that we cannot categorize this cluster as a domain-flux network.

4.5.4 Domain-flux conclusion

We aimed at identifying domain-flux networks by analyzing clusters that show outlier behavior for both the average number of words in the domain names associated with the clusters and the number of digits in the domain names. We find it interesting to note that although we identified numerous clusters that show outlier behavior when comparing the statistics, in most cases this was not directly relatable to the domain names in the hit list of the cluster. So we identified numerous clusters that contain a higher than the average number of digits in the domain names of the cluster but did not find any domain names with digits in the hit list of the same cluster. Thus analyzing these clusters always results in the categorization that it is not a domain-flux network. Furthermore, as with the detection of IP-flux networks, there have been numerous cases for which the malicious of the domain names could not be verified using open-source intelligence. Because in these cases the maliciousness cannot be verified we cannot label the entire cluster as a malicious network. So overall, we did not find any domain-flux networks in the *OpenINTEL* data set.

[1] https://sedo.com/us/

Chapter 5

Discussion

The study documented in this report revolves around the question of whether it is possible to implement a known flux-network detection mechanism that uses the *OpenINTEL* data set. The initial expectation is that this is possible because most detection mechanisms use DNS records to determine their results, so there should not be any significant obstacles into implementing a similar detection mechanism using the *OpenINTEL* DNS resource records. The only significant difference in using the *OpenINTEL* records is the fact that that *OpenINTEL* centers around an active DNS data set instead of a passive data set.

At the start of this case study, we already know that there exists a significant difference in the use of either a *aDNS* or *pDNS* data set. Namely, that the *OpenINTEL* data set contains far fewer data points for each specific domain for each day when compared to other *pDNS* data sets. *OpenINTEL* gathers its statistics for each domain only once a day while a *pDNS* can retrieve many records throughout a single day for famous domain names. However, this granulation in *pDNS* data set depends on the popularity of the domain names while *OpenINTEL* always gathers every type of DNS record for each available 2-level FQDN within the TLD in each day. So although we expect that this reduction in available data points for each specific domain can affect the detection capabilities in some way, we also expect that the property of *OpenINTEL*, in which it ensures that it collects 100% of the available 2-level FQDNs, would greatly benefit the effectiveness of any detection implementation that would use the *OpenINTEL* data set.

5.1 Implementing a known detection algorithm

The initial proposal for this case study was to focus on an existing detection algorithm and to try to mimic the functionality of the algorithm using the *OpenINTEL* records as its initial data source. There has been a wide range of studies that have researched the possibility of detecting flux-networks using DNS data sets. Because of the high dimensionality of the data set of *OpenINTEL*, the initial focus of the detection mechanism revolves around detection mechanisms that use some form of clustering before analyzing the results. Furthermore, due to the time constraints of this case study, the focus for a detection mechanism to mimic lay on studies that require access to DNS properties that are available in *OpenINTEL*.

We decided to mimic the *FluxBuster* detection methodology as much as possible for detection application for *OpenINTEL*. We choose *FluxBuster* because of the majority of the properties required by this detection mechanism are readily available in *OpenINTEL* and because the TPR and FPR of the detection methodology are sufficient to expect the proper result. We decide, however, to not implement the same classification methodology that is used by *FluxBuster* because of the extended effort it would require to implement correctly. This implementation would result in insufficient time to properly analyze and verify the actual results of the detection methodology for *OpenINTEL*.

The majority of the properties required by the *FluxBuster* detection mechanism are readily available in *OpenINTEL*; except for the TTL DNS property which is not recorded by *OpenIN-*

TEL. The fact that *OpenINTEL* does not record this DNS property is an important indication of what the focus is of *OpenINTEL* compared to other *pDNS* data sets. The focus of *OpenINTEL* centers around getting an accurate impression of an FQDN for an extended period instead of getting a historical record for each domain name. We can further substantiate this by the fact that *OpenINTEL* was developed for the initial research that centers around the research of DDoS [2] mitigation and DNSSEC [29] implementations; both types of research which require access to DNS records over an extended period but does not require access to accurate records on a minute or hourly basis. This granulation of the data records by *OpenINTEL* is not an issue for flux-network detection because for most detection mechanisms mentioned in this case-study a daily granularity is sufficient.

5.1.1 Defining the similarity threshold

Given the exception of the lack of TTL, the remaining recorded DNS features in the *OpenINTEL* data set were sufficient to mimic the detection method of *FluxBuster* further. The first major part of mimicking the *FluxBuster* detection algorithm, requires the implementation of a clustering algorithm. This algorithm centers around the Jaccard similarity of the IP-addresses to cluster relevant records. Following the implementation of *FluxBuster*, the initial clustering algorithm that we used was HCA that was implemented in the same manner as documented by Perdisci et al. [3]. The clustering algorithm does require a similarity index to determine which records are still relevant or irrelevant to other records. Basically, what is the minimal requirement for the similarity for two records to be linked to each other in a cluster? Perdisci et al. have used *natural clustering* to determine this specific threshold value; this method revolves around the idea that by mapping the number of clusters for multiple thresholds on a plot it is possible to see certain plateaus of *stable* clustering. The authors of *FluxBuster* documented this specific implementation as:

> **Perdisci et al. [3]**, . . . we look for plateau (i.e., flat) regions in the graph that are an indication of *"stability"* or *natural clustering*. Plateau regions correspond to those steps of the algorithm where the two nearest clusters that have to be merged exhibit a quite low measure of similarity.

Using the same approach as Perdisci et al., we gather a subset of 10.000 records from *OpenINTEL* to determine this specific threshold. Using the HCA clustering algorithm, we gather the number of clusters from the HCA to determine for various thresholds; basically, the threshold value van 0.0 to 1.0 in incremental steps of 0.1. Similar to the results of Perdisci et al., there formed plateaus of the number of clusters detected around certain thresholds, as is shown in Figure 3.1. As mentioned by Perdisci et al. [3], these are the plateaus of clusters that formed due to *natural clustering* and because the implementation has been equal to an already published paper, these results were used to determine the threshold of 0.58 used in the remaining of the case study.

Although we implement the same algorithm, we should note that the results between the current case study and the study performed by Perdisci et al. show significant differences with the resulting threshold. The implementation in this case study results in a Jaccard similarity threshold of 0.58 while the study by Perdisci et al. [3] resulted in a similarity threshold of 0.75. The difference of 0.12 of additional similarity that was required by Perdisci et al. [3] is notable and if the same similarity were used in this case study some of the clusters that were detected would not have been initially recorded due to a lack of similarity. A potential explanation of this difference might be related to the variation in the source of the data set. Perdisci et al. used, as many Flux-network kinds of research, a *pDNS* data set as the data source for their research. It could be possible that this variation in the characteristics of the initial data source might be the reason for this notable difference in the determined threshold.

Another potential theory for the notable difference in the resulting threshold might be due to an error in usage or implementation in the initial process of determining the threshold value by Perdisci et al.. Given the other works in flux-network detection, there has not been a similar method for determining the threshold value documented by other researches, that we analyze in this case study. This fact makes it difficult to cross-examine the validity of this particular

method because there are just no other documented cases of this particular method; as far is known in this case study.

5.1.2 Difficulty in using high dimensional data sets

After defining the similarity threshold, an initial attempt was made to implement the HCA algorithm. After initial tests, we quickly determine that this particular algorithm is not suited for high dimensional data sets, as is elaborated in Section 3.1.3. The issue revolves around the similarity matrix that is required by the HCA to perform clustering. This matrix has to contain the similarity index of each record compared to each other record. Even when using a condensed matrix, the minimum computational requirements for processing and storing this matrix increases due to the exponential growth in the data size for each record that we add to the matrix.

Also, due to the limitation that the clustering algorithm should have access to the entire similarity matrix, this algorithm cannot be distributed among multiple processing nodes to process large data sets. Due to these requirements, there exists a rough maximum of the number of records that can be clustered by HCA before the computational requirements are becoming too extensive. This exact configuration, of course, depends on the actual system that the researches used for the clustering of the data set; however, an approximate estimation can undoubtedly be made based on the typical hardware specifications used at the time.

Given this limitation of the HCA algorithm, we make two conclusions. First that this particular clustering algorithm is not usable for high dimensional data sets, making it unable to use it for *OpenINTEL* given the dimensionality of that data set. This property results in that it is not possible to implement an exact similar method for the detection of flux-networks as described by Perdisci et al. in *FluxBuster*. Secondly, we conclude that the specific implementation of *FluxBuster* have had a maximum number of records that could be processed at a single time by their detection method based on the processing requirements by the similarity matrix of the HCA algorithm. We, therefore, expect that the number of records parsed by *FluxBuster* and the results of this detection are far lower than the number of records analyzed in this current case study. Given the results of *FluxBuster* as described by Perdisci et al. [3], this seems indeed to be the case.

> **Perdisci et al. [3]**, Overall, in a period of about five months of operational deployment, FluxBuster classified $4,084$ domain clusters as flux and $3,633$ domain clusters as nonflux, which included a total of $1,743$ 2LDs ($63,442$ FQDs) and $227,667$ 2LDs ($264,550$ FQDs), respectively.

The results described by the *FluxBuster* detection method are far less than the overall results gathered from this case study using the *OpenINTEL* data set. When looking into the sum of clusters detected by *FluxBuster* 7.717 which are either benign or malicious, is far less than the total number of clusters detected in this case study given a similar time period; we show an overview of the number of clusters detected in this case study in the initial 5 months of analysis for the year 2017 in Table 5.1. This significant difference in the number of clusters that have been analyzed and detected can either be attributed due to the fact that the original data source of *FluxBuster* did not have an extensive data set available or that *FluxBuster* was not able to analyze more data due to the limitation of the similarity matrix of the HCA algorithm. Although it is improbable that the *ISC/SIE* sensor data have the same high dimensionality of the *OpenINTEL* data set, it is also unlikely that the dimensionality of the initial data source limited *FluxBuster* in the amount of data it could process. Far more likely is that no more DNS records could be analyzed/clustered in a single period due to the processing limitation of the similarity matrix.

Since other flux-network detection methods that use some form of clustering are also limited by the processing restrictions of the used clustering algorithm, it may be possible to determine that there has not been a detection method for flux-networks that use clustering and which analyzes such a high dimensional data as was performed in this case study. Of course, we are aware that the novel *OpenINTEL* data set was published only in recent years, but it is still

Description	Number
Num. clusters	3.004.868
Num. malicious clusters $sim >= 0.00$	128.226
Num. malicious clusters $sim >= 0.58$	37.654
Num. 2LDs domains	209.248.200

Table 5.1: General characteristics of the first 5 months of data

remarkable that the methods discussed in this case study would not have been able to analyze the same dimensional data sets, thus further emphasizing the novelty of this particular case study.

5.1.3 2LD domain limitation

The *OpenINTEL* data set only contains records of the available 2LD domain names within a certain TLD. Besides these 2LD domain names, *OpenINTEL* also contains records for a minimal subset of known 3LD domain names, mainly in the form of the *www.* prefix for a 2LD domain name. We attribute this implementation to how the DNS protocol is implemented and of course considerations in the resource management required to generate the amount of data that is made daily available within *OpenINTEL*. This limitation does impact the results of this case study and the general possibility to use *OpenINTEL* as some form of a data source for a method for detecting malicious domain names being flux-network or some other form of malicious network.

For example, there are known DGAs that focuses on generating random 3LD domain names using a single 2LD domain name as part of a flux-network. It would not be possible to detect these types of flux-networks in this particular case study only because the required records would are not available in the *OpenINTEL* data set. The *OpenINTEL* data set receives the initial list of available 2LD from the TLDs but lower level domain names 3LDs will just not be known by the TLD and thus cannot be shared with *OpenINTEL*, and therefore the required information cannot be gathered. The only method for detecting these types of malicious 3LD domain names is using *pDNS* with the known deficit that the gathered data is only a subset of all the available domain names in the domain space that are accessed by the users of the network.

Since there currently does not exist a *pDNS* data set available for security researches for the *.nl* TLD, it is not possible to determine whether we missed flux-networks due to this limitation of only having access to 2LD domain names. Although there do exist DNS data sets which provide partial functionality of a *pDNS* data set, such as the *Entrada* [30] data set from SIDN, they do not provide all the required resource records that are necessary to implement a flux-network detection mechanism. Therefore this deficit of only being able to analyze 2LD domain names has to be taken into account for this case study because there currently does not exist a method which could resolve this particular issue.

5.1.4 Use of a known ground-truth

The flux-network detection results we describe in this thesis are heavily influenced by the entries in the ground-truth that we use to identify the malicious clusters. The use of ground-truths for the use flux-network detection has been very diverse in recent studies, some of the researches use it only for training the initial classifier algorithm while other researches used it as a direct input in the classification whether a domain is legitimate or malicious. This discussion on how to use ground-truth in the form of flux-network detection eventually led to the paper *On the ground truth problem of malicious DNS traffic analysis* by Stevanovic et al. [16].

Since this paper [16] shows a listing of the various ground-truth sources used by other flux-network detection researches, we use this paper to identify the ground-truth sources used for this case study. Unfortunately, most of the sources documented by Stevanovic et al.

[16] only contain very few domain names that were available to use for this study, thus 2LD domain names within the Netherlands TLD, as we show in Table 3.6. We mainly attribute this lack of available domain names to the fact that the majority of the sources have only very few known malicious domain names within the Netherlands TLD. The lack of any substantial listing of known malicious domain names within the available ground-truths with regards to the Netherlands TLD might already be an indication that there is a discrepancy between the malicious use of the Dutch infrastructure and the overall malicious use of domain names within the Netherlands TLD. Namely, that the Dutch infrastructure, the IP-space attributed to the Netherlands, is abused more by malicious actors than the fact that malicious actors actively use domain names related to the Netherlands TLD.

Eventually, the majority of the ground-truth entries came from a single source in the form of *NetCraft*. This source was made available by a partner organization of the University of Twente. It is interesting to note that although the majority of public sources on malicious domain names show only a minimal list of malicious Dutch domain names, the list provided by *NetCraft* did contain several thousand entries. This difference in the ground-truth size of the various sources is notable. Because of the distribution of the ground-truth entries, the *NetCraft* ground-truth source accounts for the majority of clusters that we classified as malicious.

Given the manual verification of the identified malicious clusters during the detection of either IP-flux or domain-flux networks, it was possible to verify the classification of the entries within the ground-truth. It was interesting to note that for the majority of domains which we verified, *NetCraft* ground-truth labeled them as malicious that are mostly related to phishing attacks. This identification might indicate an overall primary type of attacks on which this particular ground-truth source focuses. It is also interesting to note that there have been numerous occurrences of domain names that did not show any malicious behavior according to public resources, such as VirusTotal. Also, a substantial segment of domain names that we verified did show malicious behavior but which only a minimal number of anti-virus solutions identify as malicious. This behavior either means that the overall quality of the ground-truth entries of *NetCraft* is rather low and contains multiple false positives which other anti-virus solutions do not accept. Alternatively, it might indicate that the ground-truth source is of such high quality that it identifies more malicious behavior than any other available anti-virus solutions. It is not possible to determine which statement might be right and which might be false. It is only relevant to take into account that the quality of the ground-truth sources has a significant impact on the overall results of the current case study.

5.2 Detection results

The most prominent result of this case study is the lack of any results for this particular case study. Mainly the fact that the current methodology for detecting flux-networks did not result in any identified flux-network being either IP-flux or domain-flux. This result is in sharp contrast with the results of *FluxBuster*, on which detection methodology we primarily base our detection mechanism, which did result in the detection of 1.743 malicious domain names associated to flux-network behavior. The results from *FluxBuster* are especially interesting since in the current case study the amount of data we analyzed is significantly larger than the number of DNS records that were analyzed by *FluxBuster*. The data processed in the current case study is an increase of $5.123.509\%$ of the data processed by *FluxBuster* when compared to the number of malicious domain names they identify.

We should consider that part of the identification of malicious flux-networks by *FluxBuster* were done based on *guilty-by-association*. This method might be sufficient in some cases; however, the current case study shows that we could not apply this method to every result. A practical example of this is for the category 4 clusters which all contain some domain names that were associated with malicious behavior but to categorize all entries of the domain-list as malicious is just an invalid conclusion. Some category 4 clusters contained a hit-coverage as low as 0.002% which is merely too insufficient to categorize the entire cluster as malicious. However *FluxBuster* performed this approach anytime if any domain which IP-addresses is listed in a

known malicious ground-truth was sufficient evidence to classify the entire cluster as a flux-network. The current case study clearly shows that methodology is insufficient and that the results of the *FluxBuster* methodology is therefore not reliable. This approach is likely one of the reasons why there exists a difference in the results generated by *FluxBuster* and the current case study.

5.2.1 Limitation of focusing on single TLD

The current case study focuses on benign and malicious domain names within a single TLD, the Dutch ccTLD *.nl*. During the process of identifying flux-network by verifying the known malicious domains using VirusTotal, we show that in some cases there is a relation with the current malicious domain name and other domain names using other TLDs. Mainly this type of association was performed by VirusTotal which identified a malicious file that was distributed by several different domain names. In all cases when we encountered this type of relation, the domain names were associated with known malicious domains, that are not located in the Dutch ccTLD but were using an entirely different TLD. We, therefore, expect that the focus of only analyzing domains in the Dutch ccTLD could have severely impacted the overall results. Given the number of TLDs available within the *OpenINTEL* data set it is advised to include additional TLDs in any future flux-network detection mechanisms that use the *OpenINTEL* data set.

5.2.2 Lack of flux-networks

Again, the most profound result of the case study is the lack of any identification of either an IP-flux or domain-flux network. The lack of any result is in sharp contrast with previous methodologies which all have identified flux-networks in some matter. A possible explanation for this result is the fact that in this case study a novel methodology has been used to link relevant clusters to a single network which existed over a more extended period. None of the discussed researches have mentioned a similar method, and it could, therefore, be possible that other researches have identified single clusters as flux-network while analyzing the characteristics of the cluster over an extended period would result in a different conclusion.

Another possibility is that the overall use of flux-networks has declined in recent years. The researches and methodologies discussed in this case study were all published a minimal of 3 years ago and the majority of new methodologies around six years ago. Due to this difference in the time when the researchers performed the analyses, it might be possible that this particular use of malicious networks is not that common anymore.

The lack of the detection of any flux-network can also be attributed to the difference in the granularity of the gathering of DNS records by *OpenINTEL* when compared to *pDNS* data sets. It is a fact that the gathering of DNS record has a lower frequency in *OpenINTEL* and that therefore changes in the appointed IP-addresses by domain names are missed. Especially with regards to domain-flux detection, it is essential to verify the changes of associated IP-addresses pointed to a single domain name. A domain related to a domain-flux network has the known behavior of quickly changing associated IP-addresses. We should note, however, that although *OpenINTEL* does not detect the majority of changes to the DNS record, it does record every domain name in the domain name space once a day. Taking into account that we analyze the records from *OpenINTEL* in weekly segments, this means that at least seven changes of a single domain name can be recorded by *OpenINTEL* and thus any domain-flux behavior can still be identified. It is still unknown whether the reduced observations of IP changes had a significant impact on the detection of flux-networks.

The most prevalent theory is that these types of malicious networks do not prominently use the Netherlands ccTLD *.nl*. The argumentation is that it, in general, it is known that the Netherlands has an efficient IT-infrastructure and contains organizational structures such as the Dutch High Tech Crime Unit or National CERT which are efficient in sharing IT intelligence and taking appropriate actions on large malicious networks. So why should an administrator of these types of large malicious networks, such as flux-networks, use domain names located in the

Netherlands ccTLD which could effectively be taking down by the appropriate organizations. Primarily because in recent years the number of *generic top-level domains* (gTLDs) have multiplied. Registering domain names in these gTLDs has the exceptional benefit that they are not related to a particular country and that therefore it is more difficult for an affected country to take appropriate actions. Due to this reasoning, we expect that the Netherlands ccTLD is just not that much used by flux-network administrators and therefore no flux-networks have been detected in this case study. For future works, this theory might be validated by performing a similar analysis as documented by this study and then focus on all available TLDs within *OpenINTEL*.

Chapter 6

Conclusion

The case study in this paper revolves around the question of whether it was possible to *use a novel active DNS measurement to identify flux-networks, and its components, for the Netherlands TLD*? To fully answer this research topic, we divided the overall topic into three separate research questions. We answer the overall research topic by clarifying the following three research questions:

Can previously researched detection methods be applied to the OpenINTEL DNS measurements. If not, are they other methods suited for identifying these networks? Yes, it possible for previously researches detection methods to be applied in some form on the *OpenINTEL* DNS measurements. However, we could only apply a similar theoretical approach for *OpenINTEL* due to practical limitations, and so we were not able to fully implement the same methods to *OpenINTEL*. The problem with applying previous researched detection method to the *OpenINTEL* DNS measurements is the difference in the dimensionality of the DNS data sets. Previous researches have all used data sets based on *pDNS* data that do not contain the same number of records that are available in *OpenINTEL*. The variation in the dimensionality of the data sets is so significant that we cannot apply the same algorithms to the *OpenINTEL* data set.

This case study has shown that it is possible to apply different methodologies that have similar results as the methodologies applied by previous researches but which we apply to the high dimensional data set of *OpenINTEL*. Due to time constraints of performing this case study, it was deemed not possible to implement the classification methodology of the previous researches fully, so to identify potential flux-networks we use a known list of malicious ground-truth. Previous studies used this methodology of identifying known malicious networks using malicious ground-truths, and we, therefore, deemed it sufficient to use in the current case study.

Are the results of the flux-network detection system sufficiently reliable to get detailed characteristics of the identified flux-networks? Yes, the results of the currently implemented detection method are sufficiently reliable to get detailed overviews of the detected networks. The results of the various stages of the detection method, such as the clustering and identification process, are sufficiently trustworthy to get the detailed characteristics of the identified cluster such as the domain-list, overall similarity score, associated IP-addresses, hits in ground truth and more. Even though the clustering process is in some small matter affected by randomness, the resulting characteristics of the identified clusters are reliable. We can further emphasize this behavior by the fact that we also show that it is possible to associate multiple clusters to a single network which exists throughout the entire data set. It is, therefore, possible to monitor the changes of an identified network for a more extended period, which we showed in our case study for a year's worth of data.

Although the actual characteristics of the identified networks are reliable, we also show that the information gathered from the detection methodology is insufficient to correctly categorize a

network as a certain type of malicious network. We show that the current results are insufficient even to make the distinction of whether the resulting clusters are malicious or not. Previous researches such as *FluxBuster* by Perdisci et al. [3] used the *guilty-by-association* in order to categorize clusters as either malicious or benign. However, the results of this case study have shown that this type of labeling of clusters cannot be done by simple association because in the majority of identified clusters the argumentation of whether we can regard an entire cluster as malicious is difficult to make. We, therefore, determine that the current methodology does not contain sufficient verification to make a proper categorization of the type of cluster which has been identified by the detection method. Furthermore, since we have not identified any flux-network, it is not possible to determine the overall reliability of the detection method in detecting flux-networks.

Are there any disadvantages of, or limitations to, using active DNS measurement data from the OpenINTEL platform to the end of fast-flux detection? Yes, the current characteristics of the data that is stored by the *OpenINTEL* data set have a significant impact on the overall results of this type of detection method for flux-networks. In general, some of the attributes of the DNS record, which are heavily used by these types of detection systems such as TTL, are just not available in *OpenINTEL* increasing the difficulty in adequately categorizing specific domain names. Furthermore, although the *OpenINTEL* is a high-dimensional data set and contains a complete list of available domain names for the records TLDs, the number of data points gathered for each domain name is rather low when compared to *pDNS* data sets. This characteristic is due to the low granularity of the DNS records that are gathered only once a day for each specific domain name located in the *OpenINTEL* data set. This property means that domain names used by flux-networks, which are rapidly changing throughout the day, only shows a single distinct record and *OpenINTEL* does not record the majority of changes that the domain name makes. This behavior increases the overall difficulty of identifying domain names related to flux-networks.

Another disadvantage to using this new active DNS measurement for flux-network detection is the fact that it only records 2LD domain names except for the 3LD *www*. Any flux-network operating using 3LD domain names instead of 2LD cannot be detected by *OpenINTEL* due to this deficit. Depending on the division of flux-network clusters using either 2LD or 3LD or higher domain names, there is a vast majority of flux-networks that by definition cannot be detected by *OpenINTEL* given its current implementation.

Appendix A

Referenced malicious networks

id	start	end	domain_size	ip_list	score	hit_size	hit_coverage	category	countries	behavior
3104	2017-02-19	2017-02-25	$1,080$	176.34.97.79, 54.247.126.249, 176.34.118.127, 46.137.113.11, 54.217.229.91, 54.247.171.154, 46.137.83.158, 176.34.239.163, 54.246.112.186, 54.247.98.164, 46.137.89.76, 54.228.226.108	1.0	3	0.277778	3	IE, Unknown	Phishing
14289	2017-03-26	2017-04-01	$6,125$	176.34.104.207, 54.246.125.136 54.247.82.16, 176.34.115.136, 176.34.188.134, 109.237.222.171, 54.246.101.195, 104.16.91.230, 104.16.89.230, 176.34.117.3, 46.137.86.155, 104.16.90.230, 104.16.88.230, 176.34.118.127 104.16.87.230, 46.137.188.130, 176.34.109.23, 54.246.102.48,	0.625	7	0.114286	4	IE, NL, US	Phishing

Continued on next page

id	start	end	domain_size	ip_list	score	hit_size	hit_coverage	category	countries	behavior
16831	2017-09-17	2017-09-23	10	192.185.109.119, 185.182.57.80	1.0	3	30.0	1	NL, US	Malware
17198	2017-09-24	2017-09-30	4	94.231.103.144	1.0	2	50.0	1	DE	Malware
22931	2017-08-20	2017-08-26	306	136.144.129.81	1.0	7	2.287582	2	NL	Phishing
23330	2017-01-01	2017-01-07	10	149.210.186.191	1.0	2	20.0	1	NL	Phishing
25219	2017-06-11	2017-06-17	6,470	72.52.4.121	1.0	2	0.030912	4	US	Malware
41211	2017-02-26	2017-03-04	54	178.22.60.93, 54.93.217.168	1.0	17	31.481481	2	DE, NL	Phishing
42105	2017-12-25	2017-12-31	4	94.231.103.144	1.0	2	50.0	1	DE	Malware
57030	2017-10-15	2017-10-21	52	178.22.60.93, 192.190.221.247	1.0	17	32.692308	2	NL, US	Phishing
59221	2017-06-25	2017-07-01	2,075	176.34.97.79, 54.247.126.249, 54.247.82.16, 176.34.115.136, 54.228.228.48, 176.34.229.98, 79.125.117.19, 46.137.103.214, 83.137.194.93, 54.246.127.20, 46.137.161.182, 54.246.113.168 54.228.226.142, 54.247.105.232 46.137.93.184, 176.34.254.234 54.75.254.125	0.6	5	0.240964	3	IE, NL	Phishing
134839	2017-07-09	2017-07-15	678	31.7.4.177	1.0	2	0.294985	3	NL	Phishing

Appendix B

Flux network detection algorithm for OpenINTEL

B.1 Main driver for spark application

```
package org.utwente.detection

import org.apache.log4j.Logger
import java.lang.Math
import java.net.InetAddress
import scala.collection.mutable.WrappedArray
import scala.collection.immutable.Set
import scala.collection.mutable.ListBuffer
import scala.collection.mutable.MutableList
import org.apache.log4j.Level
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.sql._
import org.apache.spark.SparkConf
import org.apache.spark.mllib.linalg.{Vectors, SparseVector}
import org.apache.spark.sql.functions.not
import java.io._

import scala.util.control.Breaks._

import java.security.MessageDigest
import java.nio.ByteBuffer
import org.apache.spark.SparkContext._
import org.apache.spark.mllib.linalg.{Vectors, SparseVector}
import org.apache.spark.SparkContext
import org.apache.spark.rdd.RDD.rddToPairRDDFunctions
import org.apache.spark.rdd.RDD

// Import Joda time dependencies
import org.joda.time.Days
import org.joda.time.DateTime
import org.joda.time.format.DateTimeFormat

// Import avro files
import com.databricks.spark.avro._

// Export as JSON
import org.json4s._
import org.json4s.JsonDSL._
import org.json4s.native.JsonMethods._

object Main_Driver_Fluxnetwork_research_OpenINTEL {

   /**
    * Calculate the next prime number for given integer
    *
```

```
     * g: Int -> Calculate prime number equal or higher than this Integer
     */
    def nextPrime(g: Long): Int = {

      var n = g
      var isPrime = false
      var m = Math.ceil(Math.sqrt(n)).toInt
      var start = 3
      if (n % 2 == 0) {
          n = n + 1
      }
      while (!isPrime) {
          isPrime = true
          breakable { for (i <- start to m by 2) {
              if (n % i == 0) {
                  isPrime = false
                  break
              }
          }}
          if (!isPrime) {
              n = n + 2
          }
      }
      return n.toInt
    }

    /**
     * Change IP address to long representative
     *
     * dottedIP: String -> String IPv4 address (10.10.10.10)
     */
    def IPv4ToLong(dottedIP: String): Long = {
      val addrArray: Array[String] = dottedIP.split("\\.")
      var num: Long = 0
      var i: Int = 0
      while (i < addrArray.length) {
        val power: Int = 3 - i
        num = num + ((addrArray(i).toInt % 256) * Math.pow(256, power)).toLong
        i += 1
      }
      num
    }

    /**
     * Change Long to IP address
     *
     * ip: Long -> Long number (168430090L)
     */
    def LongToIPv4 (ip : Long) : String = {
      val bytes: Array[Byte] = new Array[Byte](4)
      bytes(0) = ((ip & 0xff000000) >> 24).toByte
      bytes(1) = ((ip & 0x00ff0000) >> 16).toByte
      bytes(2) = ((ip & 0x0000ff00) >> 8).toByte
      bytes(3) = (ip & 0x000000ff).toByte
      InetAddress.getByAddress(bytes).getHostAddress()
    }

    /**
     * Calculate Jaccard Similarity, 0.0 not similar, 1.0 equal
     */
    def jaccardSet(a: Set[String], b: Set[String]): Double = {
      return a.intersect(b).size / a.union(b).size.doubleValue
    }


    def getDaysInbitween(startDate: DateTime, endDate: DateTime): MutableList[
        DateTime] = {
       // Get list of dates between startDate and endDate
       var days = Days.daysBetween(startDate.withTimeAtStartOfDay(),
```

```
                                                endDate.withTimeAtStartOfDay() ).
                                                    getDays()

    var inbitween_days = MutableList[DateTime]()
    var hdfs_urls = Seq[String]()
    for ( i <- 0 to days) {
      var curDate = startDate.plusDays(i)
      inbitween_days += curDate
    }

    inbitween_days
  }

  def getDataFrame(sqc: org.apache.spark.sql.SQLContext, date: DateTime, log:
      Logger): DataFrame = {
    import sqc.implicits._
    var curYear = date.getYear()
    var curDay = "%02d".format(date.getDayOfMonth())
    var curMonth = "%02d".format(date.getMonthOfYear())

    var hdfs_url = f"hdfs://openintel/user/openintel/nl_parquet/year=$curYear
        /month=$curMonth/day=$curDay"
    return sqc.read.parquet(hdfs_url)
      .filter($"query_type" === "A")
      .filter($"response_type" === "A")
      .filter(not($"query_name".startsWith("mail.")))
      .select("query_name","ip4_address")
  }

  /**
   * Main function
   */
  def main(args: Array[String]) = {

   var i = 0

   // How many minhash functions for LSH
   val mr = 5000000
   // How many rows for  LSH
   val nr = 10
   // How many buckets for LSH
   val nb = 210
   // How many cores should be used
   val partitions = args(2).toInt

   // Start data of analysis
   val strStartDate = args(0)
   // Until date of analysis
   val strEndDate = args(1)

   // Disable extensive logging
   System.setProperty("spark.ui.showConsoleProgress", "false");
   Logger.getLogger("org").setLevel(Level.WARN)
   Logger.getLogger("akka").setLevel(Level.WARN)

   // Specify logger and default DateTimeFormat
   val fmt = DateTimeFormat.forPattern("yyyy/MM/dd")
   val log = Logger.getLogger("detection")
   implicit val formats = DefaultFormats

   log.info("Starting Flux-network detection comparison (pDNS)")

   // Parse str input to DateTime objects
   val startDate = DateTime.parse(strStartDate, fmt)
   val endDate = DateTime.parse(strEndDate, fmt)

   log.info(s"Run analysis from [${startDate}] to [${endDate}]")

   //Start the Spark context, use $cores number of cores
   val conf = new SparkConf()
```

```
      .setAppName("LSH")
      .set("spark.sql.parquet.binaryAsString", "True")

    // Create Spark contexts
    val sc = new SparkContext(conf)
    val sqc = new org.apache.spark.sql.SQLContext(sc)
    import sqc.implicits._

    log.info("Loaded spark contexts")

    // Read ground truth from input list, and create key,value pairs with the same value
    val rdd_ground_truth = sc.textFile("hdfs://openintel/user/jonkerm/malicious_fqdn.lst").map( x => (x, x))
    // Collect ground_truth RDD as HashMap to Driver, and broadcast value to all nodes
    val ground_truth = sc.broadcast(rdd_ground_truth.collectAsMap()).value

    // Get list of dates between startDate and endDate
    var days = Days.daysBetween(startDate.withTimeAtStartOfDay(),
                                endDate.withTimeAtStartOfDay() ).getDays()

    // Determine alle days between start and end, given as input
    var inbitween_days = MutableList[DateTime]()
    var hdfs_urls = Seq[String]()
    for ( i <- 0 to days) {
      var curDate = startDate.plusDays(i)
      inbitween_days += curDate
    }

    // Get the starting date, and get the dataframe for that specific date
    var date = inbitween_days(0)
    var queries = getDataFrame(sqc, date, log)

    // Broadcast, MutableList of inbitween days to all nodes
    val b_inbitween_days = sc.broadcast(inbitween_days).value

    // Loop through all inbitween days, starting one day after startdate until enddate
    // Get dataframe for each day and join them all in large DataFrame
    for ( i <- 1 to b_inbitween_days.size-1 ) {
      var curDate = inbitween_days(i)
      var cur_df = getDataFrame(sqc, curDate, log)

      // Join all DataFrame of each day into large Dataframe. After loop queries contain records
      // from startdate until enddate
      queries = queries.unionAll(cur_df)
    }

    // Create RDD of all data (Domain, IP), domain being Key
    // 1) Change queries DataFrame to RDD
    // 2) Partition RDD into several chunks by number $partitions, which will be divided over nodes
    // 3) Makes sure that both columns are Strings, by casting explicit to String
    val rdd = queries.rdd.repartition(partitions).map(row => (row(0).toString, row(1).toString))
    log.info("Read parquet files")
    log.info(s"Total input count: <${rdd.count()}>")

    /* LSH algoritme cannot work with strings. Therefore each IP address is mapped to a unique integer. So for example, it will be mapped that
     * domainA will have IPs (133.8.10.3, 233.158.9.1) which are mapped to (1567, 102). Mapping from IPs to integer is shown below
     */

    // Create IP to Integer mapping
```

```
    // 1) Create Map (IP, 1.0) and reduceByKey (IP), basically efficient
        distinct to get mapping of unique IPs
241 // 2) Add index to unique records of ((IP, 1.0), index)
    // 3) Change mapping to (IP, index)
243 // 4) collect rdd as HashMap (IP, Index), IP being key
    val IPid = rdd.map(row => (row._2, 1.0)).reduceByKey((m, n) => m).
        zipWithIndex.map(row => (row._1._1, row._2.toInt)).collectAsMap
245 // Broadcast HashMap to all nodes
    val bIPid = sc.broadcast(IPid).value
247
    // Group IPs based on domain names, uses set to prevent double IP entries
        then transfer to List
249 // 1) Aggregate by Key, so IPs are all added to Set based by key DomainName
    val grouped_set = rdd.aggregateByKey(Set[String]())((set: Set[String],
        value: String) => set + value, (set1,set2) => set1 ++ set2)
251 // Change (Domain, Set(IPs)) to (Domain, List(IPs))
    // Dataset should be cached, because data is justed to generate points &
        vectors so indexes should be identical in both sets
253 // See lines #263 & #274
    val grouped_rdd = grouped_set.map(r => (r._1, r._2.toList)).cache()
255 // 1) Map (Domain, List(IPs)) to (Integer(len(List(IPs)))) and sum all list
         sizes
    val n_ips = grouped_set.map(r => r._2.size).reduce(_+_)
257
    log.info(s"Grouped input domain count: <${grouped_rdd.count()}>")
259 log.info(s"Grouped input IPs count: <${n_ips}>")

261 // Map (Domain, List(IPs)) to (Index, (Domain, List(IPs))), this will be
        used to map cluster entries to actual data
    // As shown by LSH module developer at: https://github.com/mrsqueeze/spark-
        hash/blob/master/src/main/scala/com/invincea/spark/hash/OpenPortDriver.
        scala#L36
263 // Cache points, this the reference to restore vectors to IP-mappings so
        index cannot be changed!
    val points = grouped_rdd.zipWithIndex().map(x => x.swap).cache()
265
    // Determine next_prime by getting number of unique IPs. Used by
        SparseVector, as max size!
267 val size = bIPid.size
    val next_prime = nextPrime(size+1)
269
    log.info(f"Next prime determined: <$next_prime> for size: <$size>")
271 log.info("Creating SparseVector list for LSH algorithm")

273 // Create SparseVector lists for every domain name containing IP -> ID
        mappings
    // Map (Domain, List(IPs) to List((IP-> ID mapping, 1.0), ...)
275 val boolean_IP_map = grouped_rdd.map(r => (r._2.map(IP => (bIPid(IP), 1.0))
        ))
    // Change the boolean_IP_map, to rows of SparseVectors
277 // List((ID, 1.0), ..) to List(SparseVectors)
    val vctrs = boolean_IP_map.map(row => Vectors.sparse(next_prime, row).
        asInstanceOf[SparseVector])
279 log.info(s"Initial vector size: <${vctrs.count()}>")

281 // vctrs is the largest data structure which will be parsed signifanctly
        for the clustering
    // Num IPv4 = 4294967294, guessed that the amount of IPv4 in NL is ~5%.
        Each IP contains 4B integer and 8B double
283 // 4294967294 * 0.05 * 4 * 8 / (1024^4) ~= 6.4G
    // The size of vctrs is thus around 6.4GB of data p/d
285
    // Perform LSH clustering, based on predefined parameters. The parameters
        have been previously determined, and
287 // are the steps used to determine values have been described in
        methodology.
    log.info("Starting LSH clustering algorithm")
289 var lsh = new LSH(data = vctrs, p = next_prime, m = mr, numRows = nr,
        numBands = nb, minClusterSize = 3)
    var model = lsh.run()
```

```
291     var n_clusters = model.clusters.count()
        log.info(f"LSH clustering finished, number clusters <$n_clusters>")
293
        /* The model generate several datasets:
295      * clusters: (ClusterID, CompactBuffer(SparseVectors))
         * vector_cluster: Mapping from VectorIndex to ClusterID, (VectorIndex,
             ClusterID)
297      * cluster_vector: Mapping from ClusterID to VectorIndex, (ClusterID,
             VectorIndex)
         * score: Mapping ClusterID with similarity score, (ClusterID, score)
299      */

301     // Map domain names to found clusters
        // 1) Join points (index, (domain, List(IPs)) with ClusterID to (index,((
            domain, List(IPs)), ClusterID))
303     // 2) Redefine data with map (ClusterID, (domain, List(IPs)))
        var points_clusters = points.join(model.vector_cluster).map(x => (x._2._2,
            x._2._1))
305     n_clusters = points_clusters.count()
        log.info(f"Initial size points_clusters <$n_clusters>")
307
        /* Validate LSH cluster against known ground-truth
309      * and create list of keys of malicious clusters
         */
311     log.info("Starting verification of ground-truth")

313     // Loop through points_cluster and create a data set containing the
            ClusterID and domain name which created hit
        // 1) Filter only points_clusters which have a hit in ground-truth, domain
            is compared to Key in Ground-truth which is domain
315     // 2) Map only hits to ground_truth_hits as (ClusterID, (1, domain))
        // Ground_truth_hits will be used to link gt_hits to cluster, indexes
            should remain the same in order to function, so cache!
317     var ground_truth_hits = points_clusters.filter{ row => {
          var domain_name = row._2._1
319       ground_truth.contains(domain_name)
        }}.map(x => (x._1, (1, x._2._1))).cache()
321
        // Use the ground_truth_hits to determine which Cluster has enough hits to
            categorise as malicious
323     // 1) Map data to (ClusterID, 1)
        // 2) Sum up data using reduceByKey resulting in (ClusterID, sum)
325     // 3) Filter out (ClusterID, sum) where sum is lower than 2
        val malicious_cluster_keys = ground_truth_hits.map(x => (x._1, x._2._1)).
            reduceByKey(_ + _).filter(_._2 >= 2)
327
        // Create HashMap from list of hits and broadcast to all nodes
329     // 1) Map data to (ClusterID, domain)
        // 2) AggregateByKey (ClusterID, Set(domains))
331     //
        val rdd_malicious_domains = ground_truth_hits.map(x => (x._1, x._2._2)).
            aggregateByKey(Set[String]())((set: Set[String], value: String) => set +
             value, (set1,set2) => set1 ++ set2).cache()
333     // Collect RDD as HashMap with ClusterID as key, and broadcast
        val malicious_domains = sc.broadcast(rdd_malicious_domains.collectAsMap()).
            value
335
        n_clusters = malicious_cluster_keys.count()
337     log.info("End verification of ground-truth")
        log.info(f"Number of malicious clusters with sim >= 0.00: <$n_clusters>")
339
        // Get characterstics of cluster sizes, by determing CompactBuffer size a.k
            .a as number of domains in cluster
341     // 1) Map (ClusterID, CompactBuffer(SparseVector)) to (size CompactBuffer)
        // Cache cluster_size, in order to prevent unnecessary recalculation on
            mean/min/max.
343     var cluster_sizes:RDD[Double] = model.clusters.map(x => x._2.size.toDouble)
            .cache()
        // Get min, max & mean from cluster sizes
```

```
345     log.info(s"Description cluster sizes: AVG<${cluster_sizes.mean()}>, MIN<${
            cluster_sizes.min()}>, MAX<${cluster_sizes.max()}>")

347     // Join the model.cluster and model.cluster_vector with
            malicious_cluster_keys so that only ClusterID which
        // have enough hits in GT will be used in the further processing
349     //
        // 1) (ClusterID, hits) joined with (ClusterID, CompactBuffer(SparseVector)
            ) results in
351     //    (ClusterID, (hits, CompactBuffer(SparseVector)
        // 2) Use map to redefine data as (ClusterID, CompactBuffer(SparseVector))
353     // New mapping of only malicious clusters should be cached. Indexes, as
            mentioned in LSH.scala, should not be changed
        log.info(s"Description cluster sizes: AVG<${cluster_sizes.mean()}>, MIN<${
            cluster_sizes.min()}>, MAX<${cluster_sizes.max()}>")
355     model.clusters = malicious_cluster_keys.join(model.clusters).map(x => (x._1
            , x._2._2)).cache()

357     // Similar approach with model.cluster_vector
        // 1) (ClusterID, hits) joined with (ClusterID, VectorIndex) results in (
            ClusterID, (hits, VectorIndex))
359     // 2) Use map to redefine data as (ClusterID, VectorIndex)
        // Use cache, as shown in line #353
361     model.cluster_vector = malicious_cluster_keys.join(model.cluster_vector).
            map(x => (x._1, x._2._2)).cache()

363     // Filter for malicious clusters with sim >= 0.58
        model = model.filter(0.58)
365
        // Gather all the information from the clusters and combine in single map
367     // 1) Join points (index, (domain, List(IPs)) with ClusterID to (index,((
            domain, List(IPs)), ClusterID))
        // 2) Redefine data with map to (ClusterID, (domain, List(IPs)))
369     // 3) Group all entries for same ClusterID using groupByKey results in: (
            ClusterID, CompactBuffer((domain, List(IPs))))
        // 4) Join data with model.scores, (ClusterID, similarity_score) to (
            ClusterID, (CompactBuffer((domain, List(IPs))), similarity_score))
371     var malicious_clusters = points.join(model.vector_cluster).map(x => (x._2.
            _2, x._2._1)).groupByKey().join(model.scores)

373     n_clusters = malicious_clusters.count()
        log.info(f"Number of malicious clusters with sim >= 0.58: <$n_clusters>")
375
        // Export data from malicious clusters using JSON format
377     // Loop through all clusters in malicious_cluster dataset
        // (ClusterID, (CompactBuffer((domain, List(IPs))), similarity_score))
379     var json_results = malicious_clusters.map(cluster => {

381       val IPs: scala.collection.mutable.Set[String] = scala.collection.mutable.
              Set()
          val cluster_id = cluster._1
383       // get list of all domain names in cluster, by mapping through all
              CompactBuffer list of (domain, List(IPs))
          val domains = cluster._2._1.map(x => x._1)
385       // get set of IPs in cluster, by mapping through all CompactBuffer list
              of (domain, List(IPs)) and adding list to set
          cluster._2._1.foreach(x => IPs ++= x._2)
387
          val lfmt = DateTimeFormat.forPattern("yyyy/MM/dd")
389
          val json: JObject =
391         ("id" ->  cluster_id) ~
            ("start" ->  lfmt.print(startDate)) ~
393         ("end" ->  lfmt.print(endDate)) ~
            ("ip_list" -> IPs) ~
395         ("score" -> cluster._2._2) ~
            ("domain_list" -> domains) ~
397         // Get list of hits from malicious_domain HashMap using ClusterID as
                key
            ("hit_list" -> malicious_domains(cluster_id))
```

```
399
        compact(render(json))
401     }).coalesce(1) // Correlate all RDD parts in 1 partition, to create single
            output file

403     // Write to sparkie dir because of kerberos authentication
        val out_fmt = DateTimeFormat.forPattern("yyyyMMdd")
405     json_results.saveAsTextFile(s"hdfs://openintel/user/jonkerm/
            rk_fluxnetwork_${out_fmt.print(startDate)}_${out_fmt.print(endDate)}.
            json")

407     log.info("Flux-network detection comparison finished (pDNS)")
        sc.stop()
409   }
}
```

Listing B.1: Main Spark driver for identification method

B.2 LSH clustering algorithm

B.2.1 LSH clustering algorithm

```
package org.utwente.detection
2
import org.apache.spark.mllib.linalg.SparseVector
4 import org.apache.spark.rdd.RDD
import scala.collection.mutable.ListBuffer
6 import org.apache.spark.SparkContext._

8 class LSH(data : RDD[SparseVector], p : Int, m : Int, numRows : Int, numBands :
      Int, minClusterSize : Int) extends Serializable {

10   /** run LSH using the constructor parameters */
    def run() : LSHModel = {
12
      /*
14     * WARNING: All actions steps in this model should be cached. LSH algoritme
           runs on restoring data points to clusters via indexes. If
       * certain steps are run again there is a chance that the indexed are
           different. This result in incorrect output.
16     */

18    //create a new model object
      val model = new LSHModel(p, m, numRows)
20
      //preserve vector index
22    val zdata = data.zipWithIndex().cache()

24    //compute signatures from matrix
      // - hash each vector <numRows> times
26    // - position hashes into bands. we'll later group these signature bins and
          has them as well
      //this gives us ((vector idx, band#), minhash)
28    val signatures = zdata.flatMap(v => model.hashFunctions.flatMap(h => List
          (((v._2, h._2 % numBands),h._1.minhash(v._1))))).cache()

30    //reorganize data for shuffle
      //this gives us ((band#, hash of minhash list), vector id)
32    //groupByKey gives us items that hash together in the same band
      model.bands = signatures.groupByKey().map(x => ((x._1._2, x._2.hashCode), x
          ._1._1)).groupByKey().cache()
34
      //we only want groups of size >= <minClusterSize>
36    //(vector id, cluster id)
      model.vector_cluster = model.bands.filter(x => x._2.size >= minClusterSize)
          .map(x => x._2.toList.sorted).distinct().zipWithIndex().map(x => x._1.
          map(y => (y.asInstanceOf[Long], x._2))).flatMap(x => x.grouped(1)).map(x
           => x(0)).cache()
```

```
38
      //(cluster id, vector id)
40    model.cluster_vector = model.vector_cluster.map(x => x.swap).cache()

42    //(cluster id, List(vector))
      model.clusters = zdata.map(x => x.swap).join(model.vector_cluster).map(x =>
          (x._2._2, x._2._1)).groupByKey().cache()
44
      model
46  }

48  /** compute a single vector against an existing model */
    def compute(data : SparseVector, model : LSHModel, minScore : Double) : RDD[(
        Long, Iterable[SparseVector])] = {
50    model.clusters.map(x => (x._1, x._2++List(data))).filter(x => jaccard(x._2
        .toList) >= minScore)
    }
52
    /** compute jaccard between two vectors */
54  def jaccard(a : SparseVector, b : SparseVector) : Double = {
      val al = a.indices.toList
56    val bl = b.indices.toList
      al.intersect(bl).size / al.union(bl).size.doubleValue
58  }

60  /** compute jaccard similarity over a list of vectors */
    def jaccard(l : List[SparseVector]) : Double = {
62    l.foldLeft(l(0).indices.toList)((a1, b1) => a1.intersect(b1.indices.toList.
        asInstanceOf[List[Nothing]])).size /
      l.foldLeft(List())((a1, b1) => a1.union(b1.indices.toList.asInstanceOf[List
        [Nothing]])).distinct.size.doubleValue
64  }

66}
```

Listing B.2: LSH clustering algorithm for Spark (1)

B.2.2 LSH clustering algorithm model

```
  package org.utwente.detection
2
  import org.apache.log4j.Logger
4 import org.apache.spark.mllib.linalg.SparseVector
  import org.apache.spark.rdd.RDD
6 import scala.collection.mutable.ListBuffer
  import org.apache.spark.SparkContext._
8

10 class LSHModel(p : Int, m : Int, numRows : Int) extends Serializable {

12   /** generate rows hash functions */
     private val _hashFunctions = ListBuffer[Hasher]()
14   for (i <- 0 until numRows)
       _hashFunctions += Hasher.create(p, m)
16   final val hashFunctions : List[(Hasher, Int)] = _hashFunctions.toList.
       zipWithIndex

18   /** the signature matrix with (hashFunctions.size signatures) */
     var signatureMatrix : RDD[List[Int]] = null
20
     /** the "bands" ((hash of List, band#), row#) */
22   var bands : RDD[((Int, Int), Iterable[Long])] = null

24   /** (vector id, cluster id) */
     var vector_cluster : RDD[(Long, Long)] = null
26
     /** (cluster id, vector id) */
28   var cluster_vector : RDD[(Long, Long)] = null
```

```
30    /** (cluster id, List(Vector) */
      var clusters : RDD[(Long, Iterable[SparseVector])] = null
32
      /** jaccard cluster scores */
34    var scores : RDD[(Long, Double)] = null

36    /** filter out scores below threshold. this is an optional step.*/
      def filter(score : Double ) : LSHModel = {
38      //compute the jaccard similarity of each cluster
        scores = clusters.map(row => (row._1, jaccard(row._2.toList)))
40
        val scores_filtered = scores.filter(x => x._2 >= score)
42      val clusters_filtered = scores_filtered.join(clusters).map(x => (x._1, x._2
            ._2))
        val cluster_vector_filtered = scores_filtered.join(cluster_vector).map(x =>
             (x._1, x._2._2))
44      scores = scores_filtered.cache()
        clusters = clusters_filtered.cache()
46      cluster_vector = cluster_vector_filtered.cache()
        vector_cluster = cluster_vector.map(x => x.swap).cache()
48      this
      }
50
      /** compute jaccard similarity over a list of vectors */
52    def jaccard(l : List[SparseVector]) : Double = {
        l.foldLeft(l(0).indices.toList)((a1, b1) => a1.intersect(b1.indices.toList.
            asInstanceOf[List[Nothing]])).size /
54      l.foldLeft(List())((a1, b1) => a1.union(b1.indices.toList.asInstanceOf[List
            [Nothing]])).distinct.size.doubleValue
      }
56
      //def compare(SparseVector v) : RDD
58

60  }
```

Listing B.3: LSH clustering algorithm for Spark (1)

Appendix C

Bibliography

[1] R. van Rijswijk-Deij, M. Jonker, A. Sperotto, and A. Pras. A high-performance, scalable infrastructure for large-scale active dns measurements. *IEEE Journal on Selected Areas in Communications*, 34(6):1877–1888, June 2016. ISSN 0733-8716. doi: 10.1109/JSAC.2016.2558918.

[2] Mattijs Jonker and Anna Sperotto. Measuring exposure in ddos protection services. In *Network and Service Management (CNSM), 2017 13th International Conference on*, pages 1–9. IEEE, 2017.

[3] R. Perdisci, I. Corona, and G. Giacinto. Early detection of malicious flux networks via large-scale passive dns traffic analysis. *IEEE Transactions on Dependable and Secure Computing*, 9(5):714–726, 2012.

[4] Symantec. Internet security threat report 2016. 2016. URL https://www.symantec.com/security-center/threat-report.

[5] L. Bilge, S. Sen, D. Balzarotti, E. Kirda, and C. Kruegel. Exposure: A passive dns analysis service to detect and report malicious domains. *ACM Transactions on Information and System Security*, 16(4), 2014. doi: 10.1145/2584679.

[6] H. Choi and H. Lee. Identifying botnets by capturing group activities in dns traffic. *Computer Networks*, 56(1):20–33, 2012. doi: 10.1016/j.comnet.2011.07.018. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-84655163180&doi=10.1016%2fj.comnet.2011.07.018&partnerID=40&md5=1f93b48f8540305133aae7fef499cd89. cited By 57.

[7] E. Passerini, R. Paleari, L. Martignoni, and D. Bruschi. Fluxor: Detecting and monitoring fast-flux service networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5137 LNCS: 186–206, 2008. doi: 10.1007/978-3-540-70542-0_10.

[8] A. Berger and W.N. Gansterer. Modeling dns agility with dnsmap. pages 3153–3158, 2013. doi: 10.1109/INFCOM.2013.6567130.

[9] J. Lee and H. Lee. Gmad: Graph-based malware activity detection by dns traffic analysis. *Computer Communications*, 49:33–47, 2014. doi: 10.1016/j.comcom.2014.04.013.

[10] B. Rahbarinia, R. Perdisci, and M. Antonakakis. Segugio: Efficient behavior-based tracking of malware-control domains in large isp networks. volume 2015-September, pages 403–414, 2015. doi: 10.1109/DSN.2015.35.

[11] T Holz, C Gorecki, K Rieck, and F.C. Freiling. Measuring and detecting fast-flux service networks. *NDSS*, 2008.

[12] S. Yadav, A.K.K. Reddy, A.L. Narasimha Reddy, and S. Ranjan. Detecting algorithmically generated domain-flux attacks with dns traffic analysis. *IEEE/ACM Transactions on Networking*, 20(5):1663–1677, 2012. doi: 10.1109/TNET.2012.2184552.

[13] Alex Chiu and Angel Villegas. Threat spotlight: Dyre/dyreza: An analysis to discover the dga, 2015. URL https://blogs.cisco.com/security/talos/threat-spotlight-dyre.

[14] F. Weimer. Passive dns replication. 2005.

[15] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. ISBN 0-13-022278-X.

[16] M. Stevanovic, J.M. Pedersen, A. D'Alconzo, S. Ruehrup, and A. Berger. On the ground truth problem of malicious dns traffic analysis. *Computers and Security*, 55:142–158, 2015. doi: 10.1016/j.cose.2015.09.004.

[17] Reza Bosagh Zadeh and Gunnar Carlsson. Dimension independent matrix square using mapreduce. *CoRR*, abs/1304.1467, 2013. URL http://arxiv.org/abs/1304.1467.

[18] Hisashi Koga, Tetsuo Ishibashi, and Toshinori Watanabe. Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing. *Knowledge and Information Systems*, 12(1):25–53, May 2007. ISSN 0219-3116. doi: 10.1007/s10115-006-0027-5. URL https://doi.org/10.1007/s10115-006-0027-5.

[19] J. Nazario and T. Holz. As the net churns: Fast-flux botnet observations. pages 24–31, 2008. doi: 10.1109/MALWARE.2008.4690854.

[20] I. Khalil, T. Yu, and B. Guan. Discovering malicious domains through passive dns data graph analysis. pages 663–674, 2016. doi: 10.1145/2897845.2897877.

[21] Evan Cooke, Farnam Jahanian, and Danny McPherson. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Proceedings of the Steps to Reducing Unwanted Traffic on the Internet on Steps to Reducing Unwanted Traffic on the Internet Workshop*, SRUTI'05, pages 6–6, Berkeley, CA, USA, 2005. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251282.1251288.

[22] Geoip2. URL https://dev.maxmind.com/geoip/geoip2/geolite2/.

[23] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *USENIX Security Symposium*, pages 1092–1110, 2017.

[24] Max Kerkers, José Jair Santanna, and Anna Sperotto. Characterisation of the kelihos.b botnet. In Anna Sperotto, Guillaume Doyen, Steven Latré, Marinos Charalambides, and Burkhard Stiller, editors, *Monitoring and Securing Virtualized Networks and Services*, pages 79–91, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-43862-6.

[25] M. Stevanovic, J.M. Pedersen, A. D'Alconzo, and S. Ruehrup. A method for identifying compromised clients based on dns traffic analysis. *International Journal of Information Security*, pages 1–18, 2016. doi: 10.1007/s10207-016-0331-3. cited By 1; Article in Press.

[26] Wablieft-corpus, 2018. URL https://ivdnt.org/downloads/taalmaterialen/tstc-wablieft-corpus.

[27] Dutch dictionary, 2016. URL https://www.opentaal.org/.

[28] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001. ISSN 1573-0565. doi: 10.1023/A:1010933404324. URL https://doi.org/10.1023/A:1010933404324.

[29] Roland Martijn van Rijswijk-Deij. *Improving DNS Security: A Measurement-Based Approach*. University of Twente, 2017.

[30] Maarten Wullink, Giovane C. M. Moura, Muller, M, and Cristian Hesselman. ENTRADA: a High Performance Network Traffic Data Streaming Warehouse. In *Network Operations and Management Symposium (NOMS), 2016 IEEE (to appear)*, April 2016.

[31] A. Pras, J.J. Santanna, J. Steinberger, and A. Sperotto. Ddos 3.0 - how terrorists bring down the internet. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9629:1–4, 2016. doi: 10.1007/978-3-319-31559-1_1.

[32] M. Jonker and A. Sperotto. Mitigating ddos attacks using openflow-based software defined networking. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9122:129–133, 2015. doi: 10.1007/978-3-319-20034-7_13.

[33] J.J. Santanna, R. Van Rijswijk-Deij, R. Hofstede, A. Sperotto, M. Wierbosch, L.Z. Granville, and A. Pras. Booters - an analysis of ddos-as-a-service attacks. pages 243–251, 2015. doi: 10.1109/INM.2015.7140298.

[34] O. Pomorova, O. Savenko, S. Lysenko, A. Kryshchuk, and K. Bobrovnikova. Anti-evasion technique for the botnets detection based on the passive dns monitoring and active dns probing. *Communications in Computer and Information Science*, 608:83–95, 2016. doi: 10.1007/978-3-319-39207-3_8.

[35] J. Ruohonen, S. Scepanovic, S. Hyrynsalmi, I. Mishkovski, T. Aura, and V. Leppanen. The black mark beside my name server: Exploring the importance of name server ip addresses in malware dns graphs. pages 264–269, 2016. doi: 10.1109/W-FiCloud.2016.61. cited By 0.

[36] Y. Nakamura, S. Kanazawa, H. Inamura, and O. Takahashi. Classification of unknown web sites based on yearly changes of distribution information of malicious ip addresses. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–4, Feb 2018. doi: 10.1109/NTMS.2018.8328683.