UiO : Department of Informatics
University of Oslo

Mitigating DDoS attacks using data mining and density-based geographical clustering

Madeleine Victoria Kongshavn Rønning

Master's Thesis Spring 2017

Mitigating DDoS attacks using data mining and density-based geographical clustering

Madeleine Victoria Kongshavn Rønning

May 23, 2017

Acknowledgement

I would like to express my sincere gratitude to my supervisors Hårek Haugerud and Anis Yazidi for immense help and support during this thesis. It was a great opportunity to work with you and hopefully our paths will cross again.

Moreover, I would like to express my thanks to Tommy Due-Løvaas for access to anomyzied datasets, which made this thesis possible.

Abstract

DDoS attacks have for the last two decades been among the greatest threats facing the internet infrastructure. Mitigating DDoS attacks is a particularly challenging task. It is known that ordinary signature based detection techniques are inefficient in undermining DDoS attacks as this type of attacks has the ability to mask itself among legitimate traffic. In this thesis, we present an paradigm for countering DDoS attacks at the targeted victim by using elements from data mining and machine learning. Two novel methods that focus on identifying hidden data structures in historical traffic are proposed, to differentiate legitimate traffic from abnormal traffic. In the first method, we resort to data mining techniques to find association rules which are able to describe the part of traffic that has higher likelihood of re-occurring. As a data structure for storing those data driven rules, we employ a binary tree structure. The second method builds on previously uncharted areas within mitigation techniques, where clustering techniques are used to create geographical clusters. In order to summarize the clustering information for real-life traffic filtering scenarios, we use the concept of bloom filters. The results show that these mitigation approaches improve the ability to separate between unknown abnormalities in the dataset and the legitimate traffic structure. Our proposed DDoS filtering schemes are able to mitigate 99% of the botnet traffic and thus countering significantly the magnitude of those attacks.

Contents

1 Introduction **3**
1.1 Problem statement 4

2 Background **6**
2.1 DDoS attack techniques 6
2.1.1 Network layer attacks 7
2.1.2 Transport layer attacks 8
2.1.3 Application layer attacks 9
2.2 DDoS trends 10
2.2.1 DDoS Attack Vector 11
2.2.2 Mega attacks 11
2.2.3 DDoS Attack Source Countries 13
2.2.4 Multi-vector attacks 14
2.3 DDOS attacks 14
2.3.1 Spamhaus.org 14
2.3.2 The Dyn attack 15
2.4 Cloud computing 15
2.5 Botnets 16
2.6 Anomaly detection techniques 19
2.6.1 Statistical analysis 20
2.6.2 Machine learning 20
2.7 Learning techniques 21
2.7.1 Supervised learning 21
2.7.2 Unsupervised learning 21
2.8 Machine learning algorithms 22
2.8.1 Classification approaches 22
2.8.2 Clustering approaches 24
2.9 Data mining 27
2.9.1 Apriori 27
2.10 Relevant research 27
2.10.1 History-based IP Filtering 27
2.10.2 Adaptive History IP filtering 28
2.10.3 Source IP addresses prediction using density Estimation . 29

3 Methods **30**
3.1 DDoS mitigation point . . . 30
3.2 Attack vector . . . 30
3.3 Mitigation technique . . . 31

4 Data mining **33**
4.1 Datasets . . . 33
4.1.1 Dataset 10 . . . 33
4.1.2 Dataset 11 . . . 35
4.2 Apriori-based frequent networks . . . 36
4.2.1 Dataset 10A . . . 37
4.2.2 Dataset 10B . . . 38
4.2.3 Dataset 10C . . . 38
4.2.4 Dataset 11A . . . 39
4.2.5 Dataset 11B . . . 40
4.3 Density-based geographical clustering . . . 43
4.3.1 Deciding core-point threshold . . . 44
4.3.2 Deciding maximum distance . . . 46
4.3.3 Optimal result . . . 48
4.4 Reduced-density geographical clustering . . . 54
4.4.1 Deciding minimum required points . . . 57
4.4.2 Deciding minimum length . . . 59
4.4.3 Optimal result . . . 61
4.5 Classifying new objects . . . 67
4.5.1 Naive Bayes . . . 67
4.5.2 Reverse search . . . 70

5 Approach **72**
5.1 Testbed environment . . . 72
5.2 External tools . . . 73
5.2.1 Repache . . . 73
5.2.2 BoNeSi . . . 74
5.2.3 Python3 . . . 74
5.2.4 Nfqueue . . . 74
5.2.5 Pybloom . . . 74
5.3 Data structures . . . 75
5.3.1 Tree structure . . . 75
5.3.2 Bloom filter . . . 76

6 Results **78**
6.1 Load test . . . 78
6.1.1 Load test on Apache web-server . . . 78
6.1.2 Load test on proposed structure . . . 80
6.2 Botnets . . . 86
6.2.1 B1 . . . 86
6.2.2 B2 . . . 89

6.2.3 B3 . . . 92
6.2.4 B4 . . . 94
6.3 Reverse search . . . 97
6.4 Real life simulation . . . 98
6.4.1 Real life simulation 1 . . . 98
6.4.2 Real life simulation 2 . . . 101
6.4.3 Real life simulation 3 . . . 103

7 Discussion **105**
7.1 Algorithms . . . 105
7.2 Attack vector . . . 106
7.3 Datasets . . . 107
7.4 Data structure . . . 107
7.5 Mitigation point . . . 107
7.6 Problem statement . . . 108
7.7 Future work . . . 109
7.7.1 Populating density-based clustering . . . 109
7.7.2 Different attributes . . . 111
7.7.3 Employing time . . . 111

8 Conclusion **113**

9 Appendix **115**
9.1 Algorithms . . . 115
9.1.1 AFN . . . 115
9.1.2 DGC . . . 116
9.1.3 RDGC . . . 117
9.2 Mitigation approach . . . 120
9.2.1 Bloom filter . . . 120
9.2.2 Binary tree structure . . . 121

Chapter 1

Introduction

A denial of service(DoS) attack can be described as an explicit attempt to render a server or network incapable of providing normal service to its users. Although, it is possible to exploit different software vulnerabilities to deny legitimate users access to services, DoS attacks often relay on continuously and excessively consuming a limited resource such as bandwidth, memory, storage, or CPU, which is necessary for the targeted service to operate correctly[52][75]. We can differentiate between denial of service(DoS) attacks and distributed denial of service(DDoS) attacks; DoS is where one attacker, with one network connection, execute an attack. While, a DDoS attack adds a many-to-one dimension to the DoS problem. Instead of using one connection, a DDoS attack often uses thousands of compromised hosts to execute an attack, amplifying both the available attack resources and the complexity of a DoS attack[27].

There is no doubt that DDoS attacks have over the last decade become an immense threat to the Internet infrastructure. Attacks have become commonplace with a wide range of global victims in everything from commercial website, educational institutions, public chat servers and government organizations[57]. DDoS attacks have grown in complexity over the last couple of years and criminal launch constantly more complicated attack patterns, containing a multitude of vectors, adapted to specific victims. There is a couple of factors to consider when trying to understand why DDoS is a rapidly growing problem and still a major problem for IT professionals, regardless of immense research and proposed solutions over the last decade. Firstly, the Internet architecture has originally been designed around openness without taking into account serious security issues. Attackers often successfully exploit this weakness by using vulnerable protocols and unpatched services into launching DDoS attacks. Mitigation approaches have often focused on designing new protocols to be network resistant against DDoS attacks. However, the different proposed solutions have proved insufficient since the approaches have not been implemented in a wide enough scale.

Internet of things(IoT) devices, refers to inter-networking devices as cameras

and baby monitors, which can communicate through network communication. IoT devices are still in its infancy and insecure devices have continued to be a problem over the last couple of years. Hackers are increasingly taking advantage of these vulnerabilities to add insecure IoT devices to already well functioning botnets. These pools of vulnerable devices have added to the capabilities of botnets which have lead to stronger attacks. In late 2016, high profile DDoS attacks, containing IoT devices, set new standards for larger and more sophisticated attacks. The largest known DDoS attack to date was reported in late 2016 and targeted Dyn's DNS servers. The DDoS attack used compromised IoT devices and consisted of an estimated attack size of 1.1 Tbps. The attack affected large parts of USA, knocking several popular websites offline[68]. Kaspersky Lab, which is a cybersecurity and anti-virus provider, reported similar attacks on ISPs in Ireland, United Kingdom and Liberia, all leveraging the use of IoT devices[40]. Akamai reported, in the fourth quarter of 2016, that 7 out of 12 attacks over 100 gbs, can be directly contributed to IoT botnets[10]. It is expected that IoT devices will continue to be fuel for more powerful attacks as devices contains severe and easily exploitable vulnerabilities.

Moreover, DDoS attacks constantly adapt to changes in mitigation techniques, by finding and exploiting new vulnerabilities in protocols and systems. Attackers no longer require extensive knowledge about the problem domain or resources to be able to execute attacks. DDoS-for-hire has emerged over the last couple of years, where attackers can rent cheap botnets. This gives attackers nearly unlimited supply of bandwidth and CPU to attack a single victim. These services gives attackers an easy and understandable GUI for as little as 25$ an hour. DDoS-for-hire has turned into a highly profitable service and the profitably can exceed over 95%. Mainly mid-sized websites are attacked today and attackers often extort victims to pay a ransom[69]. Hence, it is therefore not hard to understand why DDoS attacks are on the rise and continues to be the most popular attack mechanisms for attackers.

1.1 Problem statement

This thesis argues that the best way to protect against potential DDoS attacks is to use data mining and machine learning to find relevant and significant patterns on traffic history. Based on found traffic correlation, filtering mechanisms can be dynamically applied to prevent abnormal activity from having access to a service. This leads to the first problem statement that this thesis will explore:

How can we use data mining to find pattern correlation in data history to build efficient filtering rules that are able to mitigate DDoS attacks?

Even though DDoS attacks have steadily increased over the last couple of years, the average peak attack bandwidth and volume have continued to drop[7][9][4]. In 2015, the average duration of an attack under 10 gbps were around 39 minutes. This is long enough for a possible attacker to be able to im-

pact the infrastructure or application. However, it's not necessarily long enough to have a huge impact on the availability of the systems[26]. There is however also a rise in the amounts of DDoS attacks that are over 100 gbps. It's therefore important to explore how well the solution is able to deal with DDoS attacks of different size. This leads to the second and last problem statement this thesis will explore:

To which extent is our solution resilient to DDoS attacks of varying magnitude?

Chapter 2

Background

To develop a comprehensive and efficient defense mechanism against DDoS attacks it is necessary to have extensive understanding of the problem domain. The diversity and quantity of information available regarding the DDoS problem is overwhelming. However, it's crucial to have a better understanding of both the problem domain and the current solution space to create efficient and good defense approaches. This chapter will explore the scope and characteristics of DDoS attacks as well as the components in use. Moreover, the chapter will highlight relevant defense mechanisms that can be used to address DDoS attacks.

2.1 DDoS attack techniques

Over the last couple of years, DDoS attacks have focused on attacking the infrastructure and application layer, either by overloading the bandwidth capacity or focusing on depleting some limited network resource. Attackers often successfully exploit the open internet infrastructure into launching DoS or DDoS attacks. Leiwo and Nikander notes that new protocols should be designed to be more network resistant to DDoS attacks as any statefull protocol is vulnerable to attacks like SYN and PUSH ACK flooding[47]. The common factor for DDoS attacks revolve around pushing heavy amounts of traffic into the targeted system or network, therefore exhausting some specific resource. We can classify DDoS into different categories based on which layer the attack targets, consequently network layer, transport layer and application layer attacks. This section will go through some currently popular and known attack techniques inside of each category.

2.1.1 Network layer attacks

Network layer attacks, are DDoS attacks that targets the network layer by attempting to overwhelm the bandwidth and routing infrastructure. The attacker will often try to accomplish this goal by amplifying the original attack to a much larger attack, commonly known as an amplification attack. An amplification attack is a reflection based DDoS attack, where through various set of techniques, the attacker turns a small request or payload into a much larger payload directed at the victim, therefore successfully overloading the victims bandwidth[72]. Amplification attacks have been heavily favoured by attackers and is excepted to be one of the preferred attacks as long as misconfigured internet devices and services are available. An amplification attack can be achieved through a various set of means, among these are a DNS amplification, the misuse of a broadcast IP address and NTP reflection.

DNS reflection attack

The basic technique in DNS reflection attack is to request a large zone file with the source IP address spoofed to be the intended victim. The attackers request is only a fraction of what the DNS server will respond with, efficiently amplifying the attack to many times the size of the available bandwidth[80]. While a DNS query consist of approximately 36 bytes, a response message could easily triple that size. At worst, depending on the request, the response can come up to a couple thousands bytes. Potentially, the attacker could consume the entire bandwidth of the victim by only generating a few thousand responses[36].

These situations are often difficult to protect against since it's the DNS server that performs the direct attack. In an ordinary DDoS attack, it's a possibility to block the different bots that perform the attack. However, blocking a DNS server might damage the operation of a corporate network. As in a generally case, DNS is needed by any service, like HTTP and FTP, which requires name resolution[36]. Furthermore, if the attack successfully manages to exhaust the victims bandwidth, the software solution in place doesn't matter as the network link is completely saturated.

Broadcast reflection attack

The ICMP protocol is a diagnostic tool that can be used to test the reachability of different computer systems. A host can send an ICMP echo request message to a computer system. When the receiving system gets this message, the system will respond by sending an ICMP echo reply message back to the sender[44]. Broadcast reflection attacks or smurf-based attacks exploit this operation by sending spoofed ICMP echo requests to a router that is configured to relay ICMP messages to all devices behind the router. When all of the devices receive the ICMP echo request, the devices will reply with an ICMP reply message to the victim host, efficiently amplifying the attack[80].

NTP reflection attack

Network Time Protocol(NTP) is a protocol used for time synchronization between hosts. An attacker can misuse the *monlist* command to execute a reflection attack. The *monlist* command, in older versions of NTP, lets a user request 600 of the last hosts who connected to this service. This means that a small request can be amplified to nearly 1000 times larger than the original amount and thereby clog up network resources and bandwidth capacity.

2.1.2 Transport layer attacks

In transport layer attacks, the attacker misuses the communication protocols to launch DDoS attacks. Misusing either the UDP protocol or the TCP protocol, through SYN flooding, is common. This sub-section will go through these two DDoS techniques.

TCP SYN flooding

In a normal TCP connection, a connection is initiated by sending a SYN message to the server. The server will respond with a SYN-ACK message before the client establishes the connection by responding with an ACK message. After this procedure, the client and server can exchange data. TCP SYN attacks exploit this three-way handshake by sending large amounts of SYN packages to the server. The receiver will respond with a SYN-ACK packets to establish the connection. However, the attacker will never respond to this message, therefore holding up processor and memory resources which could be used to serve legitimate users. The server will eventually run out of resources and therefore no longer be able to respond to legitimate requests.

There are different methods of performing SYN flood attacks[80]. An attacker can either perform a direct attack or take a spoof based approach. In a direct attack, the approach is to send as many SYN messages to the victim as possible without spoofing the source IP address. The attacker then prevents the operating system from responding to SYN-ACKs responses. The attacker can also take a spoof based approach, where the source IP address is spoofed. This attack relay on the assumptions that a portion of the chosen spoofed clients doesn't respond to the SYN-ACKs messages. Either because no client system exist at that particular IP address, or that some of the spoofed clients will not respond to illegitimate SYN-ACKs. Incorporating SYN flooding attacks by using multiple agents makes the attacks nearly impossible to mitigate [80].

UDP flooding

In an UDP flooding attack, huge amounts of UDP packets are sent to the victim. UDP flooding attacks often relay on sending packets to random ports at the targeted victim. This forces the host to look for applications on these ports. If

the victim isn't running any service on the chosen port, an ICMP unreachable message is sent back to the source. The victimized system is forced to send huge amount of ICMP packets back to the sender. This will both use a large amount of system resources on the victimized system, as well as saturating the network link. Eventually, the server will succumb and legitimate request will no longer have access to the service. Since UDP is a stateless protocol, it is easy for any attacker to disguise themselves by spoofing their own address. Most operating system mitigate UDP flooding by limiting the amount of ICMP responses. Furthermore, some UDP flooding mitigation techniques focus on rate-limiting the amount of accepted UDP datagrams to a certain destination over a certain time period[80].

2.1.3 Application layer attacks

Application layer attacks have over the last couple of years grown in both complexity and prevalence. The attacks differs from other types of DDoS attacks, in the sense that the attacks focuses on sending high rates of seemingly legitimate request. These type of attacks often consume less bandwidth and are generally more difficult to protect against as the traffic doesn't differ from legitimate requests[80].

HTTP flooding attack

The most common application layer attack is HTTP flooding. HTTP flooding attacks have become more and more common over the last couple of years[53]. NSFOCUS, which provides security solution, reported HTTP flooding as the second highest attack type, in 2013, with 36.2% of all monitored attacks[61]. HTTP flood is a sophisticated attack as it doesn't use malformed packets, spoofing or reflection techniques to execute an attack. Instead an attacker exploits the seemingly legitimate HTTP GET and HTTP POST requests to attack a victim. A HTTP flood attack targets web-services, and will often try to charge an application with heavy HTTP GET or HTTP POST requests. The attacker often change and use unique web request. Therefore, request bypass the caching system and force the system to render and respond to every single request. Unique request can either be accomplished by using legitimate URL or by randomizing the used URL request.

The attacker tries to make a service allocate as many resources as possible. As the given HTTP service need to respond to every single request. It is possible to deplete the system resources with few request. The attack rate can therefore remain fairly low and infrastructure mitigation techniques such as rate-based detection engines are usually not successful in detecting and mitigating HTTP flooding attacks. HTTP flooding attacks often require an understanding of the targeted service, as each attack needs to be specifically crafted. This makes HTTP flooding attacks extremely difficult to protect against, as the use of standard URL request makes the attacker behave like a normal user pro-

file. Standard signatures to mitigate HTTP flooding are often not possible and commercial solutions out there as SNORT, MINDS and SPADE are not able to mitigate HTTP flooding[86][28][70].

As normal IDS techniques focus on signature based detection are not able to detect HTTP flooding attacks, current mitigation techniques often relay on traffic profiling and identifying abnormal traffic pattern behaviour. These mitigation techniques have a high focus on looking at normal users browser behaviour before identifying abnormal browser patterns[53][82][85][83].

Yatagai et al. proposed two solutions to mitigate DDoS attacks; firstly, the correlation between the time a host use on a certain web-page, with the information size on that web-page can be monitored. The general consensus is that the browsing time on a web-page is proportionate to the amount of increased information. Compromised hosts can therefore be removed as it is expected they don't follow the normal browsing pattern behaviour. Secondly, the browsing behaviour of every host can be monitored. compromised hosts are expected to have a continues stream of the same browsing behaviour. It is therefore possible to distinguish a compromised host from a normal host[85].

Xie and Yu proposed looking at the structure of how web pages are accessed. Here it is assumed that normal users access pages sequentially, based on hyperlinks, while bots don't follow this pattern. It is therefore possible to remove users which don't follow a certain user behaviour[81]. Beitollahi and Deconinck proposed to measure statistics and behaviour under normal traffic flow to mitigate DDoS attacks. These statistics include attributes as request rate, download rate, uptime, hyperlink depth and page popularity. Based on a normal traffic pattern. A score for any new connection can be determined and connections with lower scores can be dropped. [15]

The methods on looking at the browsing behaviour is quite simple and can often be avoided by changing the attack pattern. Moreover, many of these HTTP mitigation methods focus heavily on the idea that there is enough time to investigate the data pattern to a high degree. Proposed mitigation techniques within HTTP flooding attacks often don't consider the same time constraints as other DDoS mitigation techniques. With the rise of more common and stronger attacks, this is not necessarily an ideal scenario, as stronger attacks with HTTP flooding are becoming more common.

2.2 DDoS trends

DDoS attacks have raised in popularity over the last couple of years and attacks have become more complex and sophisticated. The profile of a typical DDoS attacks seems to change each year between different attack methods, attack volume and attack patterns. Moreover, as new vulnerabilities within protocols or system are found, new attack methods and techniques are used to initiate

and deplete some vital and finite resource. This section will look at some of the attack trends over the last couple of years.

2.2.1 DDoS Attack Vector

Today, infrastructure layer attacks accounts for most of the DDoS activity. This trend have remained for the last years, as attackers heavily rely on reflection based techniques as the primary DDoS attack method. Reflection based techniques does not only hide the true identity of the attacker, but also require fewer attack resources to overload the victim[6]. NTP amplification and DNS amplification are reported by Akamai and Versign as one of the top DDoS attack vectors in 2016[10][77]. Akamai reported that DNS and NTP reflection accounted for 34% of all attacks reported in the fourth quarter of 2016[10].

Although NTP amplification attacks have been decreasing over the last couple of years as servers are being patched or taken out of service, DNS amplification attacks have remained the top attack vector and account for 20% of all reported attacks in the fourth quarter of 2016[10]. Reflection attacks, like DNS, are heavily favored as it allows attackers to hide their identity and amplify their own traffic. UDP services like DNS, NTP and SSDP are vulnerable to these reflection attacks as the UDP protocol don't require users to authenticate before requesting data. Verisign reported that attacks employing the UDP protocol accounted for 52% in the last quarter of 2016[77].

However, although reflection based techniques remains quite popular. Kaspersky lab, reported amplification layer attacks as diminishing over the last couple of years. The problem domain around amplification attacks is well known and downward trend has shown less devices that can be used to amplify attacks. Instead, application layer attacks have had increasingly popularity and have taken over some of the attack marked previously occupied by amplification attacks[40][39]. Akamai reported Application layer attacks accounting for 10% of all DDoS attacks in 2015[6]. In the end, DDoS attacks contain a multitude of different attack vectors and hackers constantly probe the networks to find new vulnerabilities to exploit. New vectors are therefore constantly emerging and dis-emerging as vulnerabilities are being discovered and patched.

2.2.2 Mega attacks

Mainly mid-sized websites are today attacked by criminals. This is often with the intentions of extorting money from victims. An average DDoS attack still remain quite small and consist of around 900mbs to 1gbs. Moreover, around half of reported attacks are under 500mbs[20][31]. In 2015, Amazon reported the average duration of attacks under 10 Gbps to be 39 minutes. This is long enough for an possible attacker to impact the infrastructure or application. However, not necessarily long enough to have a huge impact on the availability of the systems[26]. Furthermore, Verisign reports that an average attack peak

size increased 63% between 2015 and 2016. In 2016, an average attack peaks size consisted of 11.2 Gbps, where more than half of all attacks reached over 5 Gbps[77].

Larger attacks are still rare events. However, DDoS attacks are on the rise and huge attacks containing several hundred gbs are becoming increasingly common. The last couple of years have seen an increase in the amount of reported mega attacks. Mega attacks can be defined as attacks which contains over 100 gbps. Between the fourth quarter of 2015 and the fourth quarter of 2016, Akamai reported a 140% increase in attacks greater than 100 Gbps[10]. Mega attacks have continued to increased in size over the last couple of years. Akamai reported in the third quarter of 2016, a mega attack consisting of 623 gbps[8]. This was the largest seen mega attack to this date, and builds on a consisting trend of attacks with a increasing amount of seen traffic. *Figure 2.1* shows the largest reported attacks, each quarter, from Akamai over the last two years.

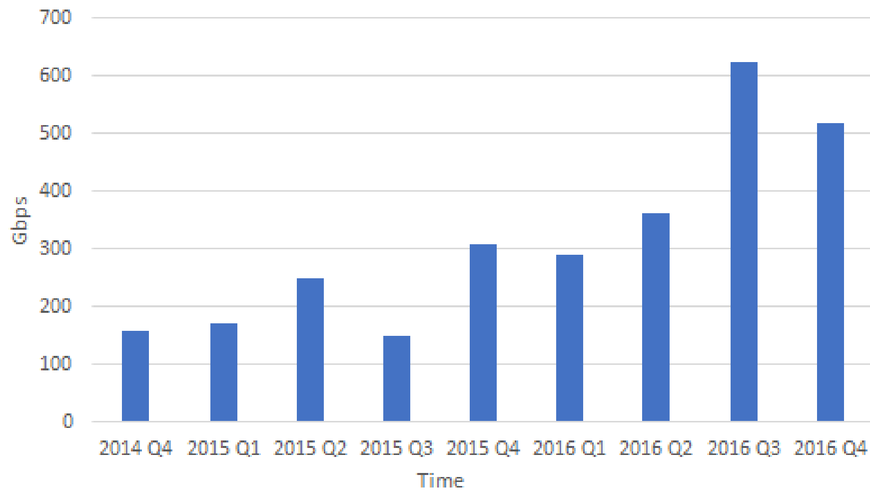

Figure 2.1: *Shows the largest mitigated DDoS attack from Akamai over the last couple of years. The largest reported attacks is calculated for each quarter.*

This trend, with larger and more heavy attacks, can be contributed to several reasons; Firstly, recent defects and vulnerabilities in Internet of things(IoT) have involved hackers to take control over millions of devices, ranging in everything from security cameras to routers, and embedding these devices into botnets. Secondly, stresser botnets, which can be used to stress test web-sites have increasingly been used for malicious intents. The use of IoT botnets are expected to become increasingly common, as long as IoT devices contains server security vulnerabilities [10]. Because of this, mega attacks are expected to continue to

increase over the next years.

2.2.3 DDoS Attack Source Countries

Since many of the DDoS attacks relay on spoofing, it is practical in-feasible to determine where all DDoS attacks originates from. However, we can look at non-spoofed IP addresses to get a certain concept of the problem domain. The available data and attack countries varies depending on the given data source. However, some countries resonates in much of the acquired statistics. Akamai has over the last three years largely reported China and USA in the top lead in the non-spoofed attacking addresses[6][5][7][9][4]. Combined, China and United states, accounted for more than 40% of the attacking traffic from the first Quarter of 2015 to the first quarter of 2016, as seen in *Figure 2.2*.

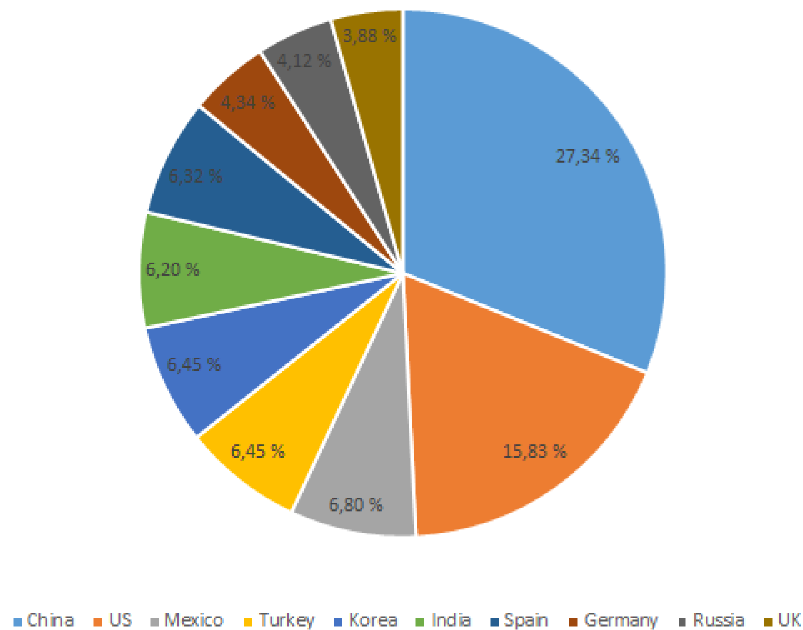

Figure 2.2: *This figure shows top 10 source countries for non-spoofed attacking IP addresses in the first quarter of 2015 to the first quarter of 2016. The statistics is gathered from Akamai's first quarterly report for 2016*

It is important to acknowledge that DDoS trends continuously change and adapt to the environment. Moreover, the acquired data is mainly based on application layer attacks that requires connection establishment. Attacks that

targets the infrastructure layer where the authentication of the users cant be established, is not included in the metric.

2.2.4 Multi-vector attacks

Multi-vectored attacks combine multiple attack techniques to launch a DDoS attack. These attacks do not only focus on attacking a single layer, but can attack both the application layer, transport layer, as well as the network layer simultaneously. A multi-vectored DDoS attack can for example contain a combination of SYN flood and GET flood. These attacks cause extensive problems as each vector contains unique attack characteristic and requires a different mitigation technique. In the first quarter of 2016 Akamai reported that 59% of all mitigated DDoS attacks were multi-vectored[4]. This is an increase up from fourth quarter of 2015, where 56% of all mitigated DDoS attacks where multi-vectored. Furthermore, Verisign reports that 86% of all mitigated DDoS attacks use several vectors in the fourth quarter of 2016[77]. This builds on a trend seen over the last years, where attackers increasingly use more sophisticated measurements, such as several vectors, to bring down a server[9][4].

Today, the majority of DDoS attacks use either one or two vectors. However, an increasing number of attacks also employ three, four or even five vectors at the same time. These multi-vector attacks are often very efficient in bringing down a server, as the attack can deliver a large number of different requests and simultaneous connections to a web server. This can efficiently consume different resources and crash the server at the weakest link. Existing DDoS solution is challenged by the complexity and volume of multi-vectored attacks, as multi-vector attacks often run in a coordinated simultaneously way that constantly adapt to the environment, making them extremely difficult to protect against. It is expected that multi-vectored attacks will increase in the following years and there is increasingly a growing need for fast adaptable solutions that manage to mitigate highly adaptable attacks[62][4][77].

2.3 DDOS attacks

Since DDoS attacks is such a large problem domain, attacks vary widely by targeting different resources and employing different attack vectors. This section will go through some commonly known and large attacks which has been seen over the last couple of years.

2.3.1 Spamhaus.org

In March 2013 the website Spamhaus.org, which provides anti-spam filtering services online, experienced one of the largest known DDoS attacks to date. The attack initially started at 10 gbps, fully saturating Spamhaus network link

and efficiently knocking the web-site offline. At the highest point Spamhaus experienced a estimated attack size of 100 gbps. These large attacks are known as layer 3 attacks where the goal is to exhaust the victims bandwidth. Simply put, if the victim experience an attack consisting of 15 gbps, while the router is only able to handle 10 gbps, the software solution in place doesn't matter as the network link is completely saturated[65][80].

The largest source of the attacks came from DNS reflection, while a small sample came from ACK reflection attack. In the Spamhaus case, the attacker was sending a request for the DNS zone file ripe.net. The open DNS resolver response consisted of approximately 3000 bytes, which translates to a 100x amplification factor. The DDoS attack was mitigated by heavily employing the use of anycast. Anycast means that the same IP address is announced from several data centers. This makes the network work as a load balancer by load balancing request and ensure that each request is routed to the nearest data center in the network. Under an attack anycast helps in mitigating or thinning out the attack strength to several data centers. Instead of a DDoS being a many-to-one problem, it becomes many-to-many. This can help preventing a bottleneck from happening and the network link from being completely saturated. Once the attack was diluted, the attack was stopped at each firewall by using traffic profiling and blocking abnormal traffic[65].

2.3.2 The Dyn attack

In October 2016 two DDoS attacks took place against Dyn. Dyn is a major domain name system(DNS) provider which provides end-users the possibility of mapping a domain name to its IP address. The DDoS attack was accomplished by sending large numbers of seemingly legitimate DNS request and the attack came from an estimate of 100 000 infected devices. The botnet who orchestrate the attack, contained mostly IoT devices and the attack has been the largest DDoS attack on record with an estimate size of 1,2 Tbps. The attack affected large parts of the traffic on the USA east and west coast. During parts of the attack. Users where unable to reach some of the customer sites that DYN provided DNS mapping for. DYN reacted to the attack by using anycast to dilute the traffic before using traffic profiling and shaping to mitigate and remove the attack traffic.

2.4 Cloud computing

The national institute of standards and technology in United States(NIST) define cloud computing as a model for enabling convenient and on-demand network access to a shared pool of configurable computing resources[54]. These resources can range in everything from services and applications to networks and storage resources. In simpler terms, cloud computing refers to the applications that are delivered as services over the Internet, and the hardware and system software in

data centers which provide these services[12]. This translates into on-demand services where a consumer can take unlimited use of computing resources and only pay for the resources they consume.

As application demands varies over time this normally cause a serious issue when planning the infrastructure. For example, if the infrastructure have an average need for 100 servers throughout the day, but at peak hours need 500 servers, the company would often plan for the highest peak window to provide good and reliable service to the users. The company would therefore decide to provision for 500 servers. In normal computing this would cause the issue that substantially amounts of resources are idle throughput the day, causing immense waste for the company. If however, the company decide to underprovision, by provisioning for 100 servers throughout the day, this can be equally serious, if not more fatal than overprovisioning. Where the cost of overprovisioning is at least measurable. The cost of rejecting users not only cause loss in revenue, but users might also not come back.

Therefore, it is not hard to understand why cloud computing is on the rise[24]. The rise of cloud computing offers new aspects which were not available before. Cloud computing allow start-up companies and enterprises the ability to use nearly unlimited resources on demand, therefore eliminating the need to plan in advance. Companies can now start small and increase the use of resources when there is also an increase in needs. For example, a new web-site often won't need to use a considerable amount of resources in the beginning. However, the web-site needs to have the ability to support a spike in the demand if the services become popular, followed by a lowering in demand when some users leave. Cloud computing comes with this ability to pay for use and allows a flexible solution for consumer, where they only need to pay for the direct resources they consume[12].

Cloud computing is an ideal component to prevent or limit the damage from DDoS or DoS attacks, since cloud computing give the ability to scale up or down depending on demand. Cloud computing can be used to scale considerable up when the traffic load start to increase substantially. This can help in limit or at best prevent a DoS or DDoS attack from succeeding.

2.5 Botnets

The term botnets are used to define networks of infected hosts called bots. Botnets, can at any given moment, be composed of a few hundred to several thousands infected hosts. According to Rajab et al. botnets represent a huge part of unwanted contribution to the network traffic, as up to 27% of unwanted or malicious connection attempts comes from botnets[1]. Botnets can be seen as one of the larger cyberthreats for the IT community today. With the rise of DDoS-for-hire- models, There is no doubt that the threshold for criminals to acquire necessary resources and skills to attack their victims have been low-

ered[17].

Botnets will in theory give criminals nearly an unlimited set of resources, as a botnet can control millions of computer processors, countless gigabytes of storage and memory, as well as having access to enough bandwidth to overwhelm even the largest cooperation's[23]. This gives criminals an efficient platform to perform illegal activities that range in everything from identity theft, click-fraud, espionage, spamming or phising to DDoS attacks against a service [3][1].

When dealing with DDoS attacks, there are often thousands, to hundreds of thousands of compromised hosts, that attack a single server at once. Since bots and botnets are the main reason for why DDoS attacks succeed, it is important to know certain characteristics which botnets carry, for us to efficiently being able to protect against potential attacks. Can we for example assume, when a single botnet attack a web server, that the IP addresses the bots use are for the most part in the same geographical region?

To understand the answer to this questions, we need to know how botnets operate and spread over time. A botnets ability to cause damage to its environments through malicious actions, will often depend on the botnets ability to propagate to new nodes as well as retaining access of already acquired nodes[3][1]. The most critical and necessary phase for a botnet is therefore during the distribution phase[17]. Botnets will try to infect new hosts by exploiting a known software vulnerability, exploiting a weak security policy or by using social engineering to trick users into downloading and executing malware[60].

Hachem et al. notes that botnets increasingly use several techniques to propagates further on to new nodes[33]. If a botnet only exploited a single vulnerability and this vulnerability was fixed once and for all, the botnet will be unable to propagate to new nodes, and the expansion will come to a halt[3]. If nodes that have been infected also manages to recover from the infection, the bot population will slowly start to decline. However, if the botnet can mutate and exploit different vulnerabilities, the botnet can continue to retake new nodes, as well as re-infect old nodes that have recovered from previous infections.

Botnets will therefore employ several methods to continuously infect new nodes. Botnets borrow strategies from several classes of malware. This can include malware such as self-replicating worms and e-mail viruses. Infected hosts can be programmed to randomly scan a given IP block, to scanning the local subnet for known vulnerabilities. Some botnets propagates via email worms and web server worms. Social engineering can for example be used to trick users to go to a malicious website, where the website then searches for vulnerabilities in the users machine. If a vulnerability is found, this is exploited to gain access to the host. Social engineering can also be used to trick victims into downloading certain malicious files[60][33].

Dagon et al. concludes that victims of a botnet often are spread quite geographically apart. However, that victims may be concentrated in particular regions[25]. This depends on the propagation mechanisms the botnet use.

However, since some attacks may use a particular language as part of social engineering scheme, or a vulnerability that only exist in some parts of the world. It is reasonable to assume that these factors might help the botnet in having a higher concentration of bots in some geographical regions. Rajab et al. was able to capture this structural features of a botnet seen in *Figure 2.3*[1].

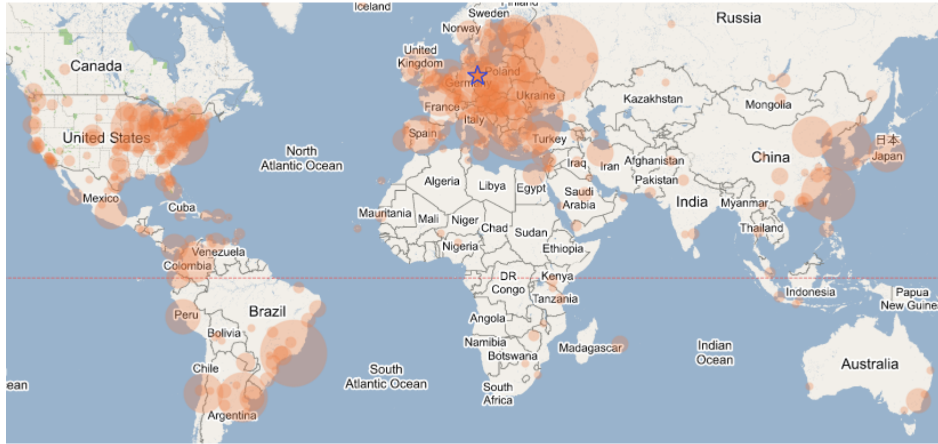

Figure 2.3: *Shows the DNS cache hits for one of the tracked botnets by Rajab et al. The star indicates the location of the IRC server for the botnet.*

Different Time-zones and locations play also an unexpected role into how well different botnets manages to propagate to new nodes[25]. When an end-host is turned off, it is not possible for other bots to exploit different vulnerabilities at the host and therefore gain access. Similarly, when a bot is turned off, it is not possible for the bot to infect new nodes. Therefore, we could partly assume that if a botnet have a high concentration of hosts in time zone x. The probability of the botnet having a high concentration of hosts in time zone $x + 1$, is higher than the probability of the botnet having a high concentration in time zone $x + 10$.

Overall, findings seem to confirm that botnets in general propagate in a diversity of ways[1]. It is important to acknowledge that each botnet is different. However, in general, we can assume that botnets will use different technique to spread and this will result in a spread demographic where bots reside. We could also assume that some part of a botnet will form several hot-spots of infected hosts. By being aware of botnets propagation characteristics, we can easier develop solutions and preventive measures that manages to minimize or stop the damage caused by DDoS attacks.

2.6 Anomaly detection techniques

In DDoS attacks, illegitimate traffic tries to mask itself among the legitimate traffic to deplete some vital system resource. This makes it substantially harder to filter out the illegitimate traffic without substantially affecting the legitimate traffic. Substantially amount of research have been done in analyzing traffic data to overcome this problem. Kim et al. conducted extensive analysis from real life traffic data, where the packet attributes were analyzed. The traffic distribution rate changed over a period of 18 hours, however, the rate of the monitored attributes values often just changed within a few percentage points of the original value[42].

Other research conducted by Kim et al. included collecting data traffic from several different web sites over a 90 seconds period. The analysis showed that the same website had a nearly equal percentage use of the different attributes values over a series of 10 seconds windows. However, if data was compared between different website, there was a distinct difference between the data composition[41]. Liu et al. also conducted research in collecting data traffic from routers located on the U.S west and U.S east coast[51]. The traffic composition showed little variation over time. However, the composition collected from different locations where considerable unique in both size, transport protocols and application protocols use.

This leads to the conclusion noted by Kim et al, that there are some packet characteristics which are different in an attack period versus a normal traffic period for any given site[42]. Based on this premises, the different characteristics in the normal traffic can be calculated and compared against attack traffic. This will enable the system to take intelligent choices in deciding which packets to either accept or drop. There are two common methods in detecting system intrusion; anomaly-based detection and signature-based detection. Signature-based detection techniques is able to detect attacks based on signatures of known attacks. This detection mechanism has a very low false positive rate. However,it also has the disability that it can't detect new types of attacks[63][13][48].

In comparison, anomaly-based detection techniques builds on the principle that traffic distribution of a service will change under an attack. Anomaly-based detection first analyses traffic based on a traffic model the system defines as normal. The system will go through a training phase to define the normal behaviour before the system enters the detection phase. Since anomaly intrusion detection compares traffic against the normally defined behaviour, the system has the ability to detect unknown attacks. However, anomaly based detection is highly dependant on training on a set of normal data. If the training data contains malicious activities the system might be unable to detect these activities when it enters the detection phase.

There exist several approaches inside of data collection, processing and filtering within anomaly detection techniques. Whereas, statistical analysis and machine learning are two of the more common approaches[58].

2.6.1 Statistical analysis

. Both statistical analysis and machine learning are concerned with the same question; how to learn from data. The biggest differences between statistical analysis and machine learning, is that they emphasis different points. Statistical analysis look at statistical inference and is a field within mathematics. Statistical analysis deals with finding relationships between variables to predict a certain outcome. The goal of statistical analysis is to identify trends within a data pattern which then can be used to make new predictions[58]

Both Entropy and Chi-Square techniques can be seen as statistical analysis methods within anomaly detection techniques[58]. Entropy can be computed on a sample of packets before comparing the entropy value to a new sample of packets. As noted by Feinstein et al. entropy levels between normal packet samples changes only narrowly. However, under an attack, these entropy levels changes to a detectable degree. This means that it is possible to detect anomalies in the traffic pattern[30].

Detection mechanisms within statistical analysis needs, to larger degree, to understand the underlying data structure. This includes statistical properties and the underlying distribution of data that is investigated. Statistical analysis relay on the analyzer to correctly identify parameters that will provide the correct data output. Because of this, statistical analysis are often concerned with low dimensional problems.

2.6.2 Machine learning

Machine learning is another approach to detecting anomalies and changes in network traffic and is about learning and making future predictions based on earlier observed data. Machine learning approaches builds on the foundation that a system can learn from data without being explicitly programmed. To accomplish this, machine learning approaches goes through a training phase before making decision on new data. Unlike statistical analysis, where a fixed filter decide what is normal and abnormal. Machine learning are able to update its filtering criteria based on new traffic[58].

While statistical analysis requires knowledge about the underlying data pattern, machine learning approaches require no prior assumption or knowledge of relationships between different variables. Machine learning is therefore often applied to high dimensional datasets. Since both machine learning and statistical analysis are concerned with learning from data and making future predictions, there are today little differences between the two approaches. Because of this, machine learning and statistical analysis has merged over the last couple of years and the use of either approaches have become interchangeable.

2.7 Learning techniques

Inside anomaly detection techniques, there exist different techniques which are used to learn the normal traffic distribution, this is mainly unsupervised learning and supervised learning. This section will go through the different learning techniques that is used as well as the weaknesses and strengths of the different approaches.

2.7.1 Supervised learning

Supervised learning is the most common branch of anomaly detection techniques, where the idea is to learn the normal pattern distribution by looking at labelled data. The system should then, based on knowledge of previous known data, be able to classify new and unknown data. This would mean that the training phase already should know that packet X is either a normal or an abnormal packet. To classify new data, the system can, based on the known behaviour, inspect different attributes and determine which category a packet fits best.

If we for example imagine that we want to classify fruits into different categories we first gather a large set of known fruits. All of the fruits are labelled with its category as for example an apple or an orange. During the training phase the machine is shown each fruit independently, a score is created for each possible category that the fruit can belong to. In the perfect scenario we would like the fruits to be given the highest score in the category it belongs to. However, as this is often not the case, a function will in the end measure the distance between the given scores and the desired pattern of scores. The machine will then modify its internal parameters, called weights, to reduce this error as close to the optimal pattern as possible[46].

As noted by Chapelle, labelled data is considerable more expensive and harder to acquire than unlabelled data[21]. It is a huge lack of labelled data in the industry. In intrusions like DDoS, it's even harder to find labelled data, as it is often just the intent of the users that differs an attacker from a legit user. A different possibility would then be to use an unsupervised learning approach.

2.7.2 Unsupervised learning

Unsupervised learning is approaches that learn the normal data distribution by looking at unlabelled data. Opposite of supervised learning, unsupervised learning allow us to look at the data with little to none known idea what the result should look like. This means that no error or reward signal would be given to evaluate a potential answer. As noted by Chapelle, It is normally assumed that the data distribution is independently and identically distributed from a common distribution point x[21]. Essentially meaning that it is assumed that

the data should follow a pattern or a structure. Based on this structure we should be able to classify new data and detect anomalies in the data pattern.

However, according to Srihari and Anitha there is often a large uncertainty associated with modeling data set based on unlabelled data[73]. It has been argued that the problem of unsupervised learning, is fundamentally that of estimating a density which is likely to have generated X[21]. In an ideal scenario we should be able to different between packets from legal users and packets from illegal users. However, in case of DDoS attacks this become considerable harder as only the intent differs a legal user from an attacker. When a DDoS attack is detected, it is therefore still a hard task to actually filter out the correct packets without affecting the legal users as Unlabeled data only makes this uncertainty greater.

This problem only escalates when considering the issue with required normalized data input. It's theoretical impossible to know for sure that the system is only given normal data as input[79]. If traces of the training traffic contains abnormalities, it might lead to the system classifying attack packets as legitimate, leading to a high false negative. Depending on the approach it is essential that additional work is done to eliminate possible abnormalities which may be present in traffic supplied to the learning phase. However, as discussed, this is often not a simple task, as there is either a lack of labelled data, or the unlabelled data is unreliable as we don't know if the traffic are contaminated.

2.8 Machine learning algorithms

A machine learning algorithm is the driving force behind deciding which packets to drop or accept. Algorithms can be classified into the broad categories; clustering or classification. This section will go through some relevant machine learning algorithms.

2.8.1 Classification approaches

An algorithm that implements classification is known as a classifier. A classifier can classify data into several predefined categories[43], however classification is often done into two categories; normal or abnormal. Classification will often run on a set of training data where the category is known, this can also be seen as supervised learning. This section will present some common and relevant classification techniques.

Naive Bayes

Bayesian Algorithms bases itself on the Bayes theorem. In a Bayesian model there is a hypothesis that a given data belongs to a particular class. A Bayesian model is not singular algorithm. But rather a term for all algorithms that bases themselves on a common principle that each considered attribute is independent

from all other attributes in the mix. The model gives answer to questions such as; if one or several independent events are observed, what is the probability for this object to belong to a certain class[63]. If we for example have a basket of fruits we can look at each fruit to determine what kind of fruit this is. If we see that a fruit is red, round and have a certain diameter, we could assume from prior knowledge of fruits that this fruit is an apple. A Bayesian algorithm will look at these attributes independently to determine this. This means that each attribute will contribute and increase the probability that the fruit is an apple. However, the correlation between the attributes is not considered. This means that the algorithm doesn't see the correlation between that the fruit is both round and red and therefore it must be an apple.

The Naive Bayes model is a heavily simplified Bayesian probability model[63]. The model is based on assigning a case or an event to the class that have largest posterior probability[56]. During the training phase the method stores a probabilistic summary for each class or category. This summary contains the conditional probability of each attribute value, as well as the base rate (prior probability) of the class. Each time the algorithm encounters a new instance, it updates the probabilistic stored with that specific class[45]. When the system now is given an unclassified object, the classifier use a function to check which of the classes the object most likely belongs to. The function or the Naive Bayes rule that is used can be seen under:

$$Probability = \frac{likelihood \cdot prior}{evidence} \tag{2.1}$$

Prior is here based on previous knowledge about that specific class. If we know that we saw 20 apples and 50 oranges in the testing phase, it is more likely that the new object is an orange. This gives the prior probability of $\frac{20}{70}$ for an apple, and the prior probability of $\frac{50}{70}$ for an orange. Moreover, the likelihood is the evidence that supports the notion that the object belongs to a certain class, while the evidence is all the evidence considered in total. The algorithm can further be written as:

$$P(c_i|A) = \frac{P(A|c_1) * P(C_1)}{P(A)} \tag{2.2}$$

Here C_i is a possible value in the session class and A is the total evidence on that attribute. Since all attribute values are independent of each other, the evidence A can be divided into pieces. Therefore dividing the function $P(A|c_1)$ into $P(a_1|c_i) \cdot P(a_2|c_i), ..., P(a_n|c_i)$[11]. This will result in the algorithm:

$$P(c_i|A) = \frac{P(a_1|c_i) \cdot P(a_2|c_i), ..., P(a_n|c_i) * P(C_1)}{P(A)} \tag{2.3}$$

If we have for example an unclassified apple that is round, green and with a diameter of 9 centimeters, we can insert the knowledge into the formula and ask ourselves what is the probability that an apple and an orange is round, green and with a diameter of 9 cm. The class that gets the highest probability is where the object will be placed.

Naive Bayes classifications are often robust to attributes that are irrelevant as the classification mechanism takes into account evidence from many attributes to make the final verdict. The downside of Naive Bayes, is that it requires strong independent assumptions of data[43]. Assuming that each feature is not correlated against any other feature in the set is not necessary correct and may have a negative influence on the end result. However, Naive Bayes is still very competitive against other machine learning algorithms[11].

2.8.2 Clustering approaches

In those cases where it is not possible to acquire sufficient knowledge about the underlying data to efficiently model new and unknown data, algorithms can instead use distance or similarity among data samples as a means for classifying new objects[38]. Clustering is one such method, where data is divided into groups or cluster of similar objects. The data pattern that forms a cluster should contain objects that are similar to each other, but dissimilar to objects in different clusters[2][84]. The goal of clustering, is to represent a finite set of unlabeled data into a finite set of natural hidden data structures[84]. Most clustering algorithms require some type of input, for example how many clusters or groups that you want. There are a various algorithms and techniques that can be used to perform clustering and in this section we will present some common known and relevant clustering techniques.

K-nearest neighbor clustering

The K-nearest neighbor (k-NN) classification approach is one of the more simple classifications methods within clustering. The algorithm makes use of similarities between objects as a means of classification new and unknown objects[64][38]. The decision on where to classify a new object depends on the K-closest neighbor classes to that object, where the value K is pre-defined. Given N training vectors, K-NN identifies the K nearest neighbors of any new vector. For example if K is 3 and a new object is in the close proximity of two objects of class Y and one objects of class X, the object will be classified into belonging to class Y. An example of K-nearest neighbor classification can be seen in *Figure 2.4*:

One of the possible problems that arise when using K-NN classification is that each sample vector or data point is given equal importance in deciding the

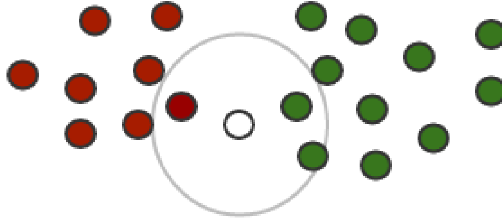

Figure 2.4: *Here two of the closest objects belong to the green class, while one object belong to the red class, therefore classifying the unknown object(white) in being in the green class.*

class of any new object. This problem often emerge when data points for each class overlap or are in close proximity of each other. If a data point in class Y, deviates from where the cluster points are normally situated. This data point should probably not be given the same strength in deciding new objects, as those data points that are truly representative for cluster Y[38]. This problem can arise when the class distribution is skewed. If a class is more dominate in the training vectors, there is a higher likability that this class will dominate in an algorithm that focuses on the majority of a data points in a class, which is close proximity to any new object.

A way to overcome this problem would be to give each data point a weight in how strong affiliation it has to each possible cluster. The cluster that has the strongest affiliation will be the cluster that the object is assigned to. When assigning a new data point to a cluster, this weight can be used to prevent unrepresentative data points to have as much say as a representative data point.

K-means clustering

K-means is one of the most widely used clustering techniques. The clustering approach aims to divide N observations into K clusters, where each observation belong to the cluster with the nearest mean[50]. The approach can be used with K-NN clustering, where K-means is used to obtain the clusters before K-NN is used to classify new and unknown data into existing clusters.

One of the more popular algorithms inside K-means clustering is to use a iterative solution to find the local minimal solution. This algorithm is often called the k-means algorithm or the Lloyd's algorithm[37]. There are several variants of this algorithm. However, Lloyd's algorithm is based on the observation that the optimal placement of a mean is at the centroid of the associated cluster[37]. The approach begins with choosing the different cluster centers or means. When the cluster centers have been chosen, the algorithm follows two

steps; The first step determines which data belongs to which cluster via nearest distance calculation from the different points to the different means. The position of the cluster centers is then recomputed and moved based on finding the nearest center from all points in a cluster. The K-means algorithm will follow these two steps until convergence is reached[34][50].

Even though this technique is able to find the local minimal solution, it's not necessary the global minimal solutions, as the Lloyd's algorithm does not specify the initial starting placement of the clustering centers[37]. This is a serious weakness as the iterative technique is sensitive to the initial starting positions of the cluster centers. In other terms, how well the clustering is, heavily depends on where the initial cluster centers are set[50][18].

There is currently no known efficient and widely accepted solutions to this problem. However, in order to obtain optimal clusters or solutions using the k-means algorithm, the algorithm is often ran several times with different starting positions for the cluster centers[50]. However, it is important to acknowledge that this is not an ideal solutions and several other techniques have been proposed. Among these solutions is an approach that tries to find a better starting condition so the algorithm can converge to a better local minimal. This can be done by trying to calculate different vector areas where the density is strongest, before setting these areas as the starting positions[18].

DBSCAN

DBSCAN is a density-clustering algorithm which builds clusters based on points that are closely linked together. The algorithm takes two parameters; *minpts* and distance d. If a point Y can reach *minpts* in a radius, based on distance d, point Y is considered a core-point. If point Y then have a path $p_1, p_2, ..., p_{n-1}, p_n$ to point N, where each point between point Y and point N is a core-point, all points p_x on the path needs to be density-reachable to p_{x+1}. Density-reachable means p_x need to have p_{x+1} within distance d. These core-points on the path $p_1, p_2, ..., p_{n-1}, p_n$, including the points that is density-reachable to a core-point in this path, is considered as a cluster. Points that are density-reachable to a core-point, but is not considered as a core-point because it doesn't contain *minpts* within radius d are considered outliers of a cluster[16][29].

DBSCAN is a popular density-based clustering algorithm which allows clusters to expand in any shape and form. Unlike K-means, which assume all points in a dataset are legitimate, DBSCAN is resistant against noise, as any point which can't satisfy the minpts criteria, and is not connected to a core-point, is considered as a outlier in the dataset. Since DBSCAN builds clusters based on points that are linked together, the algorithm doesn't need to define the amount of start-clusters. However, since the clustering algorithm needs to define *minpts* and distance d this can cause several weaknesses. Either, if the dataset are not understood correctly to choose meaningful parameters, or that it is impossible to cluster data with a high differences in densities[16][29].

2.9 Data mining

Data mining is a process of finding and discovering interesting patterns in large amounts of data. The goal is often to use different pattern techniques to extract interesting information and hidden structures from large datasets. Data mining techniques are similar to both statistical analysis and machine learning. The differences between the techniques have become blurred over the last couple of years. However, while machine learning focuses to create a model for predicting new data, data mining often focuses on finding and explaining pre-existing data patterns in a dataset.

2.9.1 Apriori

Apriori is an algorithm that uses data mining to find frequent itemsets and is a method for discovering interesting relationship between data variables. The apriori principle states that if an item or collection of items is frequent, then all of its subsets must also be considered frequent. The algorithm bases itself on identifying frequent itemsset k, before using the frequent itemset of k to find the frequent itemset of $k+1$. The itemsets is extended with one item as long as the itemset is considered frequent. When all k itemsets have been considered not frequent, itemset $k-1$ is returned and can be used to create association rules[67].

As an example, if we have a set of all possible items $I = \{i_1, i_2, i_3, ...i_m\}$. And a set of all transactions or incoming data $T = \{t_1, t_2, t_3, ...t_n\}$, where every transaction t_x is a set of one or more items from I. The algorithm will find all frequent items k in T, before extending all the found items with k+1. All frequent items of k+1 will then be used to find $k+2$. This means that if we have an itemset of 2 items, consecutively $\{i_1, i_2\}$. Then i_1 and i_2 must be considered frequent for $\{i_1, i_2\}$ to be considered frequent[67].

2.10 Relevant research

Several techniques can be used to combat DDoS attack. Everything from prioritizing different clients based on time that they have waited, to give different clients a reputation value that they can rely on. This section we will present some relevant research that can be used to help combat DDoS attacks. .

2.10.1 History-based IP Filtering

Peng et al suggested to use a database of previously seen legitimate IP addresses to counter DDoS attacks. This method bases itself on the common assumption that normal traffic differentiates itself from traffic under an attack. The idea is that the network should learn from previous network connections, before under

an attack, the learned behaviour is used to characterize incoming packets as normal or abnormal[74]. This mechanism known as History-based IP filtering(HIF) use an IP address database(IAD) containing previously seen IP addresses over a certain time period. Under an attack, only IP addresses from the IAD are allowed to access the network or service. Only a small number of source IP addresses, which is considered frequent is kept in the IAD. IP addresses are considered frequent based on two factors; The number of days the address have appeared in the network and the number of packets the IP address has sent to the network, as Peng et al. assumes frequent IP addresses is expected to send a certain number of packets to the network[74].

When updating the IAD, a sliding window is used to remove expired IP addresses based on timestamp which is saved in the IAD with the IP address. This ensures that only the most relevant IP addresses is kept in the database. When the network experience a DDoS attacks, the IAD is used to decide which packets to drop or accept. The HIF database was built using Auckland data traces and managed at most to acquire an accuracy of around 90%[74][35]. HIF have the advantages of being easily and efficiently commutable. However, this advantage also means that the approach can't differentiate between users who visit the server more often than other user, therefore giving less accurate results[32].

2.10.2 Adaptive History IP filtering

Goldstein et al suggested using an adaptive history-based IP filtering (AHIF) algorithm against DDOS attacks. The approach rely on observing data and using the Bayesian theory to derive optimal IP networks rules from this data, which then decides which packet to accept and drop based on the dynamically assigned threshold[31].

Pattern recognition is used to derive the normal traffic distribution based on previously observed data. Based on the observed data the proposed approach creates a binary tree with access control lists containing IP networks to accept. The IP addresses are stored without the last 8 bits as many users often change IP address over a 24 hour time, as well as spoofing the last 8 bits are still common today. In the binary tree each leaf represent an accept of a x bit network. If two leafs share the same father, the leafs are deleted and the father node becomes a new leaf with one lesser bit in the network mask. This algorithm results in the minimum number of possible rules[31].

The approach was tested by simulating a bot network compromised of 100 000 bots. Due to the lack of real bot network data, the source addresses of the bots are randomly distributed over the whole IP space, except for IP addresses such as private networks, multicast addresses. The setup was able to process about 100 000 firewall rules at the rate of 40 000 packets per second[31].

The approach is compared against the HIF algorithm proposed by Peng et

al that also relied on history based IP filtering. The proposed method is shown to be superior. A big reason for this is that the firewall rules are prepared before a DDoS attack takes place. It allows the system to react faster to detect attacks and it frees up more resources for the firewall to use in filtering the traffic. The algorithm also goes out of of the premises that the server can only handle a limited amount of request at any given time and therefore only drops the necessary packets to be able to prevent a server overload[31].

2.10.3 Source IP addresses prediction using density Estimation

Today, most DDoS mitigation techniques employ source IP addresses to estimate the normal traffic behaviour. However, gathered IP address distribution or data is often under-sampled due to small or limited amounts of observations, which creates a problem when creating dynamically approaches which drops traffic based on the given data observation. To overcome this problem Goldstein suggested taking advantage of IP neighborhood relations by using density estimation[32]. The idea is that IP addresses that are close or similar to each other, share similar characteristics. For example if an IP address 56.81.12.10 appears in the normal traffic distribution, we can also assume that 56.81.12.19 is a part of the normal traffic distribution and will appear later.

To evaluate the distances between the different objects, both $Xor+$ and euclidean distance is used, Xor is where the highest bit that reach 1 within two objects, is the distance between the two corresponding objects[32]. Euclidean distances is added to the distance calculation. To avoid that distances within the given network mask is always constant, regardless of the variation within the subnet, the approach used a modified k-means clustering algorithm to compute the clustering centers. Once the clustering centers where computed, an area was defined around the clusters, which would be the IP addresses the model expected to see in the future. A growing algorithm was used to grow the areas larger around the clusters that represented more data points. The method was evaluated using 90 days of real world dataset consisting of up to 1.3 million different source IP addresses, where the approach then tried to predict the users the next following 10 days[32].

Chapter 3

Methods

This chapter will discuss and determine the structures and approaches which will be used to mitigate a DDoS attack. Both the approaches taken and the different phases that will be used are discussed and determined.

3.1 DDoS mitigation point

In a general sense, it's preferable to mitigate DDoS attacks as close to the source as possible, in order to reduce as much of the collateral damage. However, this is difficult for a number of reasons; it's generally hard to identify attack traffic close to the source, and the Internet is highly build around a authorization-free nature. Which makes it hard to implement any solution to a high degree. It is therefore a strong need to mitigate DDoS attacks near the target victim, as this seems to be the only solution for the current Internet infrastructure[31]. Mitigation techniques can then filter out traffic at the server which provides the service, or closer to the access ISP.

3.2 Attack vector

DDoS attacks is a complex and vast problem domain which often relay on multiple vectors to bring down a service. These vectors include everything from attacking the network layer to attacking and misusing the application layer. Different attack techniques often require separate defense mechanisms. Some popular attack techniques such as SYN, UDP and ICMP have identifiable patterns which have lead to some of the attacks to be successfully mitigated[31].

This thesis will instead focus on one of the more upcoming and advanced attack methods, HTTP flooding attack. A HTTP flooding attack is an application layer denial of service attack which targets online web-services and websites. HTTP flooding attack is one of the more advanced methods available.

Since flooding attack focuses on sending HTTP request which follows a legitimate patterns it is extremely hard to mitigate as it is often just the intent of the connection that differs. These sophisticated threats can consume a high amount of resources from the web-server by using a small number of compromised hosts. A HTTP flooding attack will accomplish this by charging an application with heavy HTTP GET or HTTP POST requests. This means that an attacker can create a low traffic rate which appear legitimate.

3.3 Mitigation technique

Several different approaches can be taken to mitigate a HTTP flooding attack near the targeted system. Since HTTP flooding attack mimics real user's behaviour, it is difficult to use statistical approaches to identify a pattern and thereby mitigate the attack. Instead we can build on the foundation from last section, that a normal traffic profile of a system differs from a traffic profile during an attack. Based on this known premises, different characteristics in normal traffic can be calculated and compared against new unknown traffic. This can enable the system to make intelligent choices when deciding which packets to either accept or drop.

Our proposed algorithms then need to go through a training phase and a testing phase. Machine learning techniques and data mining can first be used to learn the underlying traffic profile in the training phase before the state of any new unknown packet can be identified. Different packet attributes can be used for defining the traffic profile in the training phase. These attributes include IP addresses, protocols, packet size, server port numbers, source and destination IP prefixes, Time to live, header lengths, TCP flag patterns, IP fragmentation and incorrect check sums. When deciding the attributes to use in the profiling, it is important to choose a set of attribute values that correctly represent the traffic distribution of the system. A normal traffic profile will usually consist of single and joint distributions of the packet attributes. As noted by Kim et al. deciding to use joint distributions will often give a better way to represent the uniqueness of the traffic distribution[41].

Since we would like to calculate the legitimacy of any packets quickly, it would be unfeasible to look at all attributes. Looking at the most relevant attributes would be more natural. Kim et al suggested using an iceberg style to decide the profiling. The iceberg style will only store the most frequently occurring attribute values along with their ratio in the traffic profile[41]. However, as we are dealing with DDoS attacks, we would prefer to minimize the computation power usage. As DDoS attacks rely on overflowing a server with traffic, we don't have a long time to decide the fate of any new packet. Therefore, we could base the categorization of normal traffic versus abnormal traffic on one of the most unique attributes; the source IP address. The source IP addresses are today an essential component in classifying normal traffic behaviour. This is primarily because IP addresses are a characteristic that is exceedingly unique

for different web servers and services[32].

Based on the knowledge from the traffic profiling, several different approaches can be used to determine the legitimacy of a new packet. This will vary between different algorithms and can be everything from defining score based on each incoming packet, known as conditional legitimate probability(CLP), to simply accepting networks which have been pre-defined to be legitimate in the training phase. In this thesis, we should preferably investigate both scenarios.

It is important to note that if the attacker manages to follow the attribute distribution of the normal traffic profile, it would essentially be impossible to differentiate between legitimate and illegitimate traffic. In practice, it is often difficult for an attacker to know the traffic profile of the system. However as noted by Li et al, it is fair to assume that an attacker might be able to learn sufficient information about the policy, if the attacker is able to know which packets are dropped and accepted[49]. However, HTTP flooding attacks requires that the attacker starts and finalizes a request. This will make it hard for an attacker to adapt to an accepted pattern, as attributes, such as source IP address can't be changed. It should be assumed that if we properly identify a unique pattern based on machine learning and data mining, a HTTP flooding attack can be mitigated..

Chapter 4

Data mining

This chapter will investigate different algorithms and hypothesis which can be used to define a normal traffic pattern. The chapter will first go through different datasets, before investigating different hypothesis and algorithms. .

4.1 Datasets

To ensure an optimal solution and test environment, the proposed approaches are tested on different datasets. The applied datasets, as discussed below, are obtained from web-server logs in Norway where the last *8-bit* have been anonymized.

4.1.1 Dataset 10

The first dataset, *dataset 10* or *D10*, is obtained from *cs.hioa.no* and is gathered from October 2015 to October 2016. The dataset consists of approximately 10 million requests and contains 262 thousand unique networks. The dataset is divided into three different subsets, containing a variation in the amount of data used for the training and testing phase.

Dataset 10A

The first sub-dataset, *Dataset10A* or *D10A*, employs the entire *D10* set from *cs.hioa.no*. The first $\frac{2}{3}$ or 7,1 million web requests are used as training data, while the remaining $\frac{1}{3}$ or 3,4 million requests are used in the testing set. Of the first $\frac{2}{3}$ requests, there are around 182 thousand unique networks, while the remaining $\frac{1}{3}$ requests consist of around 124 thousand unique networks, where *80 thousands* of the networks has not been seen before.

Figure 4.1 shows the amount of requests in D10A training set that belong to a network y which contains at least x amount of request, noted as the *threshold*.

From the graph it is possible to see that only 40% of the traffic from the training phase belong to a network with a threshold of 1000. Essentially meaning that only 40% of the training data is networks with 1000 or more request. It would be preferable to cover as much of the training data as possible to be able to efficiently represent the data distribution. However, to be able to cover 80% of the data distribution it is necessary to go down to a threshold of 50 packets. While to cover 90% of the data distribution, its necessary to go down to a threshold of 20 packets. This means that there are a huge amount of networks containing few packets, while a smaller amount of networks containing a larger amounts of the network traffic. For example there are 45 000 networks with 20 or more packets, while only 380 networks with 1000 or more packets.

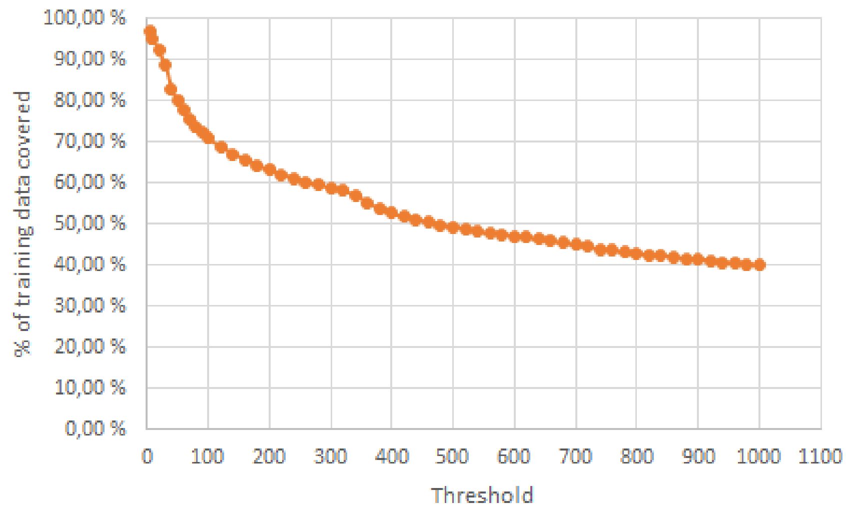

Figure 4.1: *The graph shows how much training data is covered with different thresholds for the D10A training set. A threshold x, means that a /24 bit network needs to have at least this amount of requests in total to be covered.*

Dataset 10B

The second sub-dataset, *Dataset 10B* or *D10B*, consist of *6 million* requests and stretches over 6 months, from October 2015 to April 2016. *D10B* is as the above example, divided into two sets, where the first *2/3* is used as a training set and the remaining *1/3* is used as a part of the testing set. The training phase which lasts from October 2015 to February 2016, consist of *105 172* unique networks. While the testing phase, which lasts from from February to April 2016, contains *75 340* unique networks where *49 972* Networks have not been seen under the training phase.

Dataset 10C

The third and last sub-dataset from D10, *Dataset 10C* or *D10C*, consists of 1.5 million web-request. The dataset stretches from 4 October 2015 to 10 November 2015. D10C is divided into a training set which consist of $\frac{2}{3}$ or 1 million request. The training lasts 23 days, from 4 October to 27 October. The remaining 500 thousands requests are used as a testing set and lasts to November 10. *32 891* unique networks are seen in the training set and *20 204* unique networks are seen in the testing set. Of the *20 204* networks seen in the testing set, *13 356* networks have not been seen in the training set. This means, 66% of the networks from the testing set are new networks.

4.1.2 Dataset 11

The second dataset, dataset 11 or *D11* is obtained from *Hioa.no* and is gathered from December 2016 to March 2017. The dataset consist of 3.2 million request, with *39 335* unique networks. *D11* is further divided into two subsets, subsequently; *D11A* and *D11B*.

Dataset 11A

Dataset 11A or *D11A* is the first sub-dataset from D11. The dataset consist of 1.6 million requests, where the first 1.1 million are used in the training phase, while the last 500 thousand are used in the testing phase. *D11A* lasts little over one and a half month, from December 4. 2016, to January 26. 2017, with the training phase lasting 5 weeks until January 11, and the testing phase lasting the remaining 2 weeks. The entire D11A set, consists of *24 154* unique networks, with *18 920* of the networks appearing in the training phase and *11 069* of the networks again appearing in the testing phase. Of the *11 069* networks from the testing phase, around *5 234* or *21.66%* was not previously seen in the training phase.

Dataset 11B

Dataset 11B or *D11B* is the second and last sub-dataset from D11. The dataset also consists of 1.6 million requests, where the first 1.1 million are used in the training phase and the last 500 thousand are used in the testing phase. *D11B* lasts from January 27. 2017 to March 14. 2017, with the training phase lasting 1 month from January 27. to February 28, and the testing phase lasting the last 2 weeks from February 28. to March 14. *D11B* closely resembles *D11A*, and consist of *24 075* unique networks, with *18 239* networks appearing in the training phase and *11 619* networks appearing in the testing phase. Of the *11 619* networks appearing in the testing phase, *5 836* networks was not previously seen in the training phase

4.2 Apriori-based frequent networks

The first proposed algorithm, *Apriori-based frequent networks* or *AFN*, builds on the principle that common or known prefixes which occur in the *training set* also occur later, under new and unknown data traffic. *AFN* is based on Apriori, which is an algorithm for finding frequent itemsets. Apriori focuses on finding frequent itemsets by first identifying individual frequent items in a database, before extending the items to larger items as long as the itemsets are considered frequent. Seen from training set of D10A, there exist some networks which contains high amount of data traffic. Therefore, it is reasonable to assume that traffic will reoccur from these networks under a new and unknown traffic pattern. The hypothesis for *AFN* states; *If several packets from network X reaches a server under a training phase, there is a higher chance that packets from network X, will reach the server under new incoming traffic, instead of a unknown network Y.*

AFN determines the normal network composition by going through a database of IP addresses that have previously requested resources from the server. The IP addresses are represented as a binary format. *AFN* first checks if bit *0* and *1* is frequent, meaning if they contain more than Y listings, determined by a threshold. If both of them do, the bit pattern *00, 01, 10* and *11* is checked. *AFN* continues to add *0* and *1* to every frequent bit pattern until the algorithm has gone *24* levels down, essentially meaning that only */24* bit networks which contains Y listing are left. Pseudo code for *AFN* can be seen in *algorithm 1*:

Algorithm 1 Pseudo code for *AFN* where T is the transaction database consisting of all IP addresses, F is the database consisting of all frequent items that should be returned, C is the support count that each itemset or bit format must satisfy to be considered frequent, and L is the current set of items that is checked against frequency for level k.

```
function AFN(T, C)
    L_1 ← T(Items ≥ C)              ▷ On first iteration an item is either 0 or 1
    k ← 2
    while L_{k−1} ≠ 0 and k ≥ 24 do
        L_k ← L_{k−1} + 0 ∧ L_{k−1} + 1
        for transaction in T do
            bits ← transaction.subset(k)
            if bits ∈ L_k then
                L_k ← L_k(bits) + 1
            end if
        end for
        L_k ← L_k(Item ≥ C)
        k ← k + 1
    end while
    return F
end function
```

How well *AFN* manages to represent new and unknown data depends mainly on the given data pattern. Services that have a consistent set of visitors will do well with an algorithm like this. However servers, or services that have a changing data pattern, will do moderately worse as the hypothesis no longer fit. The next section investigates to what degree *AFN* is able to represent traffic in the training and the testing phase with different applied thresholds.

4.2.1 Dataset 10A

If every network seen under the training phase are considered frequent for *D10A*, *AFN* are able to represent *60%* of the traffic from the testing phase. *Figure 4.2* shows the amount of covered data in the testing and training phase with applied thresholds from 20 to 1000. The distances between the data covered in the training and testing set decreases, as the threshold increases. This implies a correlation between frequent networks occurring in the training set, also occurring in the testing set. However, the hypothesis is not wide enough to cover a high amount of traffic from the testing set unless the threshold is set low. Further setting the threshold low, gives a high differences between the amount of covered data in the testing and training set.

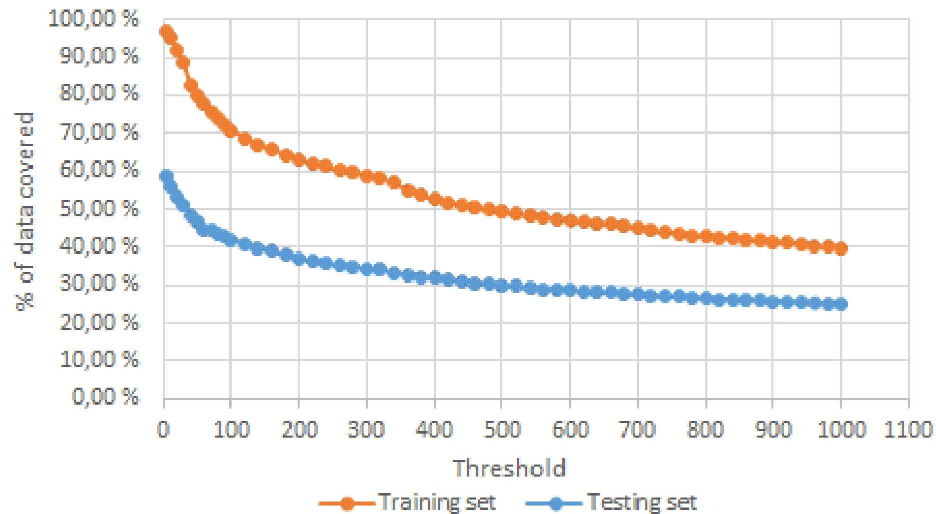

Figure 4.2: *This figure shows how much of D10A training and testing set is covered with different thresholds. The threshold is applied on the training set, before the amount of traffic from the training and testing set, which belongs to a 24-bit network, and can satisfy this criteria is counted.*

4.2.2 Dataset 10B

As both *D10A* and *D10B* contains much of the same datasets. *D10B* performs similar to *D10A* for the same considered thresholds. D10B performs with different thresholds only slightly better in form of 1-3 percentage points. *Figure 4.3* shows the amount of covered data in the D10B training and testing set if AFN is calculated on the training period with thresholds from 20 to 1000. If *AFN* employs a threshold of 1, *62,11%* of the testing set is covered. Although *D10B* is 4 million request shorter than *D10A* and the training phase goes over a period of 4 months, which is a decrease of 4 months from *D10A*, *AFN* is still only able to cover 2 percentage points more than in *D10A*. This implies that the captured data period and the subsequently data analyzes, doesn't play a significant role in deciding if a pattern is more or less legitimate. Therefore *AFN* doesn't seem to have a better ability to determine a pattern with a shorter training and testing phase.

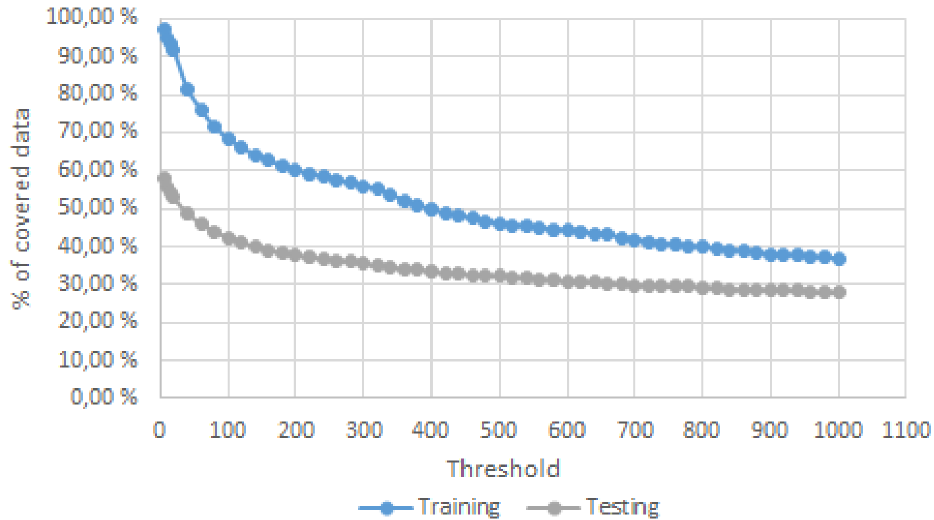

Figure 4.3: *Shows how much of D10B training and testing set is covered with different thresholds. AFN is computed with a threshold x on the training phase, before the amount of traffic from the training and testing phase, which belongs to a 24-bit network who can satisfy the threshold x is counted.*

4.2.3 Dataset 10C

Considering the same applied thresholds, *D10C*, performs worse than both *D10A* and *D10B*. AFN calculation on D10C training set with different applied thresholds can be seen in *Figure 4.4*. An applied threshold of *1000* in *AFN*, covers *17,22%* of testing pattern from *D10C*. This is a decrease from *D10A*, where a threshold of *1000* is able to cover *25,04%*.

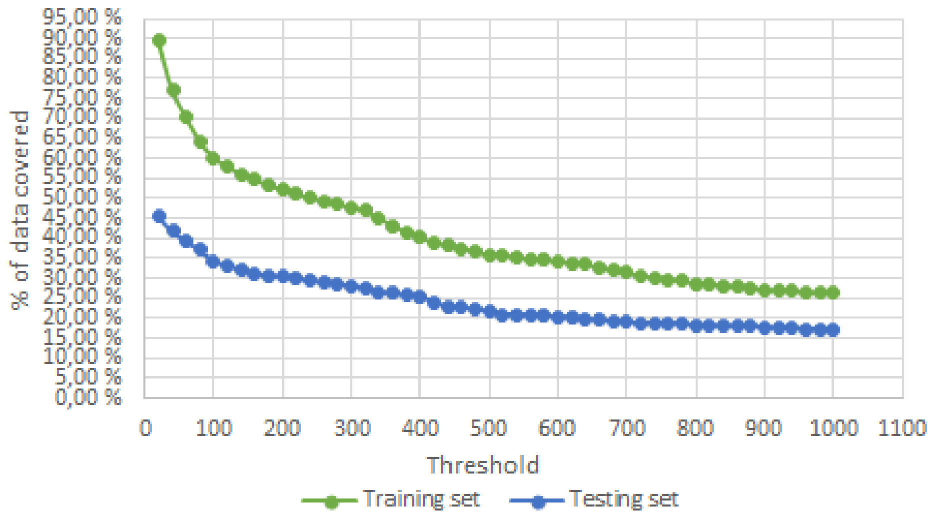

Figure 4.4: *Shows how much of D10C training and testing set is covered with different thresholds. AFN is computed with a threshold x on the training phase, before the amount of traffic from the training and testing phase, which belongs to a 24-bit network who can satisfy threshold x from the training phase is counted.*

Since *D10C* is substantial smaller than previously examined datasets its expected that *D10C* needs a lower threshold to accomplish the same amount of accepted data. To *D10C* reach a coverage of *25%*, *AFN* needs to employ a threshold of *400*. This is over half the threshold applied for *D10A*. Its hard to estimate if a dataset collected over a shorter time period is more efficient in representing new data. We can expect that some networks, regardless of time period, will continue to re-appear, while other networks will appear in periods and disappear over time. Therefore, it might be assumed that *D10C* would at least, to some degree, be able to represent new traffic better. However, *alg1* is never able to reach the same percentage of covered data regardless of threshold. For example, with a threshold of *1*, *alg1* is still only able to cover *54,38%* of traffic from the testing phase.

4.2.4 Dataset 11A

D11A performs remarkably better than sub-sets from D10. *Figure 4.5* shows the amount of covered data in the testing and training set when AFN is used to calculate frequent networks. *D11A*, as well as *D11B*, contains noticeably less noise and *AFN* is better equipped to recognize a data pattern from the training phase. With a threshold of 1000, *AFN* is able to identify *59,61%* of data from the testing phase. This stands in stark contrast to D10, where a threshold of 1, applied on different D10-subset are able to compete with *D11A*. Lowering the threshold to *1*, makes *AFN* correctly identify *86,28%* of traffic from the training

phase, which is an increase of around 20-30 percentage points when measured against D10 sub-datasets. The hypothesis; *If several packets from network X reach a server under a training phase, there is a higher chance that packets from network X will reach the server under new traffic, instead of a network Y, which was not seen in the training phase*, fits better for *D11A* than it did for D10. This means that D11A, to a higher degree, is more localized than datasets from D10.

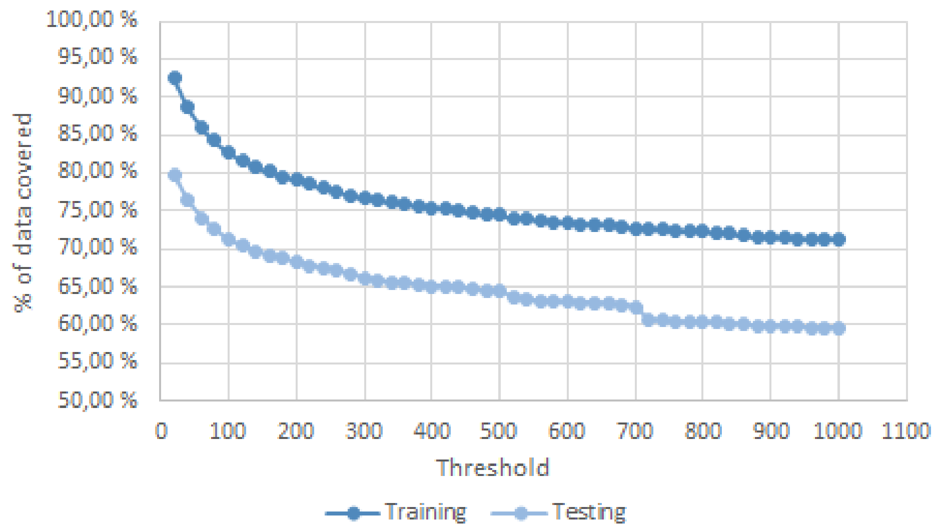

Figure 4.5: *Shows how much of D11A training and testing set is covered with different thresholds. AFN is computed with a threshold x on D11A training phase, before the amount of traffic from the training and testing phase, which belongs to a 24-bit network who can satisfy threshold x is counted.*

Comparing *D11A* against *D10C*, which is similar to *D11A*, in forms of the collected data period. *D11A* is able to cover *32* percentage points more than *D10C*. Following the hypothesis of frequent networks reoccurring, *AFN* is able to cover *42* percentage points more in *D11A* with a threshold of 1000. Although *D11A* have a larger dataset than *D10C*. *AFN* is still better equipped to identify a data pattern in *D11A* than *D10C*. This is largely due to *D11A* containing both less noise, more IP address locality and a lower amount of seen networks in the training phase than *D10C*.

4.2.5 Dataset 11B

D11B performs slightly weaker than *D11A*, still impressively better than datasets from *D10*. *Figure 4.6* shows the amount of covered data in the D11B training and testing set if AFN is calculated on the training period with thresholds from

20 to 1000. With a frequent threshold of *1000*, *AFN* is able to identify *57,27%* of all new traffic from the testing phase. While with a threshold of *1*, *AFN* is able to identify *81,66%* of all new traffic.

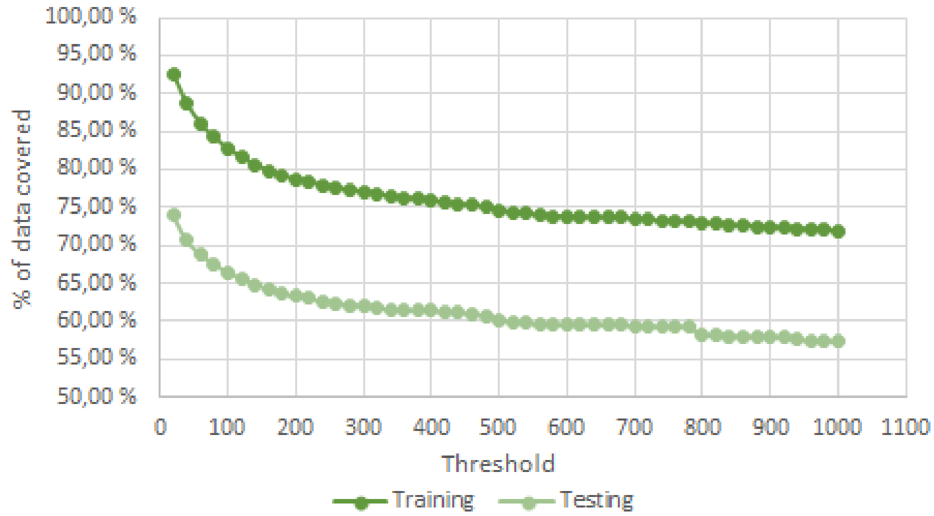

Figure 4.6: *Shows how much of the 11B training and testing set is covered with different thresholds. Alg1 is computed with a threshold x on 11B training phase, before the amount of traffic from the training and testing phase, which belongs to a 24-bit network who can satisfy threshold x is counted.*

Although, *AFN* works well on datasets from *D11* it doesn't manage to achieve a high acceptance rate for *D10*. The solution might be to use a more complex hypothesis. We could assume if traffic are seen in network x, new and unknown traffic have a higher likelihood of occurring in some networks close to network x. An easy way to see if this hypothesis have any grounds, would be to do AFN calculations with lower bit levels of $y<24$. This means with an y bit threshold. The first y bits, needs to be over a set frequency, to be considered frequent. *Figure 4.7* shows AFN calculations on D10C with y values of 16, 18, 20 and 22. The covered data pattern from D10C testing phase is counted with different y values and frequent thresholds applied on the training phase.

A lower bit amount covers more data in the testing phase. Letting AFN count down to 16 bits and accepting all 16-bit networks with 1 packet, *86,63%* of data from the testing phase are accepted. The high amount of accepted data is not an indication that the hypothesis is wrong, as simply counting down to a lower bit amount will naturally cover more data regardless of the traffic pattern. Therefore, we should ask the question; *Is the new percentage of accepted data, with a lower bit-threshold, acceptable in terms of being able to represent an equal*

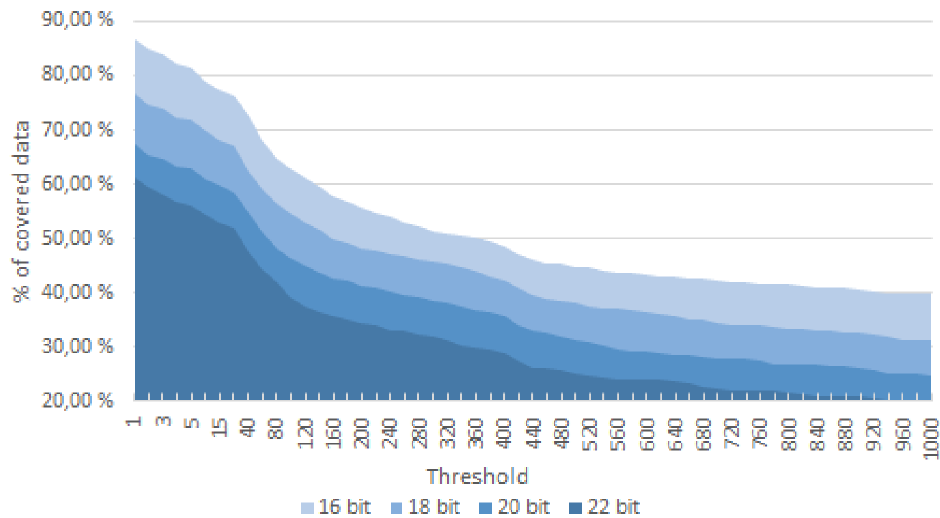

Figure 4.7: *Shows how much of D10C testing set is covered with different thresholds and different levels applied on D10C training set. Instead of forcing AFN to find frequent 24 bit networks as earlier version, AFN is computed with 16, 18, 20 and 22 bit networks.*

number of data from the testing phase, in proportion to the higher number of accepted data in an overall sense?. *9474 16*-bit networks is accepted with a threshold of 1. This translates into *14,45%* of all possible *16*-bit networks. By looking at the same bit-threshold value in a *23*-bit network *57,21%* of the testing set is accepted, whereas *22 972* 23-bit networks are accepted. This amounts to around 0,27% of all possible networks.

Therefore, even though this hypothesis have the ability to increase the percentage of accepted data, the amount of accepted data from *D10C* is highly disproportionate to the amount of overall accepted traffic. The hypothesis, that traffic reoccurs close to networks which have previously been seen, has some truth to it. However, it does not define an accurate pattern which limits the overall acceptance of data and heightens the acceptance of legitimate traffic.

4.3 Density-based geographical clustering

Although *AFN*, to some degree, depending on the dataset, is able to classify new traffic, the hypothesis, that frequent networks reoccurs, lacks the ability to cover a high pattern unless the threshold is set low. Setting the threshold to a low value, further risks that networks, which are not necessarily a good representative for the data pattern, are accepted. Therefore, a second method which tries to overcome the shortcomings of *AFN* is proposed. The new density-based geographical clustering algorithm, *DGC*, tries to estimate a data pattern based on location and builds on the hypothesis; *If network x from location y reach the server under the training phase, there is a higher chance for network z from location y+1 to reach the server, than network q that doesn't belong to a location close to network x.*

DGC begins by defining core points or start points from where a cluster can continue to expand. The core points will then look in their close proximity based on distance x to see if there are more points that can belong to this cluster. If a core point finds a point y that is within distance x, it will include the point to its own cluster. Point y will then look in its close proximity to find any new points that can be included in the cluster. If any points that are a part of a cluster is within the given distance to other points that is a part of a different cluster, the two clusters will merge and become one cluster. The pseudo code for this density-based clustering can be seen in *algorithm 2* below:

Algorithm 2 Pseudo code for this density-based clustering algorithm, DGC. The algorithm takes three arguments; the dataset containing a list of latitude/longitude locations with the frequency of packets from that locations. D, which is the maximum distance from a point to a cluster for this point to still be considered a part of that cluster and T which is the frequency threshold for a point to be considered a core point.

```
1:  function DENSITY_CLUSTERING(dataset, D, T)
2:      all_clusters ← (dataset.points >= T)
3:      for cluster in all_cluster doy
4:          cluster ← (dataset.points <= cluster.D)
5:          while cluster ≠ converged do
6:              points ← (dataset.points <= cluster.D)
7:              if points in all_cluster.points then
8:                  cluster.merge(points in all_cluster.points)
9:              end if
10:         end while
11:     end for
12:     return all_cluster
13: end function
```

The clustering algorithm, *DGC*, is based on *DBSCAN*. Unlike other clustering algorithms, where a points association to a certain cluster heavily relays on a centroids initial start positions, as well as the subsequently re-position of centroids based on expansion. Density-based clustering, ensures that a points association to a cluster is based on the points ability to be density-connected to a certain cluster, and not based on the points closest centroid.

Because of the algorithm, two essential parameters; Subsequently core-point threshold and maximum distance, needs to be pre-defined. The definition of these attributes will vary greatly between different datasets and the consequently data pattern that these clusters cover, will have different strength and weaknesses based on the chosen attributes. The next sections will to a certain degree investigate the optimal choice of core-point thresholds and the best maximum distance. Before, both the optimal result of a solution that accepts the seen IP addresses in a cluster and the optimal result of a solution where the location of all new traffic is correctly identified, are calculated.

4.3.1 Deciding core-point threshold

Defining start points is a hard computationally task, as this definition will essentially define how and where the cluster will expand, and how well the clustering algorithm will be able to represent new and unknown data. From the last section, we have concluded that frequent networks, often in some form recur, in the data pattern. Since frequent networks represent a huge portion of the incoming traffic it is reasonable to assume that every frequent network or in this case location, is automatically assumed as a start point or core point.

Since this definition of frequent locations, or core points, is just the definition of an initial cluster, which is just a single start point for a cluster to continue to expand, it is essential to decide a threshold for when a location point is considered frequent. This threshold would vary greatly between different datasets. For our own dataset *D10C*, the amount of initial clusters, with different thresholds, can be seen in *Figure 4.8*. A low threshold will, as seen by the graph, exceptionally increase the amount of initial clusters, compared to a high threshold. However, this is not necessarily a good thing as this will create clusters where there are seen considerable little data traffic.

For *D10C*, an initial core-point threshold of *60*, *80* or *100*, creates *3742*, *2826* or *1141* initial clusters. Setting an exceptional higher threshold of *1000*, will create *141* initial clusters and setting the threshold to *1720*, will further divide this amount in half and give an initial clusters amount of *71*. When deciding the initial core point threshold, we should chose a point in the graph, from *Figure 4.8*, where the exceptional decrease of clusters, more or less have flat-lined. From the graph, we can conclude that a high decrease of initial clusters, flat-lines either around a core-point threshold of *700-1000*, or even around *1500*, when only *85* initial clusters are left.

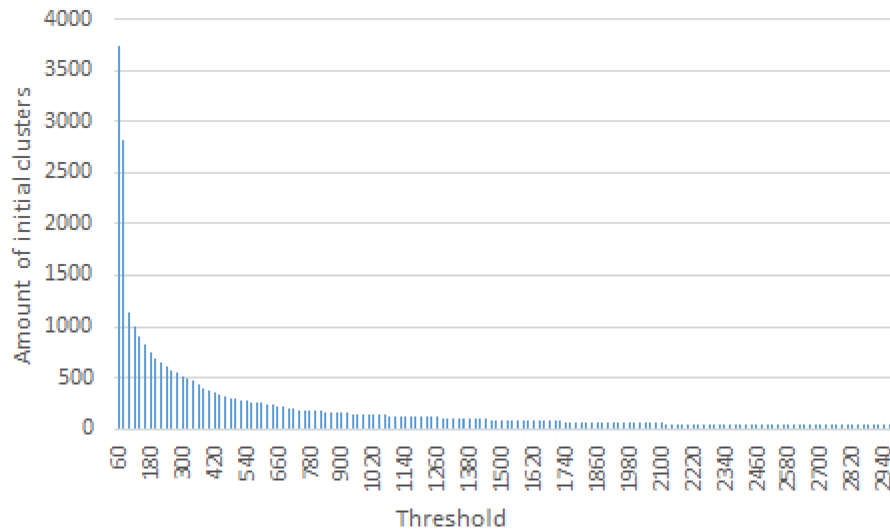

Figure 4.8: *This figure shows the amount of initial clusters, for D10C, with different core-point thresholds. Initial clusters are singular points which have a frequency of the same, or higher, than the given core-point threshold.*

The amount of initial clusters should be seen in correspondence with the amount of initial covered data. Since we want clusters to expand, we don't need to cover most of the training pattern from the beginning, as we assume, most of the relevant data pattern, will still be covered, when the cluster expand to points nearby. As seen in *Figure 4.9*, by looking at *D10C*, a low applied core-point threshold will cover most of the initial training data, while a high core-point threshold will contain notably less data. A subsequent low threshold of *100*, *280* and *520*, will cover *85,15%*, *75,27%* and *65,24%* of the given data pattern. While a notably higher threshold of *700*, *1000*, *1400*, *1800* or *2600* will cover *60,47%*, *55,49%*, *50,53%*, *45,10%* and *40,47%* of the data pattern from *D10C*.

As very high thresholds are able to cover a substantial amount from the training phase, as well as lowering the amount of initial clusters, a high threshold above *1000* would be preferable, as a threshold from *1000-2600* is still able to cover *40-60%* of the data pattern. Although the threshold most likely will be similar for *D11A* and *D11B*, a higher threshold, should in most cases be set for *D10A* and *D10B*. As in these instances, the training period lasts over a longer time and therefore a higher threshold should be set to cover the most essential data pattern.

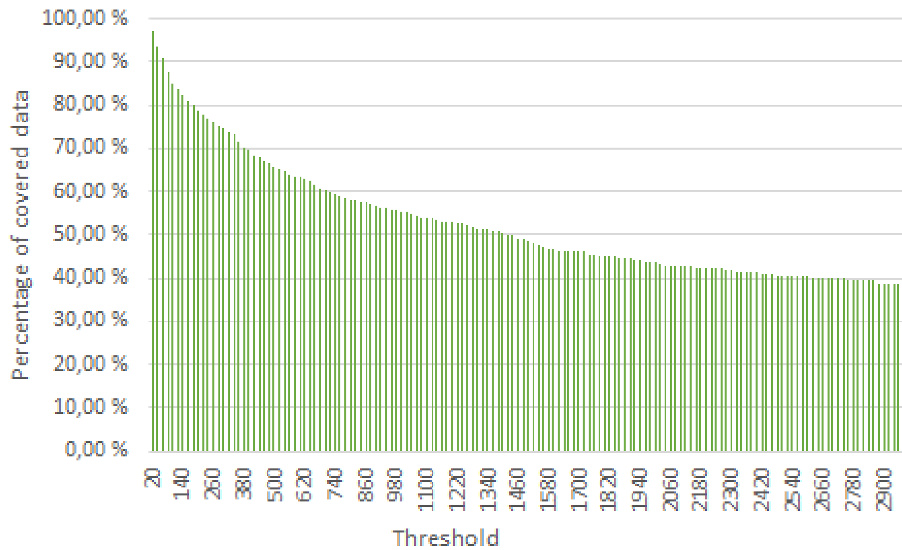

Figure 4.9: *This figure shows how much training data that is covered with different DGC cluster computations. DGC clusters are created with different core-point thresholds and a km distance of 0. Therefore, only the initial cluster start is covered and not the later cluster expansion. The cluster computation are done on D10C. Before the amount of covered data in the training phase are counted.*

4.3.2 Deciding maximum distance

As we are assuming, from the hypothesis, a lot of lower networks or locations with less frequent packets, will still be covered as they exist in the surrounding area of the defined core points. This leads to the question; what is the maximum distance for point x to point y, for point x to still be a part of point *y's* cluster? We have previously stated that every core point is automatically assumed as a cluster, the question now is; What is the maximum distance for 1 or more cluster points to any new point, for the point to be a part of the given cluster? The answer to this question would vary widely based on the training data. However, we should try to minimize the maximum distances, so we don't end up with clusters where data have little to none similarities with other data in the same cluster. Based on our hypothesis, we can assume that data points exist close to our cluster in all directions of the initial core points. Therefore we can, based on this, get a certain view of the necessary distance, by measuring how much training pattern is covered with different maximum distances.

Figure 4.10 shows how much data, for *D10C*, which is covered with different thresholds and maximum distances. A lower threshold will cover more of the training data, regardless of the maximum distance. It would be preferable to cover a high degree of the training as this will result in a higher coverage for the

test data. However, the coverage of new points should be proportionate to the higher maximum distance. In other words, we should be able to assume that a cluster does not expand forever. Therefore, when the percentage increase of new covered data decline radically with a higher maximum distance, there is little to none reason to continue to increase the maximum distance.

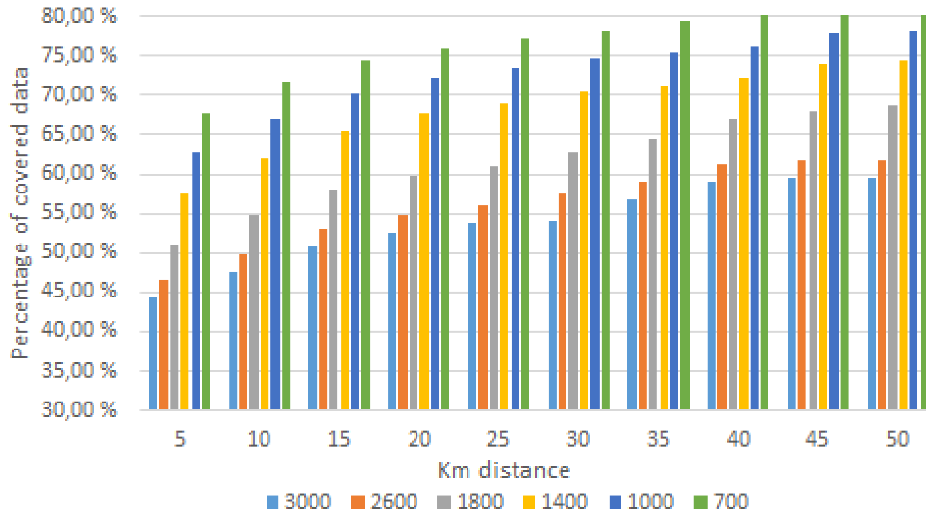

Figure 4.10: *This figure shows how much of training data, for D10C, which is covered with different DGC cluster computations. Clusters are computed with core-point thresholds of 3000, 2600, 1800, 1400, 1000 and 700 and km distances between 5 and 50.*

Figure 4.11 gives a clearer view of the difference in increased percentage points for each threshold. The figure shows the increased percentage points of accepted training data for different thresholds. The calculations is done with km distances between 5 and 50. Most clusters have a good increase in the amount of covered data until around 20-25 km. At this point, the different calculated clusters have managed to increase around 10 percentage points. At the remaining calculated distances of 30-50 km, clusters have a high variance, with no particular pattern for the remaining increased percentage points. It would therefore, for most clusters, be beneficial to have a maximum km distance between 20-25 km.

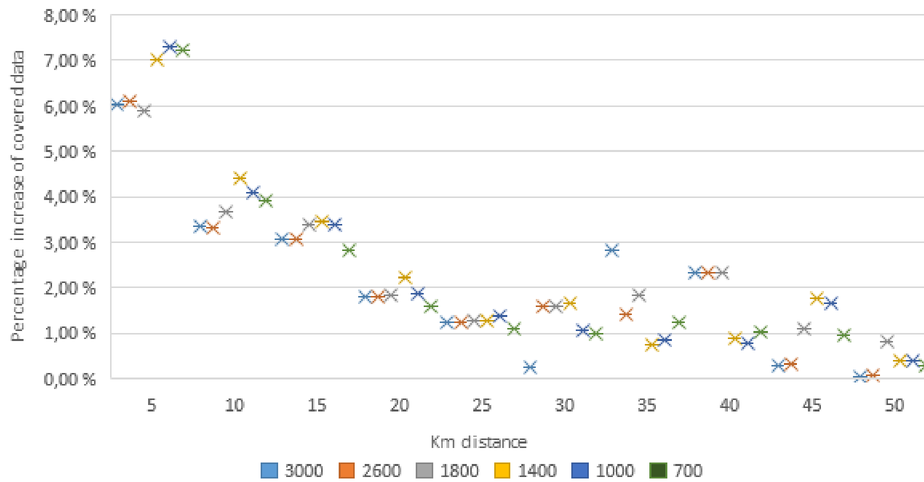

Figure 4.11: *This figure shows the increase in amount of accepted training data with different DGC cluster computations. DGC clusters are computed on D10C with core-point thresholds of 3000, 2600, 1800, 1400, 1000, 700 and km distances between 5 and 50. The graph starts on 5 km, which means the increase in covered percentage points from the initial cluster start of 0 km.*

4.3.3 Optimal result

To find the optimal result for a solution that is able to look up IP addresses and identify if it fits in a cluster, we should not only count the direct points that have appeared in the training phase, but also count points that fits inside a cluster, which has not necessarily been seen in the training phase. To accomplish this, we can create polygons based on each cluster before checking if data points fit within each polygon. An example of this computation can be seen *Figure 4.12*.

Since polygons are created based on the outer most points in each cluster, creating polygons imposes a weakness when clusters have points that lie in the outlier of the corresponding cluster. A stricter algorithm can be applied which will ensure that outliers don't affect the accepted data pattern. However, an accepted data pattern, with loosely constructed polygons, only increases a couple of percentage points between a solution that only counts points that have been directly seen in the training phase. For example, if cluster computations is done on D10C with a threshold of *1000* and a km distance of *25*. *64,96%* of the testing set is covered, when counting only the direct accepted points. On the other hand, if we count clusters as a polygon, *68,5%* of the traffic from the testing phase is covered. These few percentages will accounts for around *20 000* requests.

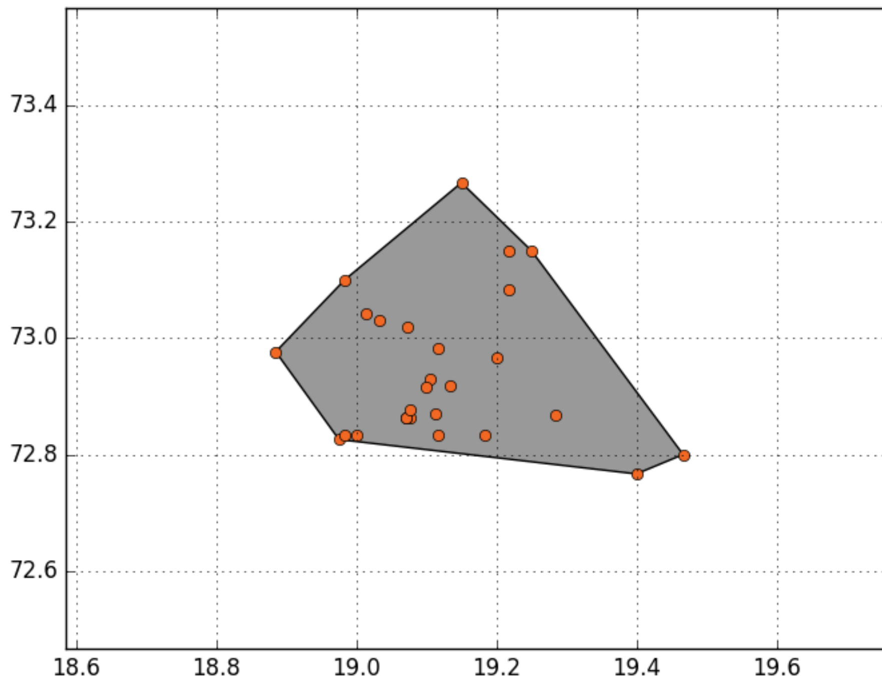

Figure 4.12: *Shows one small cluster created from an initial cluster threshold of 1000 with a maximum distance of 25 km. The cluster is created from D10C. In an optimal result all new data within the grey zone is accepted as a part of this cluster.*

This next sub-section will investigate both the optimal result and non-optimal result that different DGC cluster computations are able to accept in the testing set. An optimal result, will be if a packet can be directly identified in a cluster based on its own locations. A non-optimal result is if packets location can't be identified. Therefore, only clusters in the training phase with those clusters identified networks are accepted in the testing phase.

D10C

Figure 4.13 shows the amount of optimal accepted data in the D10C testing set with different calculate DGC clusters. The DGC clusters are created with km distances between 5 and 50, and core-point thresholds of 3000, 2600, 1800, 1400, 1000 and 700. Optimal *DGC* clusters are able to achieve an acceptance rate of around 43% to 77% for the testing phase. *AFN* managed to achieve an acceptance rate of *54,38%* in the testing phase. This means that optimal cluster calculation are able to achieve over 20 percentage points more than AFN on the same dataset. Lower cluster calculations have a harder time of reaching the same

amount of accepted data as AFN. While more loosely constructed parameters with higher km distances or lower core-point thresholds have a easier time of reaching or even surpassing the amount of accepted data as AFN.

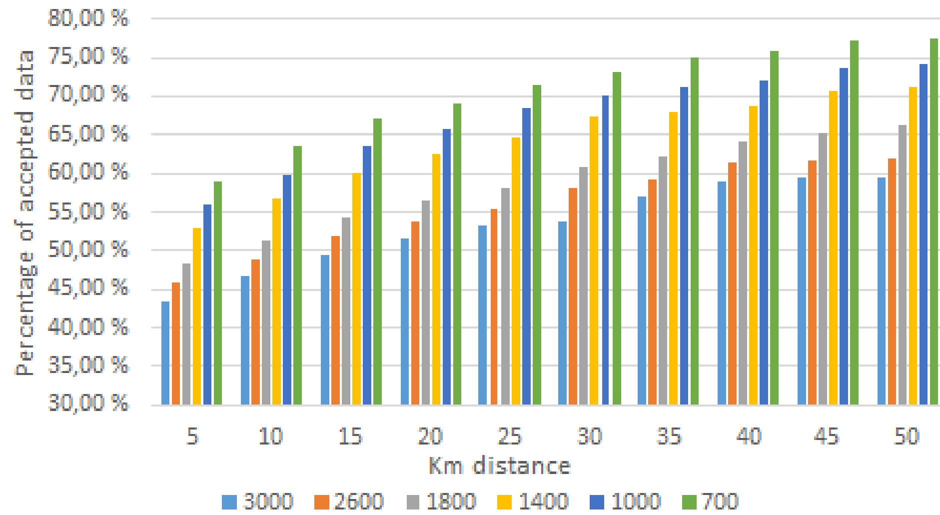

Figure 4.13: *This figure shows an optimal amount of accepted testing data, when DGC clusters are created on D10C training set. Clusters are here compute with the high core-point thresholds of 3000, 2600, 1800, 1400, 1000 and 700 and km distances between 5 and 50 on the training set. The amount of accepted packets from the testing set which belong to a cluster based on its location are then counted.*

For the non-optimal amount of accepted data, the amount ranges between 36% to 50%. Therefore, an optimal solution which are able to identify if a packet fits within a cluster, will perform 7 to 27 percentage points better than a non-optimal solution.

D11A

D11 is more localized than D10. This means D11A is better equipped to cover a high data pattern without needing to cover a high amount of location points and the subsequent networks that these points have. *Figure 4.14* shows the amount of optimal covered testing data when DGC clusters are created on D11A. DGC clusters are created on the training phase with different km distances and core-point thresholds of 3000, 2600, 1800, 1400, 1000, 700. When packets can be correctly identified in a cluster based on its own location, DGC are able to cover between 86% and 93% of the testing set. Most DGC clusters are able to cover around 90%. As *AFN* are able to cover 86%, DGC clusters are able to

cover 7 percentage points more than *AFN*.

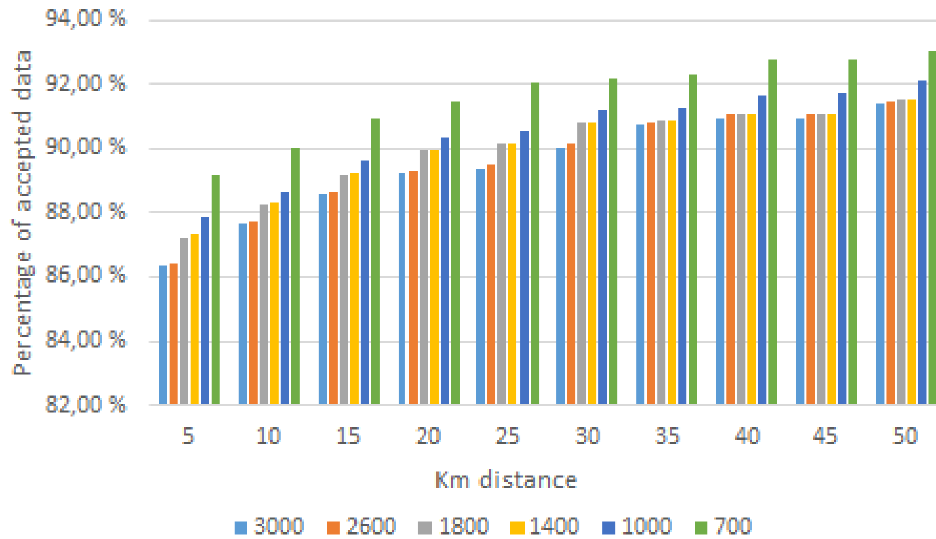

Figure 4.14: *This figure shows an optimal amount of accepted testing data, when DGC clusters are created on D11A training set. Clusters are here compute with the high core-point thresholds of 3000, 2600, 1800, 1400, 1000 and 700 and km distances between 5 and 50. The amount of accepted packets from the testing set, which belongs to a cluster based on its location, are then counted.*

The low increase in accepted optimal data between *AFN* and *DGC* can be contributed to several reasons; DGC clusters in D11A already cover a high amount of data, which makes it harder to cover remaining percentages, as some traffic might be anomalies. Secondly, D11A are more localized than D10. Because of this, the dataset doesn't have a high increase in covered data pattern with more loosely constructed parameters. For a non-optimal amount, DGC clusters are able to accept from 79% to 83% when only counting networks which have been seen in the training phase and is a part of a cluster. Between an non-optimal amount and an optimal amount, DGC clusters are able to increase the amount of accepted data with 7 to 9 percentage points.

D11B

D11B follows much of the clustering trend from D11A. D11B has also little to none expansion in accepted traffic for higher km distances. This is mainly because D11B is highly localized and higher km distances will often not affect the data pattern to a high degree. D11 points often lay in tighter cluster and singular points contain more data than what was seen for D10C. The D11 set

therefore doesn't need to expand to a notable degree to contain a high amount of data. An optimal amount of accepted testing data for D11B can be seen in *Figure 4.15*. *DGC* clusters are here created on D11B training set with core-point thresholds of 3000, 2600, 1800, 1400, 1000, 700 and km distances between 5 and 50. The amount of accepted data from D11B testing set, which fits within a cluster based on its location, is then counted. *DGC* clustering manages to cover between 77% to 88% of the testing phase. In comparison, AFN manages to cover 81,70% of traffic from the testing set. *DGC* clustering are therefore able to cover around 7 percentage points more than *AFN*.

Figure 4.15: *This figure shows an optimal amount of accepted testing data, when DGC clusters are created on D11B training set. Clusters are computed with the high core-point thresholds of 3000, 2600, 1800, 1400, 1000, 700 and km distances between 5 and 50. The amount of accepted packets from the testing set, which belongs to a cluster based on its location, are then counted.*

In comparison, we should consider the amount of non-optimal accepted data. If we only accept networks which have been seen in a cluster under the training phase, DGC clustering manages to cover between 71% to 78%. An optimal acceptance rate for *DGC*, therefore manages to cover 2% to 10% more than a non-optimal solution. *DGC* clustering manages to overcome some of the short-comings of *AFN*. While AFN depends on the exact network in the training phase repeating in the testing phase, *DGC* relays on a packets location repeating. This can amplify the amount of data which can be covered. Moreover, *DGC* clustering often manages to cover a close amount to *AFN* when only accepting networks, which have been seen in the training phase, and which are a part of a cluster. *DGC* Clustering are therefore better equipped to define a unique

pattern. DGC clustering will also differ between lower amount of seen networks based on its location and can therefore more correctly remove anomalies in the traffic pattern.

4.4 Reduced-density geographical clustering

Although, *DGC* has a potential of acquiring a unique pattern and performing well under unknown traffic, the algorithm doesn't define any considerations when defining a pattern. In a sense the algorithm's simplistic approach is also the algorithm's greatest weakness. The previous algorithm simply states; If a point x is near one or more points that are a part of cluster y, point x should also be stated as a part of *y's* cluster. This is not necessarily ideal, as multiple issues are raised in regards to the algorithm's ability to sustain a pattern and the algorithm's ability to achieve the requirements set by the hypothesis.

The hypothesis; If network x from location y reach the server under the training phase, there is a higher chance for network z from location $y+1$ to reach the server, than network q which doesn't belong to a location close to network x, empathizes important points that can be used to create a better algorithm. First and foremost, we should make sure that clusters only expand to points that are most likely to occur later. We can build on the hypothesis and make the assumption that a request is more likely to occur in a point x, if it has at least y points in its near area. Only points that satisfy this criteria, are allowed to be a part of a cluster. However, the point will only be a part of a cluster, if the point is connected to a core point based on normal density based estimation.

Secondly, to prevent the pattern composition in clusters from deteriorating when clusters are merged, merging of clusters is overlooked. Instead, clusters are forced to expand and converge separately. This raises important issues in how a cluster is allowed to expand, that would best confirm and acknowledge the hypothesis. Seen in *Figure 4.16*, after placements of the core points, a point x might fit with several clusters based on only density estimation and the minimum amount of required nearby points.

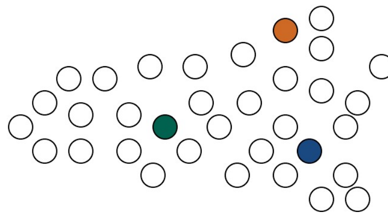

Figure 4.16: *Shows several initial clusters before expansion. All points are density connected to all other core points, which makes it hard to decide what point should be with which cluster.*

The problem bounds down to the issue; If point x based on density based clustering would fit in both cluster Y and cluster Z, which cluster should the point be considered a part of? The entire hypothesis builds on the principle that data will recur around locations that have occurred often. Therefore, it would

be sensible to group an unknown point x, to the closest core point, if it satisfies the constraint set by density-based clustering for this core point. Therefore, if $x_1, x_2, ..., x_{n-1}, x_n$ is a set of data points in a two dimensional space and $c_1, c_2, ..., c_{n-1}, c_n$ is a set of core-points which contains at least y amount of seen request. Then, if $d(x_m, c_z)$ is the geographical distances between point x_m and core point c_z. While $b(x_m, c_n, r),\ )$ is a core-point that x_m is density based to based on radius r. Meaning that there exist a point x_n within x_m's radius which also are density connected to c_z. x_m will be a part of cluster c_z if $d(x_m, c_z) < d(x_m, c_1), ..., d(x_m, c_{n-1}), d(x_m, c_n)$ for all core points which support the notion of $b(x_m, c_n, r)$.

Figure 4.17: *The first picture shows the end result of a cluster computation where all points which fit with the minpts constraint is set to be a part of its closest core point, regardless of the points ability to be density estimated to this core point. The second picture shows the end result, when points also are required to be density estimated to this core point. The clusters are computed with a threshold of 1000, distance of 20 km and a minpts constraint of 4. At this point the minimum length is zero, which means that single core points are allowed.*

If all points were able to be connected to their closest core point, the result would look like the first picture in *Figure 4.17*. Here, points that support the notion of *minimum required points* is connected automatically to the closest core point. However, as the second constraint needs to be fulfilled with points being density-based to their own core point, the final result will look like the last picture in *Figure 4.17*. As seen by the graph, some initial core points that first got points to their cluster, will not get any points when the cluster computation is finished. This raises issues in regard to a third constraint; How to deal with initial clusters if they don't receive any or few points. Simply maintaining the initial clusters as independent clusters is a possibility. However, this can cause issues if many points are not able to expand. Even though there is a higher chance for a request to reoccur in the same frequent location. Singular clusters heavily favors large cities. Instead, a third and last constraint is therefore employed to remove or relocate clusters which are not able to expand. Therefore,

if a cluster does not exist of x points, the cluster should be dismantled. The points, including the core point, should be re-positioned to other clusters nearby. The re-positioning of dismantled points should then be done in regards to the points closest core point, excluding the core point for the dismantled cluster.

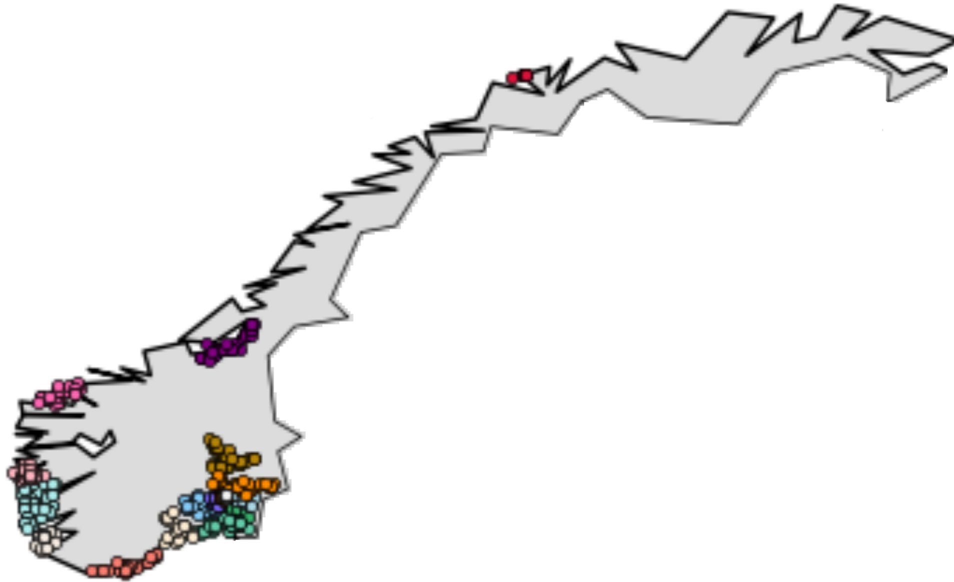

Figure 4.18: *Shows one end result RDGC clustering in Norway. The clustering algorithm was computed with a maximum distance of 20, core point threshold of 1000, a minimum nearby points of 4 and a minimum length of zero.*

This modified *DGC* algorithm are further denoted as *RDGC* or *Reducedr-density geographical clustering*. The end result of *RDGC* can be seen in *figure 4.18*. It's important to note, that clusters often don't form a sphere in high density areas. Core-points, in high density areas, lay fairly close to each other, which make core-points compete about nearby points. Clusters are instead forced to expand outwards.

RDGC goes through an iterative phase for all points that can satisfy the *minimum points* constraint. The points, that are able to satisfy this constraint are first checked for density to its closest core point. When there are no more points that are able to be density based to its closest core point, the second closest core point is checked. This process continues, until either all points have found to be density based against a core point, or if there are no more core points left to check. After the cluster expansion is finished, all clusters are checked for minimum length. If a cluster doesn't satisfy this criteria, cluster points are, if possible, relocated to other clusters nearby. *RDGC* is still able to keep the gained strength from density based clustering, as it largely still enables

clusters to expand in any shape. The different clusters are only constrained if there are other core points nearby, or if the cluster tries to add points that lay in the *outlier* of its own cluster.

With this new proposed clustering algorithm, two different areas need to be investigated to ensure an optimal result; First, what amount of *minimum required points* would ensure the most optimal result? And secondly, is it beneficial for *RDGC* to remove clusters that are not able to expand properly? If it is, what minimum length should a cluster be to be considered legitimate? The next sub-section, will investigate these issues, before the amount of data, both an optimal and non-optimal solution are able to accept, are calculated.

4.4.1 Deciding minimum required points

To check the first problem statement, clusters are computed with same threshold and km distance, but with different *minimum required points*. *RDGC* clusters are computed on D10C training set with km distance of 20, core-point threshold of 1000 and minpts variables between 1 and 10. The amount of accepted data in the training phase which fits within a cluster are counted and can be seen in *Figure 4.19*.

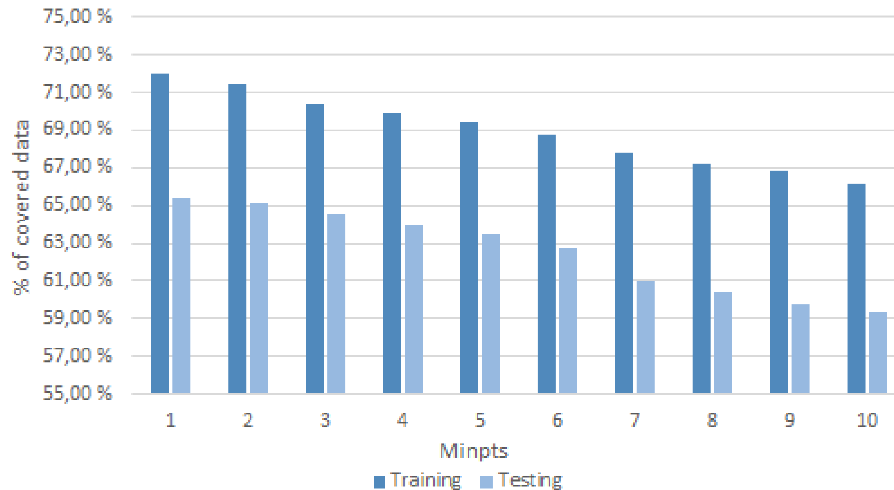

Figure 4.19: *Shows the amount of accepted D10C training and testing data, when RDGC clusters are calculated with different minpts values, core point threshold of 1000 and a km distance of 20. The optimal result is calculated for the testing set. This means data only needs to fit location wise with a cluster.*

minimum required points variable considered here doesn't mean points need to have this amount of points in z distance, which is a part of the same cluster. It simply means that a point x, needs to have y amounts of points in distance

z to be considered legitimate. Therefore, other points relationship to a cluster is considered when deciding if a point is legitimate.

The *minimum required points* value is meant to prevent outliers in the dataset. Therefore, we should choose a value that removes most possible points in the training set, compared to the least amount of data covered in the testing set. A clearer view of the amount of lost covered data with higher *minimum required points* can be seen in *Figure 4.20* for D10C. The training set continues to lose more points, in form of percentage points, until the *minimum required points* is set to *4*. This implies the best minpts value is 3. At this point, the training set has lost *1.60%* of the covered data, while the testing set only have lost *0,86%* of the covered data. This means DGC clusters on D10C, calculated with a core point threshold of *1000*, km distance of *20*, minimum required points of 3, are able to remove *354* cluster points. The 354 cluster points can therefore be removed without noticeable affecting the covered data in the testing phase.

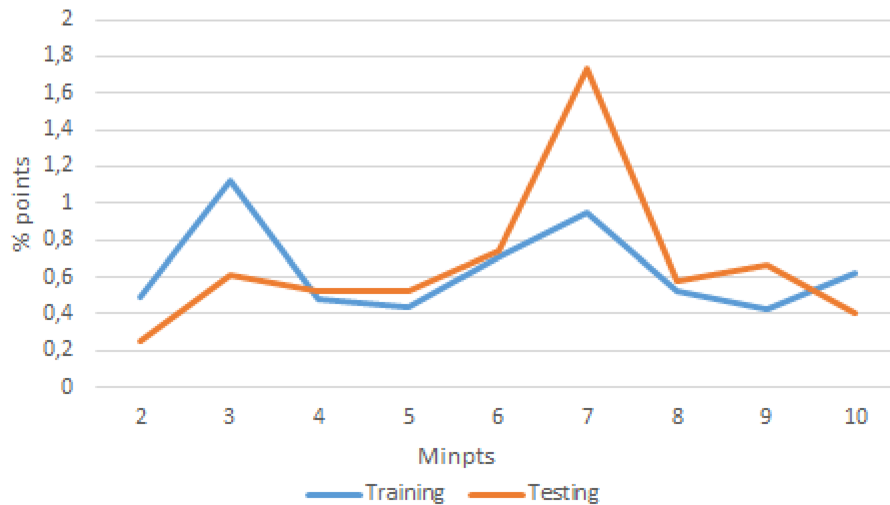

Figure 4.20: *Shows how much covered data in the training and testing set for D10C decreases, in forms of percentage points, with different minpts values. Minpts value x, means the percentage point decrease from minpts value x-1. Clusters are calculated with different minpts values, a km distance of 20 and a core point threshold of 1000. The optimal result is calculated for the testing set. This means data only needs to fit location wise with a cluster.*

The calculated training set will regardless of different *minimum required points* always lose more covered data in total. However, since D10C training set consist of 1 million requests, while D10C testing set consist of 500 thousand requests, the training set will in most cases always lose more data than the testing set. It's therefore important to look at the lost percentage points of

covered data, in regards to the amount of data in total to get a correct view of how different minpts values affects the data pattern.

4.4.2 Deciding minimum length

Secondly, we should calculate if it's beneficial to remove clusters that are not able to expand. Both previous figures; *Figure 4.19* and *Figure 4.20*, are calculated without the considerations of relocating points or removing clusters that are not able to expand. For an efficient solution to represent a unique and narrowed down data pattern, we should preferable remove clusters that are not able to expand, as long as this does not extensively, and in a negatively way, affect the covered data traffic. There are two different approaches to decide the *minimum length*; We can either remove singular core points that are not able to expand, or we can remove or relocate clusters which don't have a certain length.

Removing singular core points

When only considering the event of removing singular core points, the amount of covered data doesn't change much from DGC. RDGC clusters still largely cover the same amount of data, as most points, which exist in a merged clustering algorithm, also exist in a non-merged clustering algorithm.

When DGC clusters and RDGC clusters are computed on D10C, with a km distance between 15 and 30, core-point threshold of 1000, and RDGC clusters have added an extra limitation of *minimum required points* 3 and *minimum length* 2. DGC clusters are able to cover between 63,52% and 70,10% of data from the training phase. RDGC clusters are able to cover 57,80% to 66,35% of data from the training phase. This is a slight decrease of 4 of 6 percentage points from DGC. However, RDGC is still able to cover a large representation of the traffic pattern.

Remove cluster without a certain length

Although simply removing singular core points might be enough, it would be interesting to investigate a solution which removes or relocates clusters without a certain length; Both how optimal a solution would perform, and if it is better than just removing singular core points.

A RDGC clustering, with just removing singular points, can still give a large area of unnecessary clusters. Figure 4.21 shows a RDGC clustering result for D10C, when clusters are computed with km distance of 20, minimum required points of 3 and minimum length of 2. When a major data pattern of traffic is situated in Norway, we should preferable remove clusters from places like India. Removing only singular clusters will not necessarily accomplish this. This might not be a large problem with a solution that just accepts 24-bit binary addresses which have occurred in the clusters. However, for an optimal solution, which

are able to identify the location of any new packet, a solution should preferable accept data from as few locations as possible.

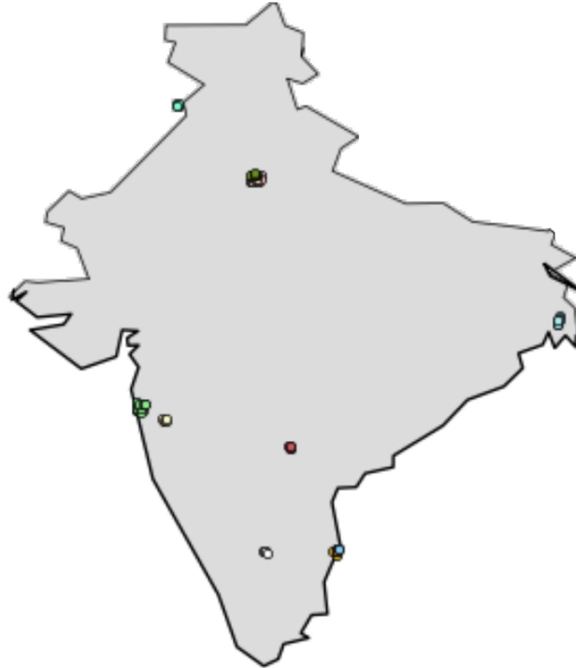

Figure 4.21: *Shows the end result of clustering with RDGC in India. The clusters are computed with a core point threshold of 700, km distance of 20 and a minimum required value of 3. Only single core points, that are not able to expand, is either removed or relocated.*

RDGC clusters calculated with km distance of 20, core-point threshold of 1000, minimum points of 3 and different minimum applied lengths, can be seen in *Figure 4.22*. Clusters are computed on D10C. The amount of optimal accepted testing data and the amount of accepted training data, are calculated and shown in the graph. Different minimum applied lengths do radically and fast diminishes the amount of accepted data. Both the training and testing phase decrease fast with a higher minimum cluster length. However, the amount of covered data in the training phase decreases faster than the amount of covered data in the testing phase. At a minimum length of 10, the amount of covered data in D10C training set decreases below the amount of covered data in the testing set. This means there is no longer a difference between the amount of accepted data in the training and testing phase.

With a minimum length of 10, 56,11% of traffic from the testing phase is accepted. This is a decrease of around 9 percentage points compared to DGC,

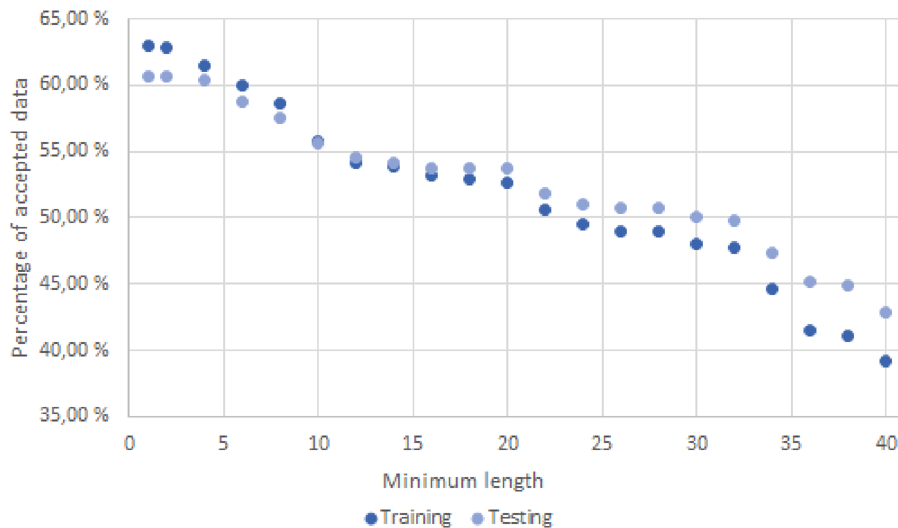

Figure 4.22: *Shows, for D10C, the amount of covered data in the training and testing phase with different minimum lengths constraints set for a RDGC cluster. RDGC Clusters are created with a core-point threshold of 1000, km distance of 20 and a minimum required value of 3*

which have around 65,63% of testing data accepted. Limiting the amount of clusters this way, can be a good way to prevent unnecessary geographical areas from having access to a service. However, limiting based on the amount of cluster points, will in most instances also have a negative effect on the data pattern. As limiting simply based on points, and not based on the amount of seen data in a cluster, will, to a certain degree, risk that clusters with a lot of data, but few points are eliminated.

4.4.3 Optimal result

It's important to note that values such as minimum length, core point threshold, minimum required points and km distances would vary between different datasets. Stricter requirements will make it harder for unwanted traffic to get access to a system. However, it will also prevent more legitimate traffic from entering. Easier requirements will include more traffic, as well as being more prone to letting in more unwanted traffic under an attack. The trade-of between how strict a cluster creation is, with the correlation of letting in less traffic, will change between each dataset and the amount of traffic that is considered. Webservers which have traffic from few locations can in a higher degree have a more strict threshold, than services which have traffic from a lot of small geographically areas. This section will investigate both the optimal result and binary result for different datasets, when clusters are computed with the different ex-

amined parameters.

D10C

the optimal covered testing data for D10C can be seen in *Figure 4.23* for RDGC clusters. RDGC clusters are computed on the training set with km distances between 5 and 50, core-point threshold of 3000, 2600, 1800, 1400, 1000, 700, minimum points of 3 and minimum length of 10. RDGC clusters can accept between 24% and 68% of testing data, while most RDGC clusters will accept between 40% to 55% depending on the parameters.

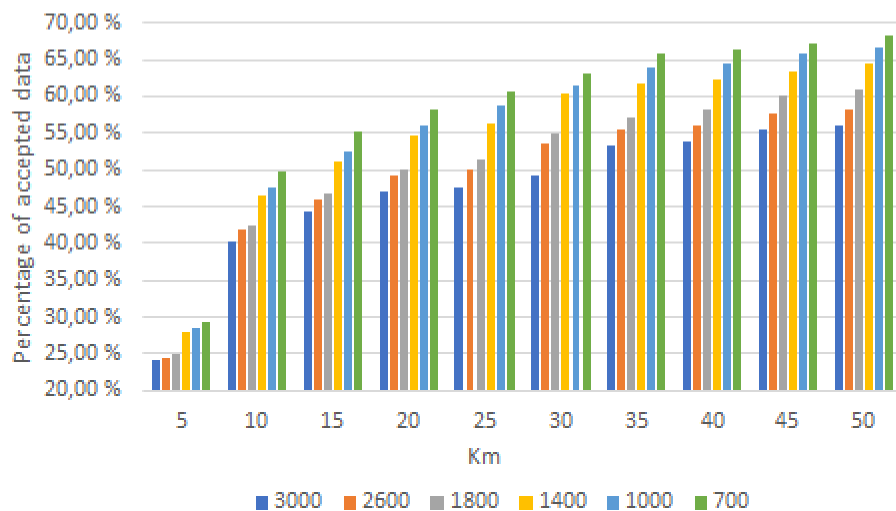

Figure 4.23: *This figure shows the optimal amount of accepted D10C testing data when RDGC clusters are created on the training phase. RDGC clusters are computed on the D10C training set with minimum required length of 10, minimum required point of 3, km distances between 5 and 50 and core point thresholds 3000, 2600, 1800, 1400. All traffic from D10C testing set, which exist location wise in a calculated cluster, are counted as a part of the optimal amount.*

The most differences seen between DGC and RDGC are with very low km distances of 5. There is then around 18 to 29 percentage points differences between DGC and RDGC. RDGC clusters will have a problem with expanding to a minimum of 10 points with very low km distances. Hence, the high differences between DGC and RDGC. For the remaining km distances between DGC and RDGC there is an average distance of 7 percentage points in covered testing data.

The perhaps most important point is to see the differences between an op-

timal solution and a non-optimal solution. In a non-optimal solution, a packets location can't be identified. Therefore, only networks which appear in the training phase, and is a part of a cluster, can be accepted. RDGC clusters are created with the same parameters above, but with a non-optimal acceptance of testing data. The amount of covered testing data for a non-optimal solution can be seen in *Figure 4.24*.

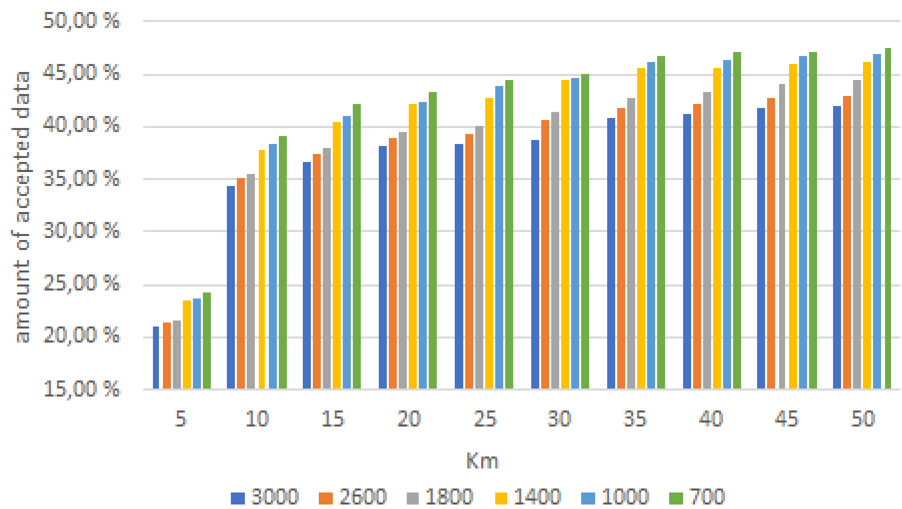

Figure 4.24: *This figure shows the amount of accepted data from D10C testing set which was seen in a cluster. Clusters are computed on a D10C training set with a minimum required length of 10, minimum required points of 3, km distances between 5 to 50 and core point thresholds 3000, 2600, 1800, 1400, 1000 and 700. Traffic from D10C testing set, is only accepted as a part of a cluster, if the 24-bit network is also seen in this cluster.*

An optimal solution are able to accept between 3 to 20 percentage points more than a non-optimal solution. At an average, an optimal cluster are able to accept 12 percentage points more than a non-optimal cluster. This means an optimal solution can, depending on the parameters, accept a high degree of new data which has not been seen in the training phase. Moreover, considering a minimum length of 2, most RDGC clusters have a small differences of 1 to 4 percentage less than DGC.

D11A

RDGC clusters created on D11A training set with km distances between 5 and 50, core-point threshold of 2600, 1800, 1400, 1000, 700, minimum length of 10 and minimum required points of 3, before the amount of optimal accepted

testing data is counted, can be seen in *Figure 4.25*. The optimal amount of accepted testing data ranges between 70% to 84%. Most RDGC clusters on D11A have little differences with lower thresholds, and it is mostly different km distances which affects the covered data pattern. RDGC clusters with a minimum length of 10, covers 8 to 15 percentage points less than DGC. Clusters with low km distances of 5 to 25 will cover around 13 to 15 percentage points less than DGC. Higher km distances will cover around 8 to 9 percentage points less than DGC.

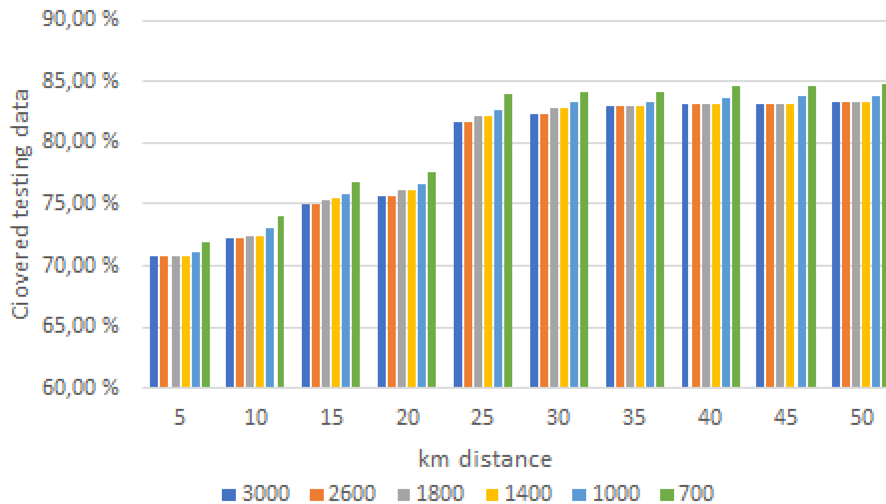

Figure 4.25: *This figure shows the optimal amount of accepted D11A testing data when RDGC clusters are created on the training phase. RDGC clusters are computed on the D11A training set with minimum required length of 10, minimum required point of 3, km distances between 5 and 50 and core point thresholds 3000, 2600, 1800, 1400. All traffic from D11A testing set, which exist location wise in a calculated cluster, are counted as a part of the optimal amount.*

When RDGC clusters are computed with the same parameters as above, but with only a non-optimal acceptance, then only networks which appear in the training phase and is a part of a cluster, is accepted. The acceptance rate diminish with 3 to 8 percentage points compared to an optimal result. Furthermore, most clusters with low km distance, lose 3 to 4 percentage points in a non-optimal solution. This RDGC calculation, with a minimum length of 10 and with a non-optimal acceptance rate for the testing set, can be seen in *Figure 4.26*.

D11A is a highly localized pattern and strict requirements might not be the

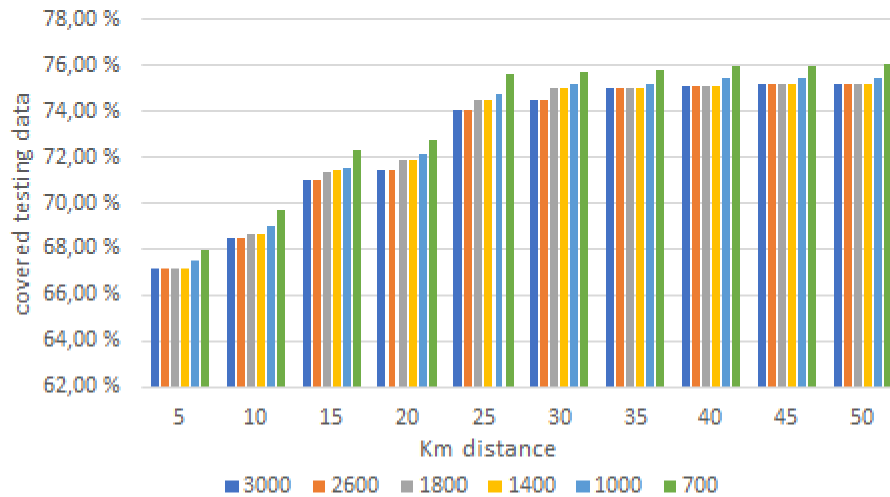

Figure 4.26: *This figure shows the non-optimal amount of accepted D11A testing data when RDGC clusters are created on the training phase. Clusters are computed with a minimum required length of 10, minimum required points of 3, km distances between 5 to 50 and core point thresholds 3000, 2600, 1800, 1400, 1000 and 700. Traffic from D11A testing set, is only accepted as a part of a cluster, if the 24-bit network is also seen in this cluster.*

best solution. Instead of applying a high minimum length, we can only remove singular core-points. A RDGC clustering algorithm which removes singular core points and have a minimum required points of 3, before the amount of optimal covered testing data is counted, have an acceptance rate of around 71% to 89%. RDGC clusters with km distance of 10 to 20 have an acceptance rate of around 77% to 83%.

RDGC clusters which removes singular core-points and have a minimum points of 3 will still lose around 7 to 15 percentage points, against a DGC clustering algorithm. However, removing singular core-points makes RDGC not diminishing a high degree of data with low km distances. While RDGC clustering with a minimum length of 10, removes 13 to 15 percentage points for km distances between 10 and 25, RDGC clustering with a minimum length of 2, only removes around 8 percentage points. This is around half the amount with a high minimum length and shows that a lower minimum length, is better suited to represent a localized data pattern.

D11B

RDGC clustering on D11B, performs even worse than on previous datasets, even when RDGC clusters only are computed with a minimum length of 2. A RDGC

cluster calculation on D11B with a minimum length of 2, minimum points of 3, core-points of 3000, 2600, 1800, 1400, 700 and km distances between 5 and 50, can be seen in *Figure 4.27*. Here, the amount of optimal accepted D11A testing data that fits within a cluster location wise, is counted.

The amount of accepted testing data ranges between 28% to 82%. However, most clusters accept only between 30% to 40%. First, when the km distances is 50, 70% to 80%, testing data is accepted. D11B seems to have a high degree of location points which are singular points with no possibility of expanding. Singular points can't therefore not support the notion of minimum points or minimum length.

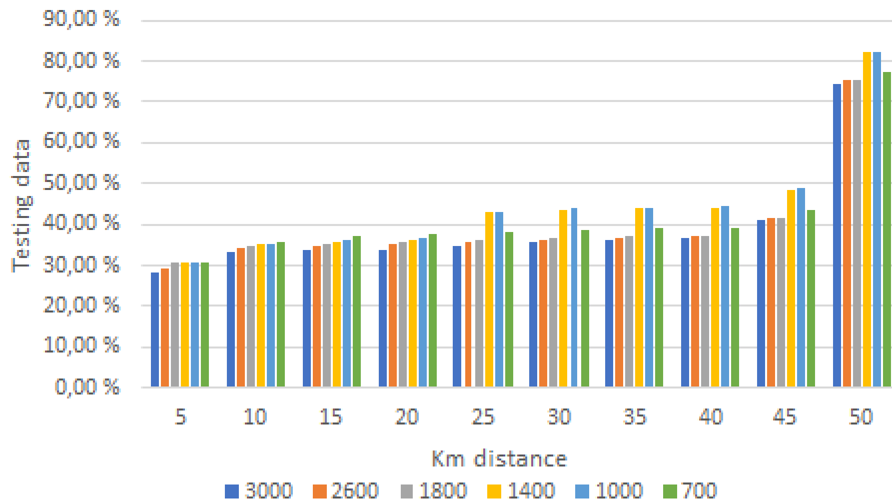

Figure 4.27: *This figure shows the optimal amount of accepted D11B testing data when RDGC clusters are created on the training phase. RDGC clusters are computed on the D11B training set with minimum required length of 2, minimum required point of 3, km distances between 5 and 50 and core point thresholds 3000, 2600, 1800, 1400. All traffic from D11B testing set, which exist location wise in a calculated cluster, are counted as a part of the optimal amount.*

DGC clusters without these extra constraints, cover over 40 percentage points more than RDGC. Moreover, RDGC clusters between an optimal and a non-optimal solution only changes with around 2 to 4 percentage points. D11B is a great example of the highest weaknesses with RDGC. RDGC builds on the principle that more location points in a certain area, automatically confirms the legitimacy of points or clusters in this area. However, the hypothesis lacks the ability to differ between location points with a high amount of request and location points with a low amount of request. Therefore, even if a location point

have 70 000 seen requests, the location point is not seen as legitimate if it can't expand further. This is a massive weakness as we would preferably like to accept location points with a high amount of requests, regardless of the points ability to expand.

4.5 Classifying new objects

These two clustering algorithms depends on the ability that locations of new traffic can be located. In a normal scenario, we could check if any new unknown packet would fit in any of the predefined clusters. However, since we have defined clusters based on location and not on already known attributes, we are unable to calculate the location without using some external resource. Under an attack it would be indefeasible to use a solution that would look up all incoming IP addresses to see if the IP address fits in any of the predefined clusters. A lookup can fast take a couple of seconds to complete and the amount of processing power required to process several thousands packets pr second would be too massive for the solution to scale properly.

To avoid this issue, it would be preferable to either identify some attributes that can tell us something about the location, or identify a pattern within each cluster that separates itself from other clusters and from data that is not within a cluster. However, opposite of identifying this pattern based on location, the pattern should be identified by the already known attributes. We can take several approaches to solve this issue and this section will investigate two approaches in how to best define a new object into a cluster.

4.5.1 Naive Bayes

As mentioned from the background chapter, *Naive Bayes* is not a singular algorithm, but rather all algorithms based on applying the *Bayes theorem*. The *Bayes theorem* which tells the probability of an event to occur based on prior knowledge of this event, is mathematically stated as the equation below. The *theorem* is built on the principle that every feature considered is independent from all other available features. This means each feature will independently heighten the chance for the object to be in a certain class. However, the relationship between the features will not heighten or diminish the chances for the object to be in a certain class.

$$P(c_i|A) = \frac{P(A|c_1) * P(C_1)}{P(A)} \tag{4.1}$$

The equation above states that $P(A|c_1)$ is the evidence that supports the notion of this event to be in a certain class. Since there can be several different evidence to consider, we can further expand the equation as below.

$$P(c_i|A) = \frac{P(a_1|c_i) \cdot P(a_2|c_i), ..., P(a_n|c_i) * P(C_1)}{P(A)} \tag{4.2}$$

The *Bayes theorem* can be employed to try and identify if a packet is likely to belong to a cluster based on its own IP address, formatted as a binary address. A straight *Naive Bayes* implementation, which calculates what cluster a new incoming packet is likely to belong, based on previously seen information, might case problems for several reason; Firstly, large clusters will have problems with identifying bits that are more common than other bits. In an ideal scenario, nearly all of the seen IP addresses will share some common bits. However, for large clusters that expand over many location points or ISPs, this will often not be the case. Secondly, large networks or locations will often override small locations. Even though small locations might fit better in a small cluster. Thirdly, small points within a large cluster will often be ignored.

To evaluate how well a directly implemented *Naive Bayes* approach performs, a *Bayesian probability* is calculated for each cluster, where each bit is computed independently from every other bit. A new IP address checks against all clusters to find the highest possible probability that this IP address are able to get. As an example, if we have several classes or clusters that amount to a total of *5000* requests, where one cluster Y have *4* seen networks in a simplified form as *01110*, *01111*, *10100* and *10110*. These networks then have a frequency of *500*, *495*, *300* and *60*. We can calculate a new packet with network address *11010* probability of being a part of cluster Y as below:

$$P(c_i|11110) = \frac{P(1|c_1) * P(1|c_1) * P(0|c_1) * P(1|c_1) * P(0|c_1) * P(C_1)}{P(11110)} \tag{4.3}$$

The equation can further be expanded or rewritten with the different considered probabilities in mind:

$$P(c_i|11110) = \frac{\frac{360}{1355} * \frac{995}{1355} * \frac{1355}{1355} * \frac{1055}{1355} * \frac{860}{1355} * \frac{1355}{5000}}{P(11110)} = \frac{0,025}{P(11110)} \tag{4.4}$$

We also need to factor in the probability of this certain combination of events, to occur independently from the class. The value *0,025* gained from the algorithm only gives the likelihood of this pattern to be in this particular cluster, and is not the same as the probability. Therefore, to normalize the likelihood we need to divide *0,025* with the probability of *P(11110)* to occur independently of the given class. This can be accomplish by dividing *0,025* with the probability for *P(11110)* to occur in class or cluster $c_i + c_{i+1}+, ..., +c_n$ as seen below:

$$\frac{0,025}{P(11110|c_i) + P(11110|c_{i+1}) + P(11110|c_{i+2}, ..., P(11110|c_n)} \quad (4.5)$$

Figure 4.28 Shows one of the probability distribution gained by *Naive Bayes*. The probability distribution is calculated on DGC clustering. DGC clusters are created on D10C with core-point threshold 1000 and km distance 5 . It's interesting to note, when *Naive Bayes* is calculated on clusters created with a small *km* distance, *Naive Bayes* is still incapable of achieving a high likelihood or probability for most of the seen request. Only *16,48%* of the seen requests have a computed probability over *80%*. In comparison *40,45%* of the seen requests have a probability under *30%*. In an ideal scenario, a calculated cluster with maximum distance of *5* km and core point threshold of *1000* should be able to cover around *55,84%* of D10C testing data. To achieve this goal, every request with a probability over *30%* needs to be accepted as legitimate traffic.

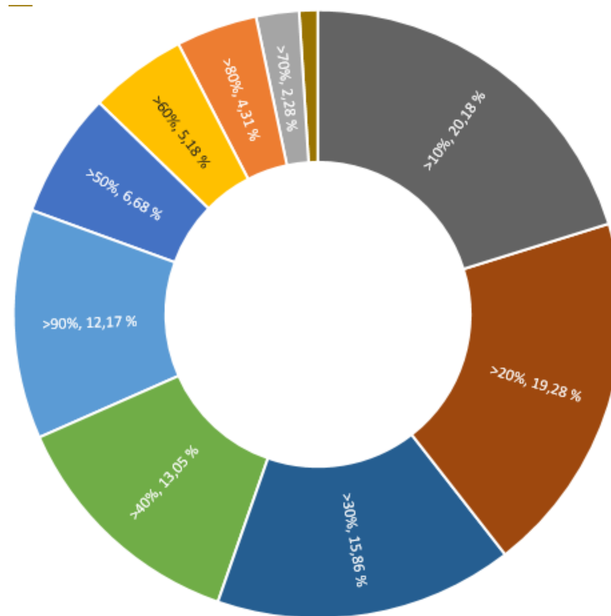

Figure 4.28: *Shows the probability distribution in D10C testing set with a Naive Bayes calculation based on DGC clusters with a maximum allowed distance of 5 km and a core point threshold of 1000. For each request, the highest computed probability is counted as the most likely for the object to belong to.*

Even though smaller clusters performs slightly better than larger clusters in a Bayesian probability model, a straight Bayesian implementation is unable

to perform close to an ideal scenario and is insufficient to represent the data pattern.

4.5.2 Reverse search

It is hard to estimate a pattern on each cluster which can identify if any unknown source address belong to a cluster. This can largely be contributed to the fact that networks which lay close geographically, don't need to lay close in the address range. Therefore, identifying a pattern based on geographical clusters, will be harder, than identifying a pattern for clustering in the address range. Using Bayesian calculation on geographical clusters will therefore in most cases not work.

Instead of using machine learning to identify a pattern on each cluster, we can simply use the found clusters to identify more networks. This can be accomplished by doing a reverse search on a geographical IP database. Table 4.1, shows the amount of accepted networks in DGC and RDGC clustering, against D11A, when a reverse search is used to populate the database with more networks. Clusters are created with a core-point threshold of 3000 and a km distance between 5 and 20. Moreover, RDGC clusters have an added constraint of a minimum length of 2 and a minimum points of 3.

Km distance	DGC	RDGC
5	608117	107788
10	683489	279804
15	927265	360763
20	941675	376277

Table 4.1: *Shows the amount of accepted networks for DGC and RDGC clustering, on D11A, when reverse search is used to populate the clusters with more networks.Clusters are created with a core-point threshold of 3000 and km distances between 5 and 20. RDGC clusters have an added constraint of a minimum length of 2 and a minimum points of 3.*

Reverse search is able to find new networks for both RDGC and DGC. Since DGC don't have any limitation for the cluster calculation, a heavier amount of networks are found. Although DGC clusters are able to cover 7 percentage points more testing data than RDGC, the differences between covered data in an optimal and non-optimal solution is around 7 percentage points for both clustering algorithms. This means RDGC clustering have managed to find the most relevant location points to accept, which resulted in the heightening of 7 percentage points. RDGC are furthermore able to accomplish this by only accepting a fraction of networks that DGC accepts. The amount of accepted testing data between an optimal solution and a solution that populates the database can be seen in Table 4.2 for RDGC cluster calculated on D11A. RDGC

clusters are calculated with a core-point threshold of 3000, minimum length of 2, minimum points of 3 and km distances between 5 and 20.

Km distance	Optimal RDGC	Populated RDGC
5	355423	356867
10	388904	389374
15	403089	402627
20	406477	405458

Table 4.2: *Shows the amount of accepted data between a optimal solution and solution which looks up new networks. RDGC clusters are created on D11A with core-point threshold of 3000, minimum length of 2, minimum points of 3 and km distances between 5 and 20.*

A solution that populates the database performs nearly identical with the amount of accepted testing data of an ideal solution. This data can be used to decide if packets are legitimate or illegitimate.

Chapter 5

Approach

This chapter will be based on the 3 proposed algorithms from last chapter, I will discuss different features which will be used to test the proposed algorithms. These features include; The virtual environment setup, the external tools in use and how the mitigation technique is built up and which data structures in use.

5.1 Testbed environment

The virtual environment is used to simulate a DDoS attack and consist of several hosts in a closed testbed environment. The architecture of the simulated environment can be seen in *Figure 5.1* and contains 4 hosts in the 11.11.11.0/24 address range. The servers have different responsibilities in the execution of a real life simulation, which include gathering statistics, simulating a DDoS attack, mitigate a DDoS attack and answering web-requests which are accepted by a mitigation solution.

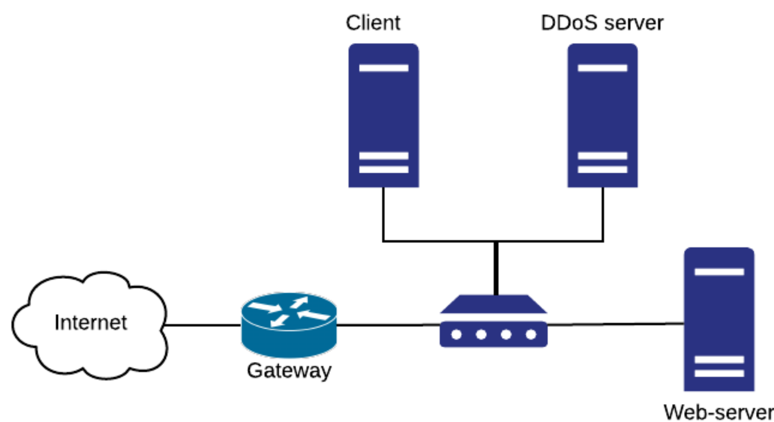

Figure 5.1: *This figure shows the architecture of the testbed environment.*

DDoS server

The *DDoS server*, contains 4 Intel Xeon 5130 processors running at a max speed of 3600 MHz. The server runs Linux Ubuntu 16.04, x86_64 architecture with version 4.4.0-66-generic. The DDoS server is in charge of simulating both HTTP flood and normal traffic flow. To ensure that simulated requests with different source addresses are not sent to the real world. A default route on the web-server sends all return traffic to the DDoS server. Moreover, the DDoS server has routing turned off, so requests are not sent out to the real world.

Client host

The client server runs Linux Ubuntu version 4.4.0-36-generic with x86_64 architecture and consist of 2 CPUs with a max speed of 5200 MHz. The client host sends requests to the web server to measure different statistics. These statistics include the round trip time for legitimate requests and the drop rate for requests that should be accepted, but which might be dropped because the web-server are not able to handle the amount of incoming requests.

Web-server

The web-server also contains 4 Intel Xeon 5130 processors running at a max speed of 3600 MHz. The web-server runs Linux Ubuntu, 64 bit architecture with version 4.4.0-64-generic. In our simulated environment, the web-server is responsible for two tasks; answering legitimate HTTP request and mitigating a possible attack. The mitigation technique makes a decision on each incoming packet independently. Packets which are accepted will have access to the web-service where requests are completed as normal.

5.2 External tools

To be able to simulate this environment, the mitigation techniques and the subsequently attacks, several tools are used. This section goes through the different tools and libraries which are used to test the proposed algorithms.

5.2.1 Repache

Repache or Apache Traffic Replay Generator is a tool that are able to use the Apache log-files to generate HTTP requests based on recorded IP addresses. Repache is used on the DDoS server to simulate real life connections to the web-server and the tool is able to establish TCP connections, request data and tear down different connections. Since Repache use previously recorded data, the tool is able to simulate a system under realistic circumstances

5.2.2 BoNeSi

BoNeSi is an open source botnet simulator for simulating TCP based HTTP-GET flood against a web-server. BoNeSi are able to establish thousands of HTTP connections from different IP addresses from just a single server. BoNeSi are used on the DDoS server and takes four arguments; The rate per second of traffic that BoNeSi should send, a file which contains the IP addresses that BoNeSi pretends is the source, the destination address, port number, and the interface which BoNeSi listens to.

5.2.3 Python3

The mitigation techniques and solutions on the web-server are coded in Python. Python is a high-level programming language, with focus on readability and to give an easy way to express concepts in fewer lines of code. As python is a high level language, mitigation techniques in python is not the best solution. This is mainly because code written in a high level language, like python, is not optimized compared to a low level language.

5.2.4 Nfqueue

The mitigation techniques on the web-server needs to have access to each packet independently to make a choice whether the packet should be accepted or dropped. Since Python is a high level problem solving language it is not possible to get access to these packets directly, as Python runs in user space and don't have access to kernel space, where packets are processed.

Instead, Nfqueue, which is a C extension module, delegates the decisions on packets to userspace software. Nfqueue can therefore be used to simulate Python handling the packets. Nfqueue links against LibnetFilter queue, which provides a way of altering packets that have been queued by the kernel packet filter. When Nfqueue provides access to these packets from the kernelspace to the userspace, a mark can be set on each packet to be dropped or accepted. Nfqueue provide access to packets which have been matched by a iptables rule in Linux. In our mitigation technique, all packets which have a destination port of 80 is sent to Nfqueue, Nfqueue then further sends the data to our mitigation technique which decides the fate of each packet.

5.2.5 Pybloom

Pybloom is a python implementation of the bloom filter probabilistic data structure. The bloom filter takes an error rate and the amount of elements in the bloom filter. Pybloom can be used to add elements and check if an element exist in the data structure.

5.3 Data structures

Because we have proposed 3 different algorithms, it would be beneficial to test several solutions or data structures to use in mitigating a possible attack. From last chapter, we get 24 bit networks which should be accepted. We can store these networks several ways before a solution makes a decision on each independent packet. This section will investigate two different methods we can use to store and decide the fate of any new packet.

5.3.1 Tree structure

Using arrays or linked lists to store gained data from different algorithms can pose several risks with regards to the efficiency. Since both arrays and lists are linear data structures, it might be necessary to search the entire structure to look for an element. At worst, a search for element x, will go through N elements, where N is the amount of elements in the set. Therefore, instead of using linear structures to store data, we can expand the notion of lists, which only points to 1 object, to lists which points to several relations.

This structure, known as a tree structure, is a hierarchical data structure and can consist of several nodes or data items which have direct or indirect relations to new nodes. A tree structure exist of a start node, known as the root node, which further points to new nodes. A tree structure contains no cycles where nodes relates back to itself. A nodes relation to new nodes is known as the nodes children nodes. The bottom of the tree, or nodes that don't have any children, are further known as leaf nodes.

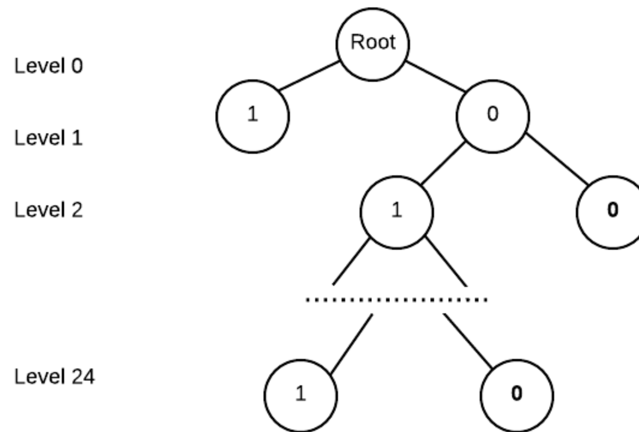

Figure 5.2: *This Figure shows the tree structure used for storing found 24-bit networks from AFN.*

Tree structures are efficient data structures which can fast and easily look up

new data. In our algorithms, where we have found 24-bit networks that should be accepted, we can use a tree structure to store the found networks. As seen in Figure 5.2, the data can be stored as 0's and 1's. When a new packet comes in, the packets 24-bit source address will be looked for in the binary tree structure. If for example, a packets source address starts on bit 0, while the tree only contains 24 bit networks which starts on bit 1, the packet can be determined to not exist in the tree on the first node.

Moreover, a tree structure of N levels will, if packets are accepted, have a time complexity of $O(n)$. For packets that are dropped, the best scenario for traversing is $O(1)$, while the worst scenario is $O(n-1)$. Since the tree structure is an efficient data structure which should quickly determine the fate of any new packet. We can use the tree structure for AFN. AFN will then find frequent 24-bit networks on the training phase, before a tree structure is used to store the found 24-bit networks.

5.3.2 Bloom filter

The remaining algorithms DGC and RDGC will, if reverse search is used, need to store a high amount of networks. It might be beneficial to use a different structure than trees to preserve memory space. Instead of using trees, we can employ a bloom filter. Bloom filters have previously been used within anomaly detection[78] and is a space efficient probabilistic structure which use hashing to determine if elements is in a list S. If an element x is determined to be in the set S, x is in set S with a known probability. If x is determined to not be in the set, x is with a 100% certainty not in the set. This means that bloom filters don't allow false negatives but allow false positives. Higher false positive will often lead to more space savings. However, the drawbacks of false positives are often out-weight by the saved space[76][19].

Three parameters define the bloom filter; The length of the bloom filter, noted as m bits, number of hash functions known as n, and number of inserted elements. A bloom will start with all m bits set to 0. As elements are added to the filter, each element will be checked against k hash functions which will results in k positions that should be set to 1. To determine if an element x is in the bloom filter the same K hash functions are executed. If all k positions points to *1* the element, is with a certain probability, determined to be in the set[76]. An illustration of this data structure can be seen in *Figure 5.3*.

Bloom filters have several weaknesses which include the lookup time for any element is highly dependent on the false positive rate. Lower false positives will often require a higher amount of hash functions, which again mean more memory accesses[71]. Hash functions are a demanding element to calculate and the time complexity of a bloom filter is not based on the amount of elements in the set, but the number of hash functions. The time complexity of a bloom filter is noted as O(k), where k is the number of hash functions. Mitzenmacher M. considered this issue, by increasing the amounts of bits m, it was possible to

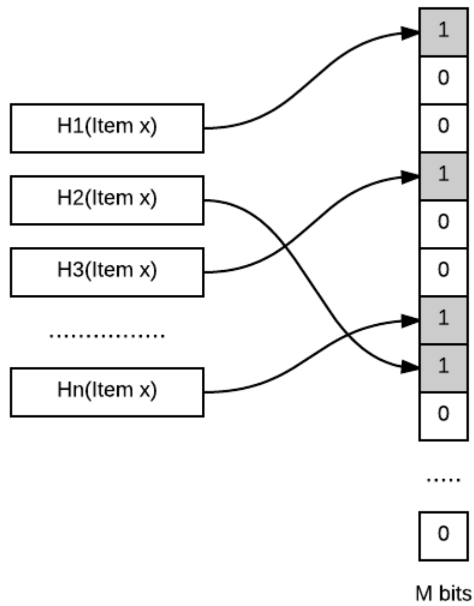

Figure 5.3: *This figure shows the structure of a bloom filter with N amount of hash functions and with M available bits data can be stored in.*

reduce the number of hash functions, and therefore the time complexity of any new lookup[55].

In our mitigation technique we can employ a bloom filter on both DGC and RDGC. How fast the bloom filter are able to determine the fate of any new packet then depends widely on the error rate. Instead of constructing an own Bloomfilter, we can employ the *Pybloom* library, which is just a python implementation of a bloom filter.

Chapter 6

Results

This chapter will, based on the data mining from last section, investigate, to a certain degree, if the proposed algorithms are able to deal with mitigating DDoS attacks. Both how fast the solutions employing the algorithms can decide the fate of any new packet, as well as how narrowed down the data pattern is to prevent the most illegitimate traffic, are calculated. Furthermore, the proposed solutions, which employ the information from the different algorithms, will be tested against a varies amount of load from different botnets. The calculated data pattern from datasets *D10* and *D11*, will be tested on a range of simulated botnets, ranging it both size and where bots are situated geographically.

6.1 Load test

We should calculate the amount of load, both the web-servers, and a solution that tries to filter out packets on the web-server are able to handle. The load, that both the web.server and other proposed solutions will handle, will vary based on configuration and how much traffic a solution have decided to accept. For example, with more accepted traffic, different solutions, like a tree structure, used with *AFN*, might use longer time to decide which traffic that can't be accepted, as some illegitimate traffic needs to travel longer down the tree before finding out that it can't be accepted.

6.1.1 Load test on Apache web-server

As seen in *Figure 6.1*, with a default Apache configuration and a time-out value of *500* milliseconds, Apache are able to handle *500* request per second, before beginning to drop or queue request. At this point, with a request per second of *600* or above, Apache, still doesn't use *100%* of the CPU. As Apache, by default, is only allowing *150* concurrent connections at any given time moment. This means, that any remaining request are queued for an available slot to open.

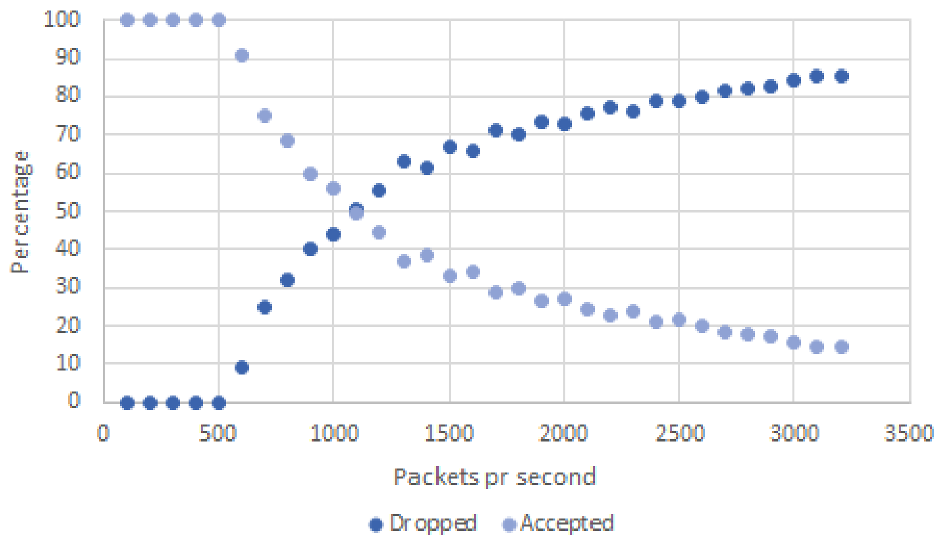

Figure 6.1: *Shows how many requests per second, noted by the percentage, Apache is able to handle. If a request have not gotten a response in 500 milliseconds, the request is assumed dropped by the web-server. Bonesi is used to simulate the amount of request per second, before a singular process on a different host starts a web request against the web-server every second for 20 minutes.*

Even though the acceptance rate of accepted requests decreases when the amount of packets per second against the web-server increases, users might still be able to get a response, if they wait longer than 500 milliseconds. As seen in *Figure 6.2*, when an average request time for all requests is calculated with the same data, as from *Figure 6.1*, we can draw a trend-line, with the function $Y = 0,0615x^2 + 1.5071x$. This function can estimate the amount of seconds that it will take to receive a response from the web-server when the server is experiences a certain rate of packets per second. Y is here the average time in seconds and X is the current amount of packets per second. However, in an ideal scenario and in the ongoing experiments with traffic load against the server, we should try not to let in over 500 request every second if we don't want users to get heavily affected by the DDoS attack.

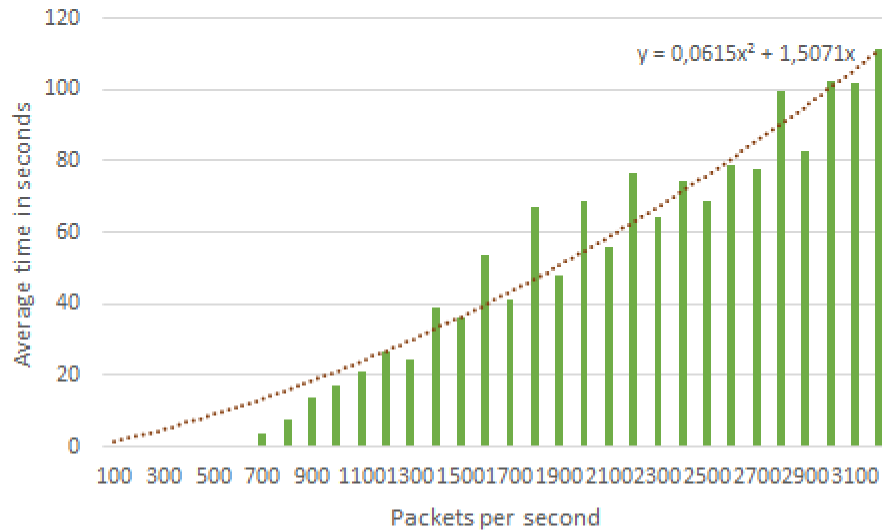

Figure 6.2: *Shows the average time, in seconds, a request takes to complete with different packet rates against the Apache web-server. Since Apache, by default, only can handle 150 concurrent connections at any given moment. Requests will eventually need to wait for an open slot to be served by the web-server.*

6.1.2 Load test on proposed structure

As discussed in the approach chapter, to mitigate a DDoS attack, both a tree structure containing binary networks and a bloom filter, which use hashing, to decide if a network should be accepted, are used. This sub-section will therefore investigate, how fast these solutions can decide the fate of any new packet as well as how much load these proposed solutions are able to handle before packets are being queued and dropped.

Tree structure

As mentioned, the binary tree structure, used by *AFN*, contains nodes representing either a zero or one. Every new packet then need to travel down the binary tree to check if the binary representation of their 24 bit network exist in the tree. If it does, the packet is accepted, if not, the packet is dropped by the solution.

If any new legitimate packet arrive and is handled by a thread immediately, it will on average take *39,55* microseconds for the packet to be processed and accepted. However, if the packet on the other hand, doesn't exist in the tree and is rejected at the first level, the average processing time is *22,61* microseconds. This means, that it takes on average $39,55 - 22,61 = 16,94$ microseconds to traverse the tree. The main reason for the processing time to not diminish below

this, results from the fact that the solution is coded in python and operates in the user mode. For the solution to then decide the fate of any new packet, the source address, encoded as a bytes array, needs to be translated into a *24* bit binary address, before the packet can traverse the tree. This calculating, takes on average *8,77* microseconds, which is a little under *1/3* of the time for a packet that is dropped right away and a little under *1/5* of the time for a packet that needs to traverse the entire tree.

How many packets per second, a solution that uses a binary tree are able to handle, will not only vary based on available resources at the targeted system, but also based on the amount of *24-bit* networks represented in the tree and the given attack pattern. In the best scenario, a solution would drop all illegitimate traffic by the first checked bit or first node, while at the worst scenario, illegitimate packets, if not accepted, are dropped at node *23*. *Figure 6.3* shows the amount of overall CPU usage, with different simulated packets per second, for a binary tree, which either drops all illegitimate traffic at the first or at the last node. A tree which needs to drop illegitimate traffic at the last node, will at worst increase the CPU usage by around *10* percentage points against a binary tree which drops all illegitimate traffic at the first node.

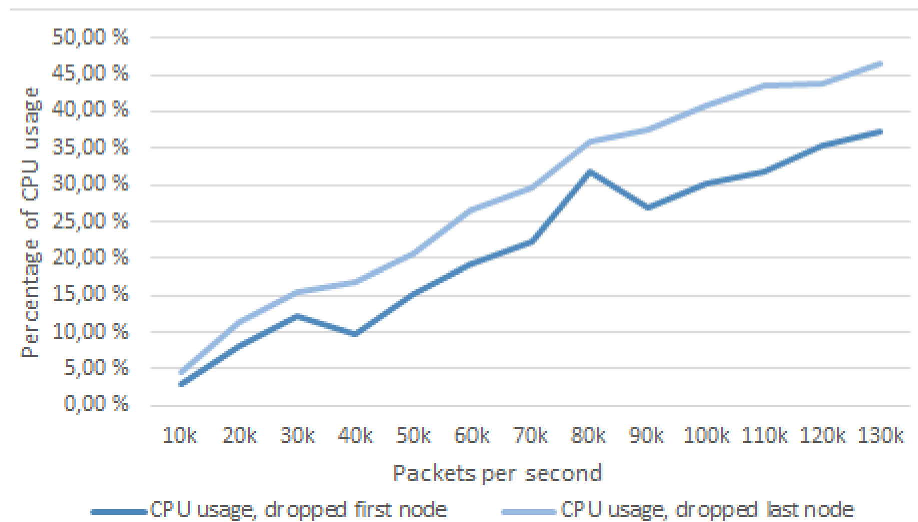

Figure 6.3: *Shows the amount of CPU usage for a binary tree solutions, at our targeted system, which drops all illegitimate traffic at either the first or last node. The binary tree are able to process from 10 000 to 130 000 packets per second without a huge issue.*

As seen in *Figure 6.3*, our simulation of botnet traffic, with BoNeSi, against our targeted system, are not able to bring down the binary tree solution. As the tree, regardless of the length a packet needs to travel, are still efficiently

able to decide the fate of any new packet. Moreover, even though the average number of elements in the queue, as seen in *Figure 6.4*, increase with a higher amount of packets per second, the queue never reach a high amount of packets. Therefore, the attack effect on new incoming requests, is minimal to none. As seen in *Figure 6.5*, in a binary tree solution which drops all packets immediately on the first node, does not notices an increase in RRT for a legitimate request. Moreover, with a huge amount of nodes, that might force all packets to go down to level 23 in the tree, the amount of time a request needs to wait will only increase by a couple of milliseconds.

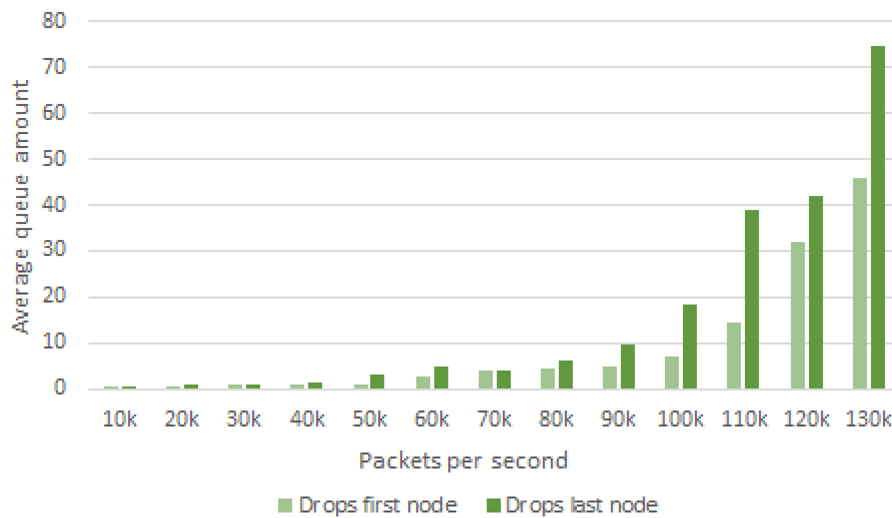

Figure 6.4: *Shows the average amount of elements in the queue, for a binary tree solution, when all processed packets, which should not be accepted, either are dropped at the first node or at the last node in the tree. The queue will increase with some amounts, however, as seen in figure 1.5, the effects are minimal.*

Therefore, in our constrained environment, it is not possible to bring down the web server with HTTP flood, as the attacking server are only able to send 130 000 request per second. Although a higher packet rate might increase the likelihood of the server going down, the proposed solution of a binary tree structure can also be heavily improved by coding directly into the kernel, and avoiding the need for Nfqueue to send packets to user space. Therefore, with later botnet simulation and real life datasets, the binary tree solution, needs to at least, accept around 500-600 illegitimate packets before being able to affect the legitimate traffic more than necessary.

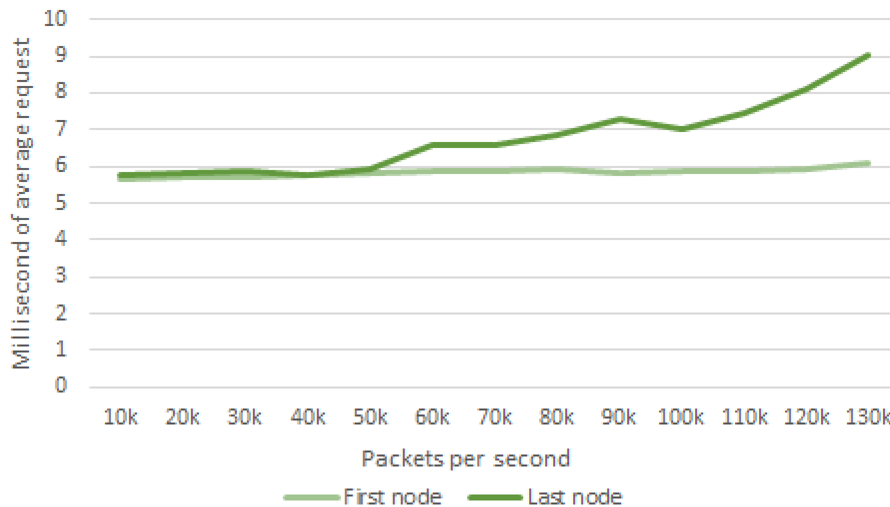

Figure 6.5: *Shows the average amount of time, in milliseconds, a legitimate request takes with different packet rates of illegitimate traffic. The average time for a legitimate request will change depending on the amount of illegitimate traffic and how much of the illegitimate traffic which are dropped at the first and last node in the binary tree.*

Bloom filter

A bloom filter is a space-efficient probabilistic data structure, which is designed to quickly and memory efficiently decide if a new and unknown element is preset in a set of elements. The price paid for this efficiency are false positives in the set. This means that if an element x, based on a hash function is determined to not be in the set, the element is with a 100% certainty not in the given set. However, on the other hand, if an element y is determined, based on a hash function, to be in a set, element y is likely to exist in the set. However, the element might also, based on an error rate, not be in the set and therefore be a false positive.

The average time a request will use to be processed with a bloom filter differs between different error rates and the amount of networks in the bloom filter. On average, if a thread processes the request immediately, a request will use around 50 to 60 microseconds. As seen in *Figure 6.6*, the different error rates of 0,001, 0,0001 and 0,00001 will increase the processed time with around 2 to 8 microseconds. For example, an error rate of 0,0001 will make a request without considering other processing aspects, take around 27,62 microseconds to be processed.

The bloom filter is slight more inefficient, regardless of the tree structures ability to accept a packet at the first or last node, in deciding the state of any

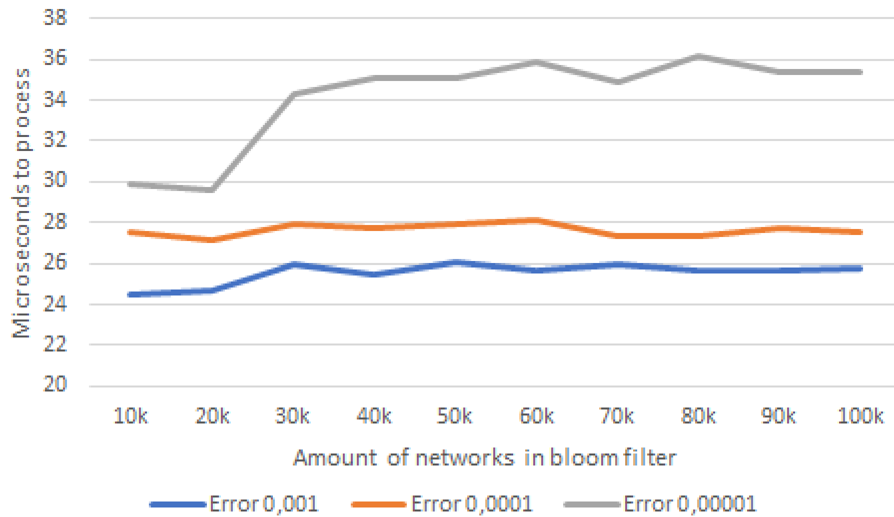

Figure 6.6: *Shows the different processing times for a bloom filter with a certain error rate against a certain amount of networks in the set. The processing time measured, is the time from the packet is checked against the bloom filter and not the entire processing time. As the entire processing time would also include converting the IP address to binary and dropping or accepting an incoming packet.*

new element than a tree structure. However, a bloom filter use considerable less space than a tree structure and with subsequently 10 000, 20 000, 50 000, 100 000, 150 000 and 200 000 networks that need to be accepted, a tree structure will use around 24, 28, 36, 49, 56 and 68 MiB, while a bloom filter will use around 13 MiB regardless of the amount of networks which need to be accepted.

Moreover, as it takes longer time for a packet to be processed, both the average queue amount and the overall CPU usage is higher than a tree structure. As seen in *Figure 6.7*, for a bloom filter with an 0,0001 error rate, the average CPU usage, when the filter needs to process 130 000 packets per seconds, is 55,76%. This is an increase of 9 to 46 percentage points from a tree structure. As a tree structure would use 37,26% when discarding 130 000 packets at the first node and 46,54% when discarding 130 000 packets at the last node in the tree. However, the bloom filter is still a rather efficient structure and a botnet, in our simulated environment, will, as also seen in a tree structure, not single handily be able to bring the web-server down by simply trying to overload the proposed solutions with the amount of packets that needs to be processed.

The average amount of queue elements in Nfqueue which are waiting to be processed, also increase from a tree structure. At best, when traffic is dropped

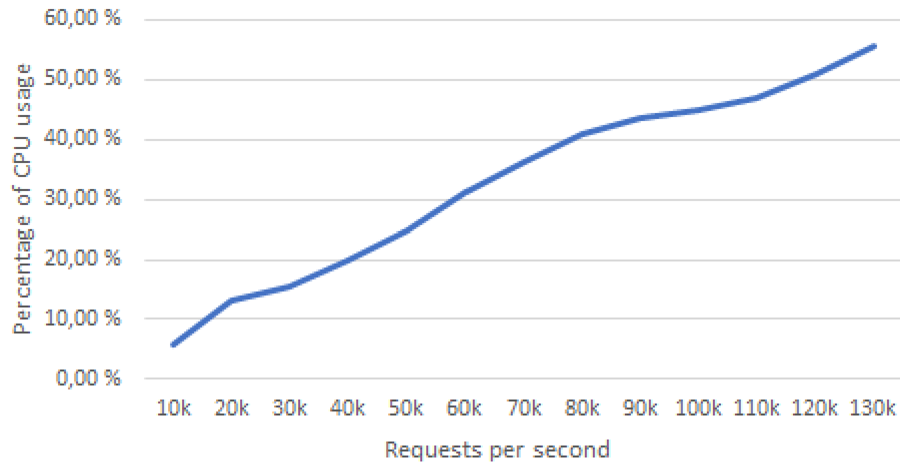

Figure 6.7: *Shows, for a bloom filter with an 0,0001 error rate, the average percentage of CPU usage, when the bloom filter needs to process from 10 000 to 130 000 packets per second. Although the bloom filter is less efficient than a tree structure, the bloom filter still manages to process huge amount of packets without a massive issue.*

at the first node, a tree structure, with 130 000 seen packets per second, have on average only 45 elements in the queue. At worst, when all processed packets are dropped at the last node, the average queue consist of 74 elements. When a tree structure is compared against a bloom filter, the average queue elements for 130 0000 packets per second, as seen in *figure 1.8*, is 87. Although this is higher than for a tree structure, the amount of time a request needs to wait, for bloom filter, doesn't increase notably for different packet rates, even though different packet rates affect the number of elements in the queue. For example, with a packet rate of 30 000, 60 000, 90 000 and 120 000 per seconds, the average time for a request is 5,6, 5,9, 5,9 and 6,3 milliseconds, which is well within any possible error rates. Comparing this value against a tree structure, the value is similar to a tree structure which are able to drop all illegitimate traffic at the first node.

A bloom filter produces a certain error rate. Using a bloom filter, in the wider sense, over a tree structure, in a DDoS mitigation technique, means that the trade-of between the amount of false positive and the strength of more available memory, should be heavily investigated. Moreover, as a bloom filter seems to use more CPU usage than a normal tree structure this should as well be considered when deciding to use a bloom filter or a tree structure.

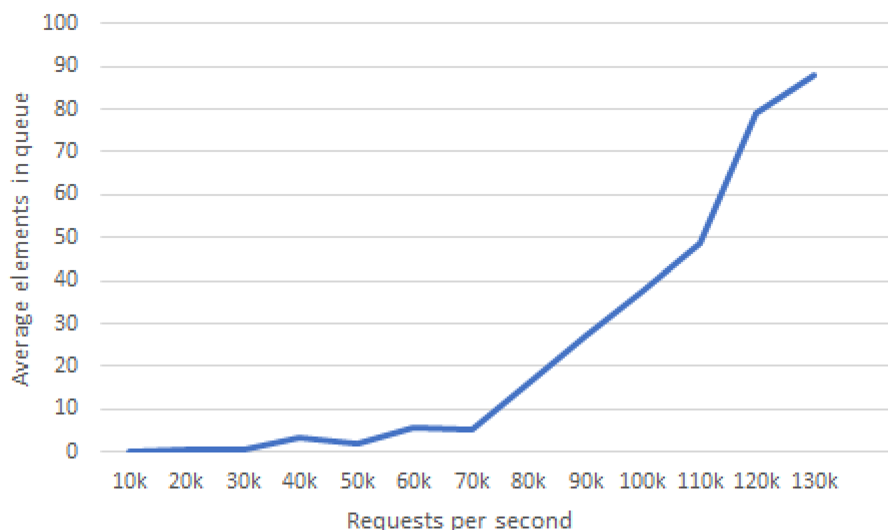

Figure 6.8: *Shows, for a bloom filter with an 0,0001 error rate, the average elements in Nfqueue waiting to be processed when the bloom filter gets from 10 0000 to 130 000 packets per second.*

6.2 Botnets

As discussed from the background chapter, botnets vary in both size and where bots are geographical situated. Botnets will often have areas with a higher and lower densities. It's therefore important to check a possible solution against several botnets with different geographical patterns. This section, will therefore investigate different simulated botnets, to see how much data from each botnet is accepted as legitimate with the use of different algorithms and structures.

6.2.1 B1

The first simulated botnet or B1 contains 100 000 unique IP addresses and are randomized over the IP address space. All proposed algorithms should be able to easily sort out the illegitimate traffic.

AFN

AFN is well equipped to sustain an attack from a B1. At worst 1.17% of B1 are able to gain access to a service if D10A is used to calculate frequent networks. As D10A is the biggest dataset with its 7.1 million, its naturally that this dataset will accept more traffic regardless of threshold. With smaller datasets of D10C, D11A and D11B, which only contains 1 and 1.1 million training data, 0.21%, 0,12% and 0.1% of data from B1 is accepted.

Frequent threshold	D10A	D10B	D10C	D11A	D11B
1	1177	682	206	123	105
5	500	294	89	55	48
10	389	231	65	34	34
20	289	179	50	26	21
50	110	66	22	11	12
100	58	33	8	6	8
250	26	14	5	1	3
500	7	5	1	0	2
1000	5	2	0	0	1

Table 6.1: *This table shows the IP addresses from B1, which is let in if AFN is used to find frequent networks. First, A frequent threshold is used to find 24 bit networks in the training phase which have this amount of seen traffic. Frequent networks are calculated on the datasets D10A, D10B, D10C, D11A and D11B. Then a binary tree structure is applied to enforce that only B1 IP addresses, which can be considered frequent, are accepted. Calculated frequent thresholds are 1, 5, 20, 50 100, 250 and 1000.*

DGC

As seen in Table 6.2, DGC is used to calculate density based geographical clustering with a threshold of 3000 and different km distances. Before the amount of accepted B1 address is counted based on a bloom filter with a 0,0001 error rate. Reverse search is not used on the clusters to populate the database with more networks. DGC clustering perform quite well and achieve similar results as seen for AFN.

However, Although DGC performs quite well, DGC are not able to cover as few addresses as AFN. This primarily stems from the notion that DGC finds more networks in its area, based on density-based clustering, than AFN. AFN is only able to reach a high amount of accepted botnet traffic if the frequent threshold is set low. Therefore, both DGC and RDGC will usually have instances where they accept more botnet traffic than AFN. This is still true, even though reverse search is not used to populate the database with more IP addresses. Moreover, the error rate also have a play in how much bot traffic which are accepted. AFN don't use a bloom filter to accept new data and therefore don't deal with unnecessary false positive rates.

Km distance	D10C	D11A	D11B
5	49	40	40
10	62	43	48
15	75	55	57
20	81	55	56
25	77	56	48
30	86	66	58
35	95	73	59
40	102	63	53
45	99	69	64
50	97	73	62

Table 6.2: *This table shows the IP addresses from B1, which is let in if DGC is used to find and accept data from geographical clusters. First, DGC is used to find geographical clusters in training data from D10C, D11A and D11B. 24-bit addresses in the geographical clusters are used to create a bloom filter, before B1 addresses are checked against this filter. The bloom filter is created with an error rate of 0,0001 and the amount of accepted B1 addresses are then counted. Clusters are created with a core-point threshold of 3000 and km distances between 5 and 50. Reverse search is not used to re-populate the database.*

RDGC

Since DGC was easily able to handle B1 traffic, RDGC, with its stricter requirements, should as well have a low acceptance rate for the same botnet traffic. RDGC is in Table 6.3 used to calculate geographical clusters with core-point thresholds of 3000, km distances between 5 and 50, minimum points of 3 and minimum length of 10. The clusters are calculated on D10A, D10B, D10C, D11A and D11B. A bloom filter with an error rate of 0,0001 is used to store the found networks. Then, B1 traffic is checked against the bloom filter and the amount of covered IP addresses is shown in the table. There is some slight differences between the amount of accepted B1 traffic in DGC and RDGC. However, the differences is not big enough to matter significantly and the small differences is well within the bloom error rate of 0,0001.

Km distance	D10C	D11A	D11B
5	31	35	39
10	45	36	48
15	62	42	49
20	73	52	54
25	68	58	53
30	74	63	53
35	84	71	59
40	89	71	63
45	91	64	71
50	101	73	63

Table 6.3: *This table shows the amount of accepted B1 addresses when RDGC is used to find geographical clusters. First, RDGC is used to find geographical clusters in the training set of D10C, D11A and D11B. Geographical clusters are calculated with a threshold of 3000, minimum points of 3, minimum length of 10 and km distances between 5 and 50. A bloom filter with an error rate of 0,0001 is used to store the found 24 bit networks. Then, the IP addresses from B1 is checked against this filter. The amount of accepted addresses are shown in the table. Reverse search is not used to re-populate the database.*

6.2.2 B2

The second simulated botnet is B2 and contains 118 000 IP addresses. The botnet is situated in Europe and stretches mainly over central Europe. Bots are for the most part situated in Great Britain, France, Spain, Italy, Germany and Poland. Unlike B1 which are a randomized botnet, B2 have a unique geographical pattern and B2 fits more with the way botnets are actually geographical situated than B1.

Datasets which have clusters, or frequent networks, in countries that B2 traffic originate from, will have a harder task to mitigate DDoS attacks, than datasets which don't have traffic from these regions. However, as B2 don't have traffic from Norway, where most of the datasets have a major traffic pattern from, all algorithms should be able to easily mitigate a DDoS attack from this botnet.

AFN

As B2 is placed in central Europe, more traffic is accepted than for B2, where the traffic pattern is randomized. Seen in Table 6.4, when different frequent thresholds are applied for AFN, before the amount of accepted data from B2 is counted. Accepted bots ranges from 0 to 2904. At most 2,46% of botnet traffic is allowed. Compared to AFN on B1, most accepted botnet traffic is 1.17% or 1177 bots. This means, that when we specifically try to create a pattern in Europe, there is a 1 percentage points increase from when we specifically don't simulate a more accurate botnet pattern.

Frequent threshold	D10A	D10B	D10C	D11A	D11B
1	2904	1616	419	379	349
5	872	535	168	114	138
10	658	442	149	38	63
20	480	307	71	24	35
50	171	104	29	4	4
100	70	47	15	0	4
250	41	26	9	0	0
500	18	7	1	0	0
1000	14	5	0	0	0

Table 6.4: *This Table shows the amount of accepted B2 addresses when AFN is used to find frequent 24 bit networks. Frequent networks are found based on threshold 1, 5, 10, 20, 50 100, 250, 500 and 1000 for datasets D10A, D10B, D10C, D11A and D11B. Next, B2 addresses are checked against a 24-bit tree structure with the found 24-bit address. The amount of accepted B3 addresses is counted and shown in the table.*

DGC

Seen in Table 6.5, DGC clusters with a core points threshold of 3000 and km distances between 5 and 50, before the amount of accepted B2 traffic is counted, against a bloom filter with an error rate of 0,0001. DGC is extraordinary able to remove B3 traffic and most datasets don't have over 20 accepted B2 addresses, which amounts to only 0,017% of all bots from B2.

Km distance	D10C	D11A	D11B
5	7	4	18
10	8	8	156
15	15	9	26
20	8	1	15
25	10	11	12
30	12	25	7
35	17	9	3
40	26	43	9
45	16	14	17
50	13	34	20

Table 6.5: *This table shows the amount of accepted B2 address when DGC is used to create clusters. DGC clusters are created with a a core point threshold of 3000 and km distance between 5 and 50. Clusters are created on datasets D10C, D11A and D11B. Next a bloom filter with an error rate of 0,0001 is used to store the found 24 bit networks. B2 traffic is then checked against this bloom filter. The accepted botnet traffic is counted and the amount is shown in the able. Reverse search is not used to re populate the database.*

As bots are created based on geography and not randomized as before, it's possible to see a higher correlation between the hypothesis applied on AFN and DGC. When botnets are simulated based on geography, a greater difference between AFN and DGC emerges. With randomized botnets seen in B1, the differences between these algorithms is not necessarily as clear. However, with geographical clustering in B2, the differences becomes easier to see. Geographical clusters often accept more networks, compared to an algorithm which just accept frequent networks. This means that with the previous botnets, seeing a difference between a frequent networks algorithm and an algorithm which use clustering, clustering algorithms will often have a higher acceptance rate of botnet traffic, largely due to the fact that the clustering accepts more overall networks.

For B2, the correlation of more accepted networks in the training phase, with more accepted botnet traffic, becomes flawed. AFN, which generally accepts less networks, has a higher B2 acceptance rate than DGC. If AFN is used to estimate frequent networks with a threshold of 20 in D11A, AFN will accept 3063 networks. AFN will also accept 24 bots from B2. In comparison, DGC will, with a km distance of 20 and a core-point threshold of 3000, accept 7249 networks and 15 bots. This means that DGC are able to accept twice the amount of networks without the accepted B2 traffic being proportionate to the amount of accepted networks.

RDGC

When RDGC cluster are created with a threshold of 3000, km distances between 5 and 50, minimum point of 3 and minimum length of 2, before the found 24 bit networks are stored in a bloom filter and the amount of accepted B2 traffic is counted against this bloom filter, the rate of accepted B2 data remains largely the same as the amount of accepted B2 traffic for DGC. The amount of accepted B2 traffic, which can be seen in Table 6.6, for a bloom filter with 0,0001 error rate, makes it difficult to see if the added constraints pay a difference in the amount of accepted botnet traffic. The bloom filter affects the amounts of accepted data for each cell differently. However, the acceptance rate is still impressively low and RDGC is exceptionally able to mitigate a DDoS attack from B2.

Km distance	D10C	D11A	D11B
5	21	8	13
10	7	10	7
15	11	4	6
20	32	11	9
25	13	20	6
30	12	14	7
35	40	26	15
40	29	17	8
45	19	13	12
50	14	13	15

Table 6.6: *This table shows the amount of accepted B2 traffic when RDGC is used to create clusters. RDGC clusters are created with a core point thresholds of 3000, km distances between 5 and 50, minimum points of 3 and minimum length of 2. A bloom filter with an 0,0001 error rate is used to store the found networks and B2 addresses are checked against this bloom filter. The amount of accepted B2 addresses are counted and shown in the table. Reverse search is not used to populate the database with more networks.*

6.2.3 B3

The third simulated botnet B3, is located in large parts of Asia and consist of 82000 unique IP addresses. The botnet stretches over countries as China, India, Japan and Bangladesh. B3 is even further away from where the normal traffic pattern is located. All algorithms should therefore be easily able to mitigate this attack without any particular problem.

AFN

AFN accepts more botnet traffic when a botnet is located in Asia than for a botnet in Europe. This have a correlation with B3 managing to have more bots in areas of normal traffic pattern. However, it doesn't mean all botnets in Asia are able to accept more traffic than botnets from Europe. It simply means that this exact botnet are able to correctly identify more networks which AFN also identified. Table 6.7 shows the amount of accepted B3 traffic when AFN is used to find frequent networks. Frequent networks are found for D10A, D10B, D10C, D11A and D11B. AFN calculates frequent thresholds from 1 to 1000.

AFN follows the same acceptance pattern on B3 as previous botnets. AFN manages to efficiently remove botnet traffic when employing high thresholds and struggles more with low thresholds.

Frequent threshold	D10A	D10B	D10C	D11A	D11B
1	8984	5715	2083	436	476
5	4990	3168	1156	192	191
10	4003	2679	952	109	120
20	3254	2165	735	60	52
50	1651	1079	350	23	5
100	983	574	147	2	2
250	419	190	15	2	1
500	175	68	1	1	0
1000	55	11	0	0	0

Table 6.7: *This table shows the amount of accepted B3 traffic when AFN is used to find frequent networks. First, AFN is used to find frequent 24 bits networks in the training set of D10A, D10B, D10C D11A and D11B. Then the amount of data from B4, that fits within the found networks, are counted. AFN is counted with the frequent thresholds 1, 5, 10, 20, 50, 100, 250 and 1000.*

DGC

DGC clusters created with a threshold of 3000 and km distances between 5 and 50, before the found 24 bit networks, are stored in a a bloom filter with an 0,0001 error rate, then the amount of accepted B3 traffic is counted against this bloom filter. This clustering and the accepted botnet traffic can be seen in *Figure 6.8*.

Km distance	D10C	D11A	D11B
5	526	10	6
10	606	14	29
15	631	15	7
20	627	15	8
25	624	11	6
30	634	25	25
35	632	18	6
40	635	13	11
45	660	28	7
50	630	15	15

Table 6.8: *This table shows the amount of accepted B3 traffic when DGC is used to create clusters. DGC clusters are created with a core point thresholds of 3000, km distances between 5 and 50, minimum points of 3 and minimum length of 2. A bloom filter with an 0,0001 error rate is used to store the found networks and B3 addresses are checked against this bloom filter. The amount of accepted B3 addresses are counted and shown in the table. Reverse search is not used to populate the database.*

The amount of accepted B3 traffic follows much of same hypothesis as B2. When a botnet is created based on geographical approaches and a solution is also created based on geographical patterns, the accepted botnet traffic remains low, even though a geographical approach might accept more networks in total. geographical approaches have some weaknesses when the geographical pattern align with botnet traffic. D10C have some clusters in India. Therefore, when B3 traffic also comes from India,the amount of accepted B3 traffic gets higher.

RDGC

RDGC clusters created with a threshold of 3000, km distances between 5 and 50, minimum required points of 3 and minimum length of 10. Manages to accept much of the same botnet traffic as DGC. A major difference between DGC and RDGC, is that RDGC accepts less botnet traffic for D10C. This is largely due to the extra constraints which prevents some cluster creations in China and India. These extra constraints remove 200 to 300 bots from B4.

6.2.4 B4

The fourth and last simulated botnet, B4, is located in Norway and consist of 38000 unique IP addresses. B4 is not necessarily representative for where botnets are located geographically, as botnets are normally situated in larger areas than just a single country. Moreover, since both D10 and D11 are largely located in Norway, mitigating a DDoS attack from this location is practical impossible, unless heavy constraints is set on the found data pattern. Setting heavy constraints on any data pattern, when bots are located in the area of normal traffic, will give a trade-of between the amount of accepted legitimate traffic and the amount of accepted botnet traffic.

AFN

When AFN is used to find frequent networks before the amount of accepted B44 traffic is counted, AFN has very high amounts of accepted traffic before the frequent threshold reaches 100 to 250. Table 6.9 shows the amount of accepted B4 traffic, when AFN are used to determine frequent networks. AFN are able to mitigate DDoS attacks from a botnet which is located in Norway, or from areas with high amount of seen traffic in the training phase, if the algorithm keeps high thresholds. Since AFN builds on the common hypothesis that frequent networks reoccur, AFN is able to determine networks which have high amount of seen traffic with the least amount of seen networks. Therefore, with a threshold of 500, AFN accepts a fraction of networks than with a threshold of 1. Accepting a small amount of networks makes it hard for a DDoS attack to render a service incapable. This is mainly contributed by the smaller probability for bots to have access to a service and knowing the legitimate pattern.

Frequent threshold	D10A	D10B	D10C	D11A	D11B
1	30502	27461	15770	17353	15212
5	25467	19318	7306	8890	8304
10	21502	15113	5277	5650	5562
20	16485	10397	3299	3688	3390
50	8277	4917	1285	1572	1329
100	3963	2398	674	608	584
250	1591	896	225	174	156
500	663	448	47	59	75
1000	245	159	9	34	44

Table 6.9: *This table shows the amount of accepted B4 traffic when AFN is used to find frequent networks. First, AFN is used to find frequent 24 bits networks in the training set of D10A, D10B, D10C D11A and D11B. Then the amount of data from B3 that fits within the found networks is counted. AFN is counted with the frequent thresholds 1, 5, 10, 20, 50, 100, 250 and 1000.*

DGC

DGC clusters created with a core-point threshold of 3000 and km distances between 5 and 50, have little to no possibility to mitigate a DDoS attack, if bots are located directly where a major normal traffic pattern is. Seen in Table 6.10, if geographical clusters are created where a high amount of bots also originates from, it is nearly impossible to filter out the botnet traffic. At this point, the botnet has managed to infiltrate the exact pattern which DGC creates cluster by. Therefore, trying to drop traffic based on the same pattern, is practical unfeasible.

Km distance	D10B	D10C	D11A	D11B
5	15843	8420	8511	5925
10	19155	9560	10216	8093
15	22555	11788	11803	8998
20	22899	12802	12389	9953
25	23614	12910	12551	10477
30	25938	13069	13939	12863
35	26072	13086	15593	14298
40	26157	14082	15624	14406
45	26226	14107	15619	14436
50	26261	14326	15627	14397

Table 6.10: *This table shows the amount of accepted B4 traffic if DGC is used to create geographical clusters. Geographical clusters are created with a core-point threshold of 3000 and km distances between 5 and 50. The found networks are stored in a bloom filter with an 0,0001 error rate and B4 addresses are checked against this filter. The amount of accepted B4 addresses are counted and shown in the table. Reverse search is not used to populate the database.*

The main reason for AFN being able to succeed in mitigating a DDoS attack, when the botnet pattern is located in the exact area of normal traffic flow, is that AFN can cover few networks, while still allowing a large amount of legitimate traffic. However, DGC builds on a separate hypothesis, that initial clusters should expand to find other points nearby. This hypothesis will often cause a huge amount of networks to be covered, as long as the networks lay close geographical. This means that if a network have only a couple of seen packets, while still being close to a core point, this network will as well be covered by the hypothesis. Therefore, the amount of accepted B4 traffic is highly correlated to the amount of accepted networks. This same correlation can be seen in B1. B2 and B3, which is created by a different hypothesis, where traffic originates from specific areas, where major clusters are not located, the amount of covered networks are not necessarily correlated with the amount of covered botnet traffic.

RDGC

RDGC clusters created with a core point threshold of 3000, km distances between 5 and 50, minimum point of 3 and minimum length of 10 have also a hard time of mitigating a DDoS attack from B4. Seen in Table 6.11, with stricter requirements, RDGC manages to remove some of the excess botnet traffic. However, the added constraints is not nearly enough, and the removed botnet traffic, of a dozen to a couple hundred bots, give nearly no differences.

Km distance	D10C	D11A	D11B
5	7884	8075	5883
10	9079	9365	7885
15	10897	11631	8833
20	12707	12293	9306
25	12834	12439	10432
30	12962	13807	12756
35	13043	15480	13309
40	14106	15519	14274
45	14078	15547	14310
50	14295	15548	14309

Table 6.11: *This table shows the amount of accepted B4 traffic, if RDGC is used to create geographical clusters. Geographical clusters are created with a core-point threshold of 3000, minimum length of 10, minimum points of 3 and km distances between 5 and 50. The found networks are stored in a bloom filter with an 0,0001 error rate and B4 addresses are checked against this bloom filter. The amount of accepted B4 addresses are counted and shown in the table. Reverse search is not used to populate the database with more IP addresses.*

One could except that RDGC would be better able to mitigate a DDoS attack from Norway because of the added constraints. However, adding a constraints

of a minimum length, don't necessarily mean that RDGC will cover less traffic in this area as long as other clusters can take over the dismantled points. The minimum length, simply states that clusters need to consist of a certain amounts of points to be considered legitimate. If a cluster then is dismantled because of few points, the points are still allowed to relocate to other clusters. Since Norway is a high density area, the area consist of several core points, and removed clusters will have a higher possibility of being a part of new clusters. This means that the added constraint will have a harder time of removing traffic, unless the cluster area is in a region where there is few clusters from before.

6.3 Reverse search

The previously discussed algorithms, DGC and RDGC, haven't accounted for a reverse search to populate the database with more networks. This means that only found networks from the training phase, which exist in a cluster, are allowed to access the service under the testing phase. However, the benefits of geographical clustering is to also allow new address under the testing phase. Table 6.12 shows the accepted botnets addresses for B1, B2 and B3 when RDGC clusters are calculated on D11A. RDGC clusters are calculated with a core-point threshold of 3000, minimum length of 2 and minimum points of 3.

Km distance	B1	B2	B3
5	591	56	37
10	1683	57	46
15	2176	84	49
20	2267	92	49

Table 6.12: *This table shows the amount of accepted botnet traffic for RDGC clusters on D11A. RDGC clusters are created with a minimum length of 2, minimum points of 3, core-point threshold of 3000 and km distances between 5 and 20. Reverse search is used to populate the database with more networks.*

Reverse search is used to populate the database with more networks. The amount of accepted botnet addresses remains quite small for D11A in B2 and B3. This is mainly contributed to the fact that D11A have most clusters in Norway. A reverse search to populate these clusters will therefore not contribute to an increase in accepted botnet traffic for botnets in Asia and Europe. However, populating B1 with more IP addresses will also have an increased risk of allowing more botnet addresses. How much a reverse search affects the risk of allowing more botnet traffic differs widely between different datasets and whøpere an attack pattern is located. Widely distributed clusters have a large risk of allowing more botnet traffic and elements as constraints where a cluster can expand, should help in mitigating the effects of populating a database with more networks.

6.4 Real life simulation

This section will go through 3 different real life simulations which use the botnets from last section to execute an attack against different datasets. Both AFN, DGC and RDGC are used to mitigate a DDoS attack and allow as much legitimate traffic as possible. Based on the previous section which tackles problems statement as; how much data traffic different structures can handle, the amount of allowed traffic in the testing phase and the amount of allowed botnet traffic with different structures. The real life experiments should be straight forward and not give any surprise in how well the technique is able to mitigate and deal with high amounts of traffic.

6.4.1 Real life simulation 1

The first real life simulation noted as *RL1* is run from B2, located in Europe, against D11A. A small D11A dataset is used to simulate the real traffic pattern under the attack. The small D11A subset consist of 50 000 requests, which last over 24 hours. DGC clusters calculated with a km distance of 10 and a core-point threshold of 3000 are used to accept new data. This DGC cluster consist of 5800 networks which should be accepted. If no reverse search is used to populate the database, 28800 of the 50 000 requests should be accepted. This amounts to 57,6% of the data pattern. This sub-set of D11A have a lower acceptance rate than the entire testing set of D11A, as with a core-point threshold of 3000 and km distance 10, 80,34% of the entire testing set is accepted.

For the simulation, 120 thousand requests are sent every second from B2 situated in Europe. Both the acceptance rate of legitimate request, bot request and the average time in millisecond a requests takes, are monitored. The CPU statistics and memory processing power are as well monitored on the web-server. A bloom filter with a *0,0001* error rate is used to store the found 24-bit networks that should be accepted. *Table 6.13* shows the amount of accepted legitimate and illegitimate traffic under the entire attack period. In total, more botnet traffic is accepted. However, this is mainly because B2 sends a high amount of traffic every second. When counted as an average of the entire attack period, 8.072 bot requests are allowed to access the service every second of the 120 000 possible requests.

Traffic	Accepted	Dropped
Legitimate traffic	28 793	21 202
Illegitimate traffic	969 317	14 313 921 586

Table 6.13: *This table shows the amount of accepted legitimate and illegitimate traffic under the RL1 simulation. The RL1 simulation consisted of B2 attacking a web-server with normal traffic pattern from a subset of D11A.*

To get a better view, we should look at the percentage of accepted and non-

accepted data pattern. At this point, 99,99% of all botnet traffic is dropped and 57.59% of all legitimate traffic is accepted. This data is shown in *Table 6.14*. The amount of accepted legitimate traffic correlates with how much data that should be covered in total for this subset of D11A. Data mining determines that 28800 addresses from the sub-set of D11A should be accepted with a core-point threshold of 3000 and km distance of 10. In our simulation 7 requests that should be accepted, was not accepted.

Traffic	Accepted	Dropped
Legitimate traffic	57.59%	42,41%
Illegitimate traffic	0,0067%	99,993%

Table 6.14: *This table shows the total amount of accepted and dropped requests noted by percentage points of all monitored legitimate and illegitimate requests in RL1. The RL1 simulation consisted of B3 attacking a web-server with normal traffic pattern from a subset of D11A.*

Statistics from the *client server* , which monitors the throughput to the web-sever, shows that the average request time for a request was 15,63 millisecond. Under the bloom filter section, an average request should take 5 to 6 milliseconds. This means that the heavy amount of attack traffic over an 24 hour period makes the round trip time of a request to take 10 milliseconds longer. The maximum, minimum, average and median time for a legitimate request to be processed and accepted, can be seen in *Table 6.15*.

Minimum	Maximum	Average	Median
4 ms	7,02 sec	15,64 ms	11 ms

Table 6.15: *This table shows the average, maximum, minimum and median request time that was seen for legitimate requests under the attack period from RL1.*

Although the highest measured time of a legitimate request is 7 second, this is only a minority of the monitored request. 1 legitimate request was sent from the client server to the web-server every second. Over the attack period, this amount to around 119 800 requests. The time distribution for the different monitored requests can be seen in *Figure 6.9*. This figure indicates the same as seen in Table 6.15, that most legitimate request use around 8 to 15 milliseconds to be processed. Of the 119 800 monitored requests over this time period, 99% used less then 53 milliseconds. Only 45 requests seen under this period, used over 100 milliseconds to be processed.

Moreover, the average NFQUEUE queue elements waiting to be processed was 211 elements. NFQUEUE has a max queue line of 65 556 elements before elements are dropped. If elements are dropped they will not be processed by

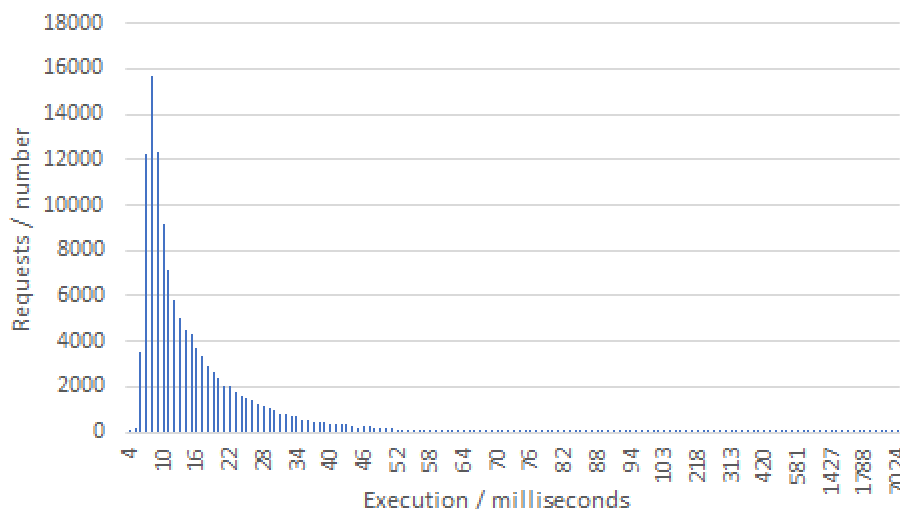

Figure 6.9: *This figure shows the request time for all legitimate request that the client server issued against the web-server under RL1.*

our solution and we can risk getting a high drop rate of legitimate requests that should be accepted. NFQUEUE never dropped an element during the 24 hour simulation. However, it was reported a drop rate of 105 170 packets because netlink was not able to send packets to userspace. This might imply that either the netlink buffer size was too small or that NFQUEUE was not able to handle this amount of traffic over a larger period of time. The amount of dropped messages between NFQUEUE and user-space has nothing to do with the mitigation techniques on our side. However, a solution that operation in the kernel-mode would have avoided this problem all together.

The Apache server never got a constraint in the amount of necessary packets to process and the CPU usage remained close to 0,9%. The average CPU usage on the mitigation techniques that processed 120 000 packets every second remained at 82%. This includes the measured CPU usage of the 4 different threads that had the task of mitigating the DDoS attacks. The average CPU usage in total remained at around 52,357%

Overall the DGC mitigation technique has been successful in mitigating a DDoS attack from B2. This knowledge should already be clear by looking at the previous sections. In overall with a B2 botnet located in Europe, DGC clusters with core-point threshold 3000 and km distance of 10, should let in 8 botnet addresses for D11A. Moreover, the average acceptance score of D11A should be around 80%. Although this is not the case, the acceptance rate will mostly increase to 80% with a higher amount of tested data pattern.

6.4.2 Real life simulation 2

The second real life simulation is noted as RL2. The real life simulation is run on botnet B3, situated in Asia, against a sub-dataset of D11A. Unlike the previous real life experiment, reverse search is used to populate the database. RDGC clusters are calculated with a minimum required points of 3, minimum length of 2, km distance of 15 and a core-point threshold of 3000. Without a reverse search, *6 487* unique networks are identified and accepted in the testing phase. *354 276* more networks are identified during a reverse search. Of the newly identified networks, 1483 networks which were seen in the training phase were not re-identified during a reverse search. This means that a bloom filter should accept *355 759* networks. The bloom filter use an error rate of 0,0001.

The sub-dataset D11A contains *150 000* requests over a time-spam of 8 hours. *111 886* requests should be accepted in solution which accept seen networks from the training phase which also exist in a cluster. However, in a solution that are able to accept a packet as long as it belong to a cluster based on its location, 117 188 should be accepted. Over the 8 hour experiment 100 000 request was sent from B3 every second. The amount of accepted legitimate and illegitimate traffic can be seen in *Table 6.16*.

Traffic	Accepted	Dropped
Legitimate traffic	116 313	33 687
Illegitimate traffic	2 113 845	3 059 007 969

Table 6.16: *This table shows the amount of accepted legitimate and illegitimate traffic under the RL2 simulation. The RL2 simulation consisted of B3 attacking a web-server with normal traffic pattern from a subset of D11A. RDGC clusters with core-point threshold of 3000, minimum length of 2, km distance of 15 an minimum required points of 3 was used to mitigate the attack.*

Table 6.16 shows that 116 313 legitimate request was accepted under RL2. This is around 875 requests shorter than the expected amount of 117 188. Netlink has a reported drop rate 218 713 packets between kernelspace and userspace. This can be a large factor to why the solution was not able to achieve the optimal rate. The percentage score of accepted legitimate and illegitimate traffic can be seen in Table 6.17. Very little traffic is accepted from B3 even though reverse search is used to populate the database. At an average 66.89 bot request are accepted every second. D11A is mainly situated in Norway and don't have any clusters in Asia where most of B3 traffic arrives from. This means, even when re-populating the database, a solution is easily able to mitigate an attack. Moreover, this solution with reverse search, makes the optimal amount of 77,542% from D11A-subset to be accepted. This is 3 percentage points higher than a solution which can't look up new addresses.

Based on statistic from the client server, which sends 1 legitimate request to the web-server every second, a request used an average of 13.15 milliseconds.

Traffic	Accepted	Dropped
Legitimate traffic	77,542%	22,548%
Illegitimate traffic	0,0690%	99,9309%

Table 6.17: *This table shows the amount of accepted legitimate and illegitimate traffic under the RL2 simulation. The acceptance rate is noted as the percentage point of either all accepted botnet traffic or all accepted legitimate traffic. The RL2 simulation consisted of B4 attacking a web-server with normal traffic pattern from a subset of D11A. RDGC clusters with core-point threshold of 3000, minimum length of 2, km distance of 15 an minimum required points of 3 was used to mitigate the attack.*

The minimum time a request made was 5 milliseconds and the maximum time a request took was 3,01 seconds. *Figure 6.10* shows this distribution of execution time for requests under RL2. The execution time is slightly better than RL1. Which is mainly due to the decrease of bot request every second.

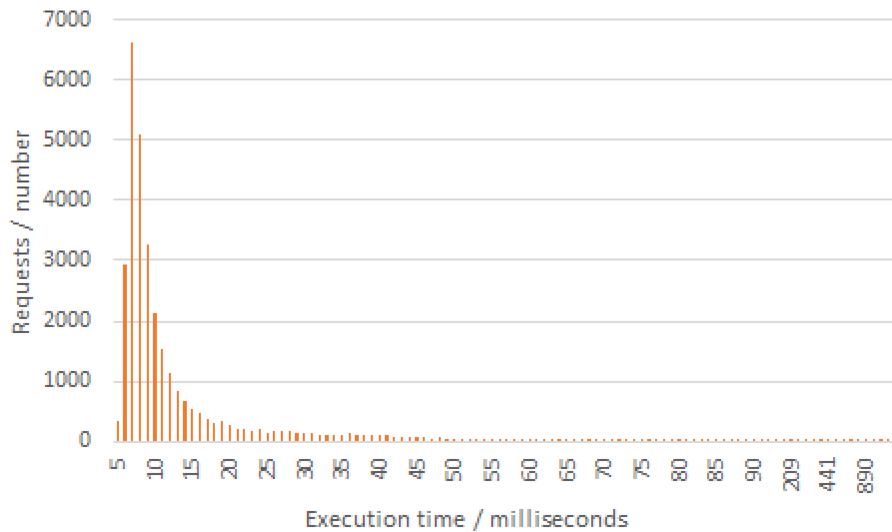

Figure 6.10: *This figure shows the request time for all legitimate requests that the client server issued against the web-server under RL2.*

As RL2 allowed more botnet traffic than RL1, the Apache server got a slightly heavier load and the CPU usage remained at 5,7%. However, since the mitigation solution dealt with 20 000 less packets each second than RL1, the average total CPU usage remained at 49,7%. RDGC clustering on D11A, with the use of reverse search to populate the database, manages to mitigate DDoS attacks when the attack comes from specific areas where geographical clusters

don't exist.

6.4.3 Real life simulation 3

The last real life simulation RL3 is done using B1 to send 130 000 HTTP requests every second against a sub-set of D11B. AFN is used with a threshold of 1 to find *frequent networks*. A tree structure is used to contain the networks which should be accepted. AFN with a threshold of 1 will accept all networks which was seen in the training phase. This amounts to 18093 networks. Under the simulation, 100 000 request is sent from D11B sub-set over a timespan of 5 hours. Data mining shows with a AFN threshold of 1, 93811 of the 100 000 legitimate requests should be accepted. This amount to 93,81% and are 12 percentage points higher than for the entire D11B set, where the acceptance rate would be 81%. Table 6.18, shows the amount of accepted legitimate and illegitimate traffic over this simulation period.

Traffic	Accepted	Dropped
Legitimate traffic	92.8348%	7,1652%
Illegitimate traffic	0,104%	99,896%

Table 6.18: *This table shows the amount of accepted and dropped legitimate requests under RL3. The amount is shown as a percentage points of either all legitimate or illegitimate traffic.*

Of the 130 000 requests sent from B1 every second, 134 packets are on average accepted. This B1 acceptance rate causes both higher wait periods for legitimate requests, and higher CPU usage in total. Instead of AFN accepting 93811 legitimate requests, AFN accepted 92832 legitimate requests. This is around 1000 fewer requests than the expected amount. This decrease can be contributed to several reasons; Firstly, Netlink reported a drop rate of 43 million packets between the kernelspace and userspace, and this can have caused some packets to not be processed. Secondly, longer waiting periods can have caused some legitimate requests to be dropped. The reported drop rate by Netlink has nothing to do with the actually mitigation technique. Instead, the problem stems from the fact that the solution is created in user-space and not kernel space, which would have avoided the issue of transporting packets to userspace. *Table 6.19* and *Table 6.20* shows different request times and CPU usages seen under RL3. The average time for a request have increased to 123 ms. Moreover, the Apache usage have increased to 12% from 5,7% in RL2.

The higher bot traffic of 130 000 every second caused longer waiting periods in NFQUEUE. At an average, the queue had 2306 elements waiting to be processed. The queue never got overfilled and dropped 0 elements. Except for a slight inconvenience with Netlink, AFN seems to be successful in mitigating attacks from B1. The three simulations are mostly successful in mitigating large

Minimum	Maximum	Average	Median
4 ms	5,2 sec	123 ms	33 ms

Table 6.19: *This table shows the average, maximum, minimum and median request time that was seen for legitimate requests under the attack period from RL3.*

amounts of data traffic without overflowing the CPU. The highest problem with AFN, DBGC and RDGC relays mostly to the acceptance rate of illegitimate traffic. This acceptance rate will vary greatly between different botnets. However, we should be able to assume that as long as an attack pattern differentiates from a legitimate pattern, the attack pattern is possible to mitigate.

Total CPU usage	Apache CPU usage	Tree CPU usage
51,3%	12,237%	75%

Table 6.20: *This table shows the average of the total CPU usage, the apache CPU usage and the tree/solution CPU usage. The CPU usage is calculated under the attack period of RL3.*

Chapter 7

Discussion

This chapter will discuss the presented algorithms and results. Emphasis will be put on how credible the results are and what strengths and weaknesses the different algorithms have. Moreover, both future works and changes that should be applied to ensure an optimal solution, are discussed.

7.1 Algorithms

This thesis have proposed 3 algorithms. The first algorithm, AFN, builds on the idea that frequent networks will repeat to a high degree. Results from the data mining chapter acknowledge that higher threshold applied in AFN decreases the distance between covered data in the training and testing phase. AFN have the ability, to cover the least amount of networks for the most amount of covered data. This is because, AFN, will always accept networks which have the highest applied previously seen traffic. This makes AFN an effective algorithm to mitigate large DDoS attacks, since the accepted data pattern can be narrowed down to a high degree. From the result chapter, AFN are able to mitigate DDoS attacks even though the botnet traffic is directly in the area of where normal traffic flow is located. AFN strongest disability, is that it requires a network to be seen enough times to be considered legitimate. Moreover, AFN don't have the ability to differentiate between legitimate and abnormal traffic from networks which has small amounts of seen traffic. This can be seen in the data mining chapter, where lower thresholds applied in AFN, gives an increase in the amount of differences between covered data in the training phase and testing phase. Moreover, AFN, might need to set a threshold low to accept enough legitimate traffic, and this can cause abnormalities in the dataset to be accepted.

The second and third algorithm builds on traffic reoccurring in geographical areas of previously seen traffic. Geographical clustering within anomaly detection techniques have not been previously proposed. However, this thesis argues that geographical clustering gives great benefits which have not been seen be-

fore. Geographical clustering have the ability to determine legitimate networks which have little seen traffic and can easier differentiate between anomalies in the traffic pattern. Although, this is a strength, this is also a weakness within geographical clustering. The algorithm might accept too many networks and this can cause issues for services with very widespread patterns. However, comparing the amount of accepted data in the training and testing phase, density-based clustering are able to cover less data in the training set, while covering larger amount of data in the testing set. This shows a trend, that geographical clustering are easier able to identify a unique pattern which have a higher likelihood of occurring.

Moreover, geographical clustering will be more resistant against DDoS traffic, as long as the attack pattern is not directly located within a cluster. Geographical clustering have the ability to determine new networks within a cluster. This gives great opportunities for services, when there is a small amount of known data. This approach of populating the database, have some weaknesses by not determining areas which are more likely to occur. However, the approach shows promising results which can be used to create better algorithms. Moreover, the second proposed clustering algorithm, RDGC, shows that by simply adding some constraints, the algorithm are still able to achieve the same amount of new seen networks in the testing phase as DGC, while still keeping a lower amount of accepted networks in total.

7.2 Attack vector

This thesis has primarily been concerned with HTTP flooding attacks. The real life experiment with BoNeSi have only been tested against HTTP flooding. However, this doesn't mean that the proposed techniques won't work against different attack vectors. The mitigation technique will have no effect on DDoS attacks which targets the network layer and overflow the network link before the mitigation technique. However, the system will be able to mitigate SYN flooding or other attack vectors, which focus on depleting a limited server resource. The success of mitigating these attacks depends only on two factors; The normal traffic pattern needs to follow the proposed hypothesis from the data mining chapter, and the attack pattern needs to be distinct from normal traffic flow. Therefore, for the AFN(Apriori-based frequent networks) algorithm to successfully mitigate other attack vectors, the normal traffic flow, needs to repeat to a high degree. While for density-based clustering to succeed in mitigating different attack vectors, the normal traffic flow needs to repeat in the same geographical area as before. The mitigation techniques are therefore only dependent on these factors to successfully mitigate new and different attacks, which focus on depleting server resources.

7.3 Datasets

The datasets D10 and D11 are taken from Oslo and Akershus University College of Applied Sciences(HiOA). D10 shows sign of being a distributed dataset and contains a high amount of noise, while D11 is more localized and contains less noise. The datasets are collected from web-servers, and the sets show signs of having a repeated data pattern. This follows remarks from the background chapter, where a data pattern, within a service, is often very consistent. Because of this, our two hypothesis mentioned in chapter 4.2 and 4.3, has been verified, and we have managed to classify new data. More research should be concluded to determine if different services, with different datasets, also have a consistent data pattern, as only datasets with consistent data patterns are able to follow the hypothesis and perform well when categorizing new data.

Moreover, although AFN only needs a data pattern to emerge within the same network, density-based geographical clustering needs a data pattern to emerge in geographical regions. Geographical clustering works well for datasets with regions of traffic hot spots, and worse for datasets with highly distributed traffic. However, as remarked by Chapelle[21], on unsupervised learning, it is assumed that a data pattern is identically distributed from a distribution point x. Meaning that a data pattern should follow an unknown structure. It should be assumed that finding this unknown structure based on geographical regions is more legitimate than finding the structure based on different attributes.

7.4 Data structure

The networks found from the different algorithms are stored in two different data structures; A Bloom filter and a binary tree structure. Both data structures manage to quickly decide the fate of any new packet. However, the binary tree structure is slightly faster. This is primarily because a Bloom filter with low error rate has a high time complexity. The time complexity of a Bloom filter is not related to the amounts of packets in the filter, but the number of hash functions. The trade-off between more available memory and a higher time complexity will differ between different datasets and anomaly detection approaches. Anomaly detection methods, where the time issue isn't as severe as for DDoS attacks, might do well with a structure that saves memory. However, if a detection mechanism needs to handle a high amount of packets every second, the trade-off with more memory, against higher error rate and higher time complexity, will often not work.

7.5 Mitigation point

Mitigating techniques can be applied at three different points in the internet infrastructure; source-end, core-end and victim-end[14]. Source-end means close

to the source of the attacker. Core-end is at high-performance backbone routers on the internet. Victim-end is close to the targeted server. These mitigation points have different weaknesses and strengths. Our own mitigation technique is applied at the victim-end. Filtering mechanisms applied at the victim-end can easier discriminate between legitimate and illegitimate traffic. However, victim-end mechanisms will often suffer from having a saturated bandwidth link.

Source-end filtering would technically be the best point to filter malicious traffic. Any source-end mechanism will only experience small amount of the malicious traffic. The source-end defense also needs less processing power to process incoming packets. However, filtering mechanisms applied at the source-end, have a harder time of filtering suspicious traffic. Moreover, source-end mechanisms are not able to detect its effect on the data traffic and can suffer from a high number of false positives and false negatives. An example of a filtering mechanism applied close to the source end is ingress filtering. Due to large amounts of traffic, high-performance backbone routers, are unlikely to use a part of their own processing power to filter illegitimate traffic. Backbone routers also have a harder time of filtering out illegitimate traffic[14].

Although victim-end mechanisms can suffer from a saturated bandwidth, the mitigation approach doesn't need to be situated directly at the targeted victim. However, it still needs to be rather close to the victim source. Our mitigation approach can therefore be placed at the gateway or at edge routers of the ISPs. The edge routers can then be configured to filter traffic independently for each source address that provides a service. This can minimize the risk of the network link to be over-saturated, as links further away from the source will have more available bandwidth.

7.6 Problem statement

Two different problem statements were stated in the introduction chapter. The first problem statement builds on the foundation that hidden data structures within datasets can be found: *How can we use data mining to find pattern correlations in data history to build efficient filtering rules that are able to mitigate DDoS attacks?*. This statement has been investigated in the data mining chapter. Results show that both density-based geographical clustering and AFN, manages to find hidden data structures which can determine if new data is legitimate or illegitimate. Both approaches use unlabelled learning techniques to determine hidden structures. Density-based clustering have the ability to sort out anomalies based on geographical regions and the clustering algorithms are able to cover more data in the testing set with less covered data in the training set. Moreover, density-based geographical clustering also have the ability to accept relevant networks, although some networks don't contain a high amount of traffic. AFN can sort out anomalies in the dataset by relaying on relevant networks repeating to a high enough degree. AFN are able to cover considerable less network while still remaining a high acceptance rate of legitimate traffic in

the testing set. In the end, both these algorithms show that they are heavily able to find hidden patterns which can be used to determine new traffic.

The second problem statement states; *To which extent is our solution resilient to DDoS attacks of varying magnitude?*. In our simulation, the largest simulated attack rate is 130 000 requests per second. This resulted in a CPU usage of around 40-60%, meaning both a Bloom filter and tree structure should be able to handle 260 000 requests per second with only 4 CPU cores. Sucuri, which is a company within website security, reports that HTTP flooding attacks generates, on average, 7 282 HTTP requests per second with a peak of 49,795 requests per second. Moreover, attacker botnets used on average 11 634 different IP addresses, where the largest botnet used 89 158 different IP addresses[22]. Our mitigation techniques, which has a primary focus on removing application layer flooding attacks, manages easily to deal with this amount of traffic. Moreover, for attacks which targets the networking layer, the attack rate is measured in Mbps. The mitigation of these attacks relay on the traffic reaching the mitigation approach. However, most reflection based attacks can be mitigated without using traffic profiling, as these attacks contain signatures which can be used to mitigate the attacks. These attacks themselves don't care about a mitigation approach as they focus on exhausting the network bandwidth before any mitigation approach is reached. Moreover, the efficiency of our approach can be improved by writing the mitigation code as a part of a kernel module in order to be able to handle a higher packet amount. The proposed solution is therefore highly adaptable to deal with high amounts of packets per second. Additionally, since the mitigation technique only focus on one attribute, the round trip time for legitimate users can remain fairly low, even with high packet rates per second.

7.7 Future work

This section will address possible future work, which can make the approach more robust to application layer attacks. Both better defined populating mechanisms and the use of more attributes are discussed.

7.7.1 Populating density-based clustering

Density-based geographical clustering have the extra advantage of being able to find new relevant networks, if a system has a limited amount of normal traffic. Populating the database with more networks can be an easy way to ensure that more legitimate traffic are accepted. However, the database should be populate to a minimum degree. This is to optimize the acceptance rate of legitimate traffic without accepting heavy amounts of traffic. Accepting more traffic will ultimately lead to more illegitimate traffic also being accepted. To keep this trade-off favorable, clusters should preferable only be populated in the most relevant regions with new networks.

RDGC clustering was an attempt of accomplishing this task. Heavier constraints on where clusters can expand limit the clusters creation and therefore the amount of accepted networks, when populating the database. Table 7.1 shows the amount of accepted networks when RDGC and DGC clustering are used on D11A with a core-point threshold of 3000. DRGC clusters are created with a minimum length of 2 and minimum points of 3. The clusters are then populated with more networks.

Km distance	DGC	RDGC
5	609 339	107 788
10	684 814	279 804
15	928 774	360 763
20	943 322	376 277

Table 7.1: *Shows the amount of accepted networks for GC and RGC clustering when populating the database with more networks.*

This is a simplistic process of deciding which geographical areas are deemed more relevant. However, with the use of heavier constraints, legitimate networks, which otherwise fit with the hypothesis, are still dropped. Moreover, RDGC clustering are not able to prevent widespread clusters from not getting a high degree of new networks. Therefore, even though populating a solution with more networks might be an efficient way to find new networks, this should be done carefully.

It is possible to use a different approach to decide what location points are allowed to be populated with more networks. It can be assumed that heavier density-areas within clusters are allowed to look for more relevant networks. The density-areas can be assumed centered around a common core-point x. As seen in Figure 7.1, if a cluster exists of Y points, with the core-point marked as a black dot, a high density area around core-point x can be seen as more legitimate than cluster points, which lay further away from the core-point. Here green dots are marked as laying closer to other green dots near them. While red dots are marked as laying further away from other connected red dots.

Although an algorithm might include all points into one cluster, only the highest density area around the common core-point is included in a possible reverse search. This can ensure that clusters only include the most relevant new networks, which has not been seen in the training phase.

Figure 7.1: *The first picture, shows a cluster of X points, with the core-point remarked as black dot. The second picture shows the distance between the different point towards the points it is density connected to. Green dots have a shorter distance with their density connected green dots. Red dots are further away from their density connected red dots.*

7.7.2 Different attributes

Within density-based geographical clustering it is possible to easier find relevant attributes within each cluster. We should be able to assume that these clusters share other unique attributes. These attributes can include the TTL value and browser behaviour. A web-server which has several languages could assume that clusters located in Germany might use the web-site in German, while a cluster located in England might use the web-site in English. Moreover, for a college located in Norway, clusters located here, might assume these users work or study at the college and therefore have a different browser behaviour than clusters in other countries. Using geographical clustering to achieve this goal gives an easier way of grouping together these data. Furthermore, this makes it easier to sort out anomalies in the set and makes it substantially harder for an attacker to identify a unique pattern to gain access to a service.

7.7.3 Employing time

The discussed algorithms, when employed in a real-life DDoS scenario, use only the IP address to identify if an address fits within a cluster. The IP address is therefore the only parameter which decides if a packet should be accepted. Even though this in most cases works well in practical terms, the algorithms will struggle to cover a significant pattern without compromising the server, when dealing with servers which have a high traffic distribution. Moreover, if reverse search is used to increase the amount of accepted traffic, it will cause significant issues when dealing with widespread clusters. Therefore, instead of only relaying on one attribute, clusters can be configured with regards to several attributes. One of these attributes could be to calculate the time for each cluster when traffic is most seen, before creating a solution which only accepts traffic from a cluster, when traffic have a higher probability of occurring.

Employing time builds on the hypothesis that each server has its own significant pattern. As seen in *figure 7.2*, when all requests for the training phase of *D10C* is mapped for the time the requests appeared. This shows a pattern where most requests appear from 10:00 to 20:00 occurs.

Moreover, considering only clusters in Norway. Traffic for Norwegian clusters contains unlike the general consensus of the entire traffic pattern. a pattern that is largely divided into significant traffic during the day, while little to none traffic during the night. This means that traffic from these clusters, or for Norwegian cluster as a whole, can be denied during the night, while accepted during the day. Moreover, if these clusters contain enough traffic to determine a time pattern from each cluster, an algorithm can independently compute when traffic should be accepted for a specific cluster.

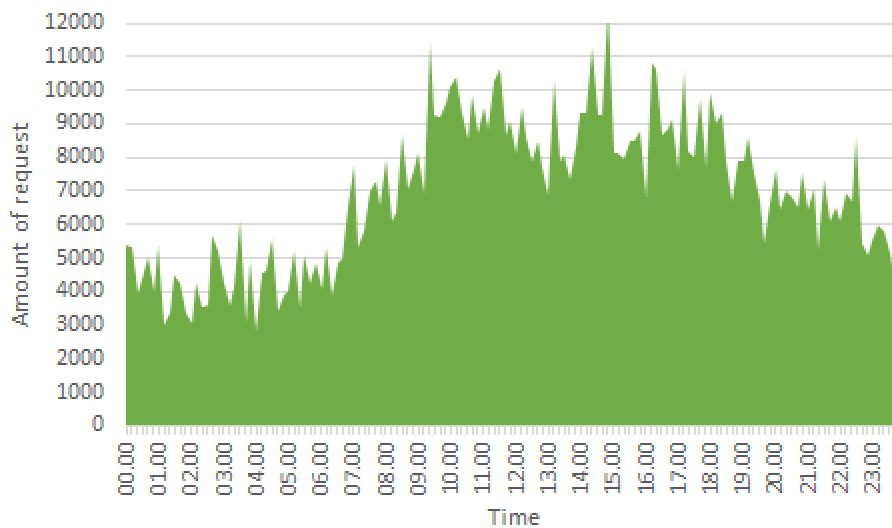

Figure 7.2: *Shows the total amount of incoming traffic, for D10C, during the training period mapped to the specific time the requests appeared. The number of requests every 10 minute during the training period, is individual mapped on the graph. The D10C web server is located in Norway and UTC+01:00 is used to map the requests to their specific time.*

Chapter 8

Conclusion

Three different algorithms have been proposed in this thesis to mitigate DDoS attacks. The algorithms are mainly applied to mitigate HTTP flood attacks, but can also be used to mitigate other attacks at the application layer. The first algorithm is build on finding frequent networks based on previously seen data, while the second and the third algorithm are build on creating clusters in geographical areas.

AFN has the ability to identify networks which have a high amount of seen traffic. This will allow AFN to accept few networks for a high amount of accepted traffic. In one way, AFN will more easily mitigate a DDoS attack than geographical clustering, since it will allow fewer networks. However, if we still want to accept relevant users, even if they don't belong to a frequent network, geographical density-based clustering is a better suited alternative. Geographical density-based clustering have not previously been proposed within anomaly intrusion detection and have provided promising results. Previous, clustering algorithms within anomaly intrusion detection focuses on clustering based on known attributes, and not based on geographical regions. These attributes include features from the TCP protocol and IP protocol. The attributes can differ in everything from source address and destination address to packet size, TTL and flags[66][32][86].

This thesis argues, that clustering techniques based on these common attributes, can give an incorrect view of the hidden data structure. One of the most central and unique features to base clustering on, is the source IP address. Anomaly based mitigation techniques often use this feature in clustering[32][59]. However, these clustering methods build on the hypothesis that if an IP address appears in address space X, this address is close to IP address $X + 1$. In other terms, this clustering technique takes advantage of the IP neighborhood relationship between networks to cluster them together. This is not necessarily ideal, as networks which are close in the address space, might not be close geographically. Moreover, this assumption might lead to networks which are far away in the address space being assumed to be anomalies, even though they are

close together geographically.

Geographical density-based clustering overcomes this shortcoming and allow IP addresses to be clustered together if they lay close geographically. This gives several advantages that has not been seen with normal IP address clustering; anomalies can be more correctly determined, new networks and IP addresses can be found based on defined clusters, and clusters can correctly identify data which have a higher likelihood of occurring in the testing phase. Moreover, based on geographical clusters, different attributes can be related to each independent cluster and can give a more correct view of the hidden data structure. This will make it harder for an attacker to adapt the attack pattern to the normal traffic flow.

This thesis has proven that both machine learning techniques and data mining are efficient mechanisms to counter DDoS attacks. Results show that both AFN and density-based clustering are able to mitigate upwards of 99% of simulate botnet traffic. More research should be executed to ensure optimal results within density-based geographical clustering. However, geographical density-based clustering has given a new and exciting way to cluster data, and has given the possibility of overcoming problems with sampling of normal traffic patterns, and the ability to counter anomalies in the datasets.

Chapter 9

Appendix

9.1 Algorithms

9.1.1 AFN

```
#!/usr/bin/python3
import sys
def alg1(T, C,frequency,level):
   frequency['1'] = 0
   frequency['0'] = 0
   k = 1
   while len(frequency) > 0 and k <= 32:
      #Goes through the list and updates the frequency for every bit-string.
      with open(T) as sOutput:
         for line in sOutput:
            bit = line[0:k]
            if bit in frequency:
               frequency[bit] = frequency[bit] + 1
      #Deletes keys that don't have over the accepted frequency
      for key in list(frequency):
         if frequency[key] <= C:
            del frequency[key]
      if k == level:
         for key in list(frequency):
            with open("frequentBits_{}".format(C), "a") as f:
               tekst = "{} \t {}\n".format(key,frequency[key])
               f.write(tekst)
         exit()
      #Creates new keys for k+1
      for key in list(frequency):
         del frequency[key]
         frequency[key+'0'] = 0
```

```
            frequency[key+'1'] = 0
        k = k + 1
threshold = int(sys.argv[1])
lvl = int(sys.argv[2])
binary_file=(sys.argv[3])
alg1(binary_file, threshold,{}, lvl)
```

9.1.2 DGC

```
#!/bin/python
import sys
import os.path
from geopy import distance
from geopy import Point
import re
def extend_one_cluster(core_point, dataset, threshold,frequent_threshold,core_points):
    cluster = [core_point]
    for cluster_value in cluster:
        for new_point in reversed(dataset):
            if find_distance(cluster_value, new_point) <= threshold:
                cluster.append(new_point)
                dataset.remove(new_point)
                if int(new_point[2]) >= frequent_threshold:
                    try:
                        core_points.remove(new_point)
                    except ValueError:
                        pass
    return cluster
def write_cluster(cluster, threshold,km):
    with open("cluster_{}_mc_{}_alg2_D11A.km".format(threshold,km), "a") as f:
        for point in cluster:
            f.write("{} {} {}\n".format(point[0], point[1], point[2]))
        f.write("\n")
def extendCluster(dataset, core_points, threshold,frequent_threshold):
    clusters = []
    for core_point in core_points:
        dataset.remove(core_point)
        clust =  extend_one_cluster(core_point, dataset, threshold, frequent_threshold,core_points)
        write_cluster(clust, threshold,frequent_threshold)
        clusters.append(clust)
def find_distance(pointX, pointY):
    try:
        p1 = Point("{} {}".format(pointX[0], pointX[1]))
        p2 = Point("{} {}".format(pointY[0], pointY[1]))
        return  distance.distance(p1,p2).kilometers
    except ValueError:
```

```
        return 10000
def get_corePoints(dataset, threshold):
    core_points = []
    for object in reversed(dataset):
        if len(object) >= 3 and int(object[2]) > threshold:
            core_points.append(object)
    return core_points
def get_datapoints(datafile):
    dataset = []
    with open(datafile) as sOutput:
        for line in sOutput:
            ansi_escape = re.compile(r'\x1b[^m]*m')
            line_edited = ansi_escape.sub('', line)
            x_regex="[0-9\.\-]{1,50}"
            val = re.findall(x_regex,line_edited)
            dataset.append(val)
    return dataset
km=int(sys.argv[1])
threshold=int(sys.argv[2])
datapoints=(sys.argv[3])
dataset = get_datapoints(datapoints)
core_points = get_corePoints(dataset, threshold)
extendCluster(dataset, core_points,km,threshold)
```

9.1.3 RDGC

```
#!/bin/python
import sys
import os.path
from geopy import distance
from geopy import Point
import re
def adbc(clusters, dataset,distance,minpts,thoints,file,minLength):
    all_cores_for_each_point = []
    all_points = []
    for dataset_point in reversed(dataset):
#       print(dataset_point)
        num_points = 0
        for checked_point in dataset:
            if find_distance(dataset_point, checked_point) <= distance:
                num_points = num_points + 1
                #every point will match with itself, therefore (minpts+1)
                if num_points >= (minpts+1): # we now this  point have enough points to join a cluster
                    core_points = {}
                    for clust_int in range(0,len(clusters)):
                        new_dist = find_distance(clusters[clust_int][0], dataset_point)
```

```
21                          hash = str(clusters[clust_int][0])
22                          core_points[hash] = new_dist
23                      lists = sorted(core_points.items(), key=lambda x:x[1])
24                      all_cores_for_each_point.append(lists)
25                      all_points.append(dataset_point)
26  #                   print("point is: {}".format(all_points[0]))
27  #                   print("first core are: {}".format(all_cores_for_each_point[0][0][1]))
28                      break
29      #All possible cores is computed for all points that satisfy the minpts constraint.
30      #Start computing the core the dataset belong to based on density estimation.
31      #check if the point can belong to closest core, next closest core and so forth.
32      iterations = 0
33      ant_cores = len(clusters)
34      change = False
35      while iterations < ant_cores:
36          change = False
37          for point_int in reversed(range(0, len(all_points))): #All points.
38              point = all_points[point_int]
39              core = all_cores_for_each_point[point_int][iterations]
40              #Find the cluster this core is related to!
41              for cluster in clusters:
42                  if str(cluster[0]) == core[0]: #This is the correct cluster for this core point.
43                      for cluster_point in cluster: #Check if point is density based to this cluster
44                          if find_distance(point, cluster_point) <= distance: #point is dens based.
45                              cluster.append(point)
46                              del all_points[point_int]
47                              del all_cores_for_each_point[point_int]
48                              change = True
49                              break
50                      break
51          if change == False:
52              iterations = iterations + 1
53      recompute_unexpandable_clusters(clusters,distance,minLength)
54      with open(file, 'a') as f:
55          for cluster in clusters:
56              for point in cluster:
57                  f.write("{} {} {}\n".format(point[0], point[1], point[2]))
58              f.write("\n")
59  def recompute_unexpandable_clusters(clusters,distance,minLength):
60      for cluster_check in reversed(range(0, len(clusters))):
61          #This cluster is not long enough
62          if len(clusters[cluster_check]) <= minLength: #The cluster clusters[cluster_int]  only exist of 1 point, the core point, check computati
63  #           core_points = {}
64              all_cores_for_each_point = []
65              all_points = []
66              for remove_point in clusters[cluster_check]: #Goes through all points that should be removed.
67                  core_points = {}
68                  for clust_int in range(0,len(clusters)): #old clusters
```

```
69                      if clusters[cluster_check] is not clusters[clust_int]: #if this cluster is not the cluster we are checking
70                          new_dist = find_distance(clusters[clust_int][0], remove_point)
71                          hash = str(clusters[clust_int][0])
72                          core_points[hash] = new_dist
73                  lists = sorted(core_points.items(), key=lambda x:x[1])
74                  all_cores_for_each_point.append(lists)
75                  all_points.append(remove_point)
76      #Start computing the core the dataset belong to based on density estimation.
77      #check if the point can belong to closest core, next closest core and so forth.
78              iterations = 0
79              ant_cores = len(clusters)
80              change = False
81              while iterations < ant_cores-1: #Need to not count the 1 core we want to remove.
82                  change = False
83                  for point_int in reversed(range(0, len(all_points))): #All points that should be removed.
84                      point = all_points[point_int]
85                      core = all_cores_for_each_point[point_int][iterations]
86                      #Find the cluster this core is related to!
87                      for cluster in clusters:
88                          if str(cluster[0]) == core[0]: #correct core!
89                              for cluster_point in cluster:
90                                  if find_distance(point, cluster_point) <= distance:
91                                      cluster.append(point)
92                                      del all_points[point_int]
93                                      del all_cores_for_each_point[point_int]
94                                      change = True
95                                      break
96                              break
97                  if change == False:
98                      iterations = iterations + 1
99              del clusters[cluster_check] #The single cluster point is removed in the end, if it is close to an other core, it is already added to tha
100 def get_corePoints(dataset,core_point_threshold):
101     core_points = []
102     for object in reversed(dataset):
103         if len(object) >= 3 and int(object[2]) > core_point_threshold:
104             core_points.append([object])
105             dataset.remove(object)
106     return core_points
107 def get_datapoints(datafile):
108     dataset = []
109     with open(datafile) as sOutput:
110         for line in sOutput:
111             ansi_escape = re.compile(r'\x1b[^m]*m')
112             line_edited = ansi_escape.sub('', line)
113             x_regex="[0-9\.\-]{1,50}"
114             val = re.findall(x_regex,line_edited)
115             dataset.append(val)
116     return dataset
```

```
117 def find_distance(pointX, pointY):
118     try:
119         p1 = Point("{} {}".format(pointX[0], pointX[1]))
120         p2 = Point("{} {}".format(pointY[0], pointY[1]))
121         return distance.distance(p1,p2).kilometers
122     except ValueError:
123         return 10000
124 minpts = int(sys.argv[4])
125 leng = int(sys.argv[3])
126 threshold=int(sys.argv[2])
127 dist = int(sys.argv[1])
128 file = "cluster_{}_mc_{}_min{}_len{}_alg3_D11A.km".format(dist, threshold,minpts,leng)
129 dataset = get_datapoints("frequent_dataset_11A_training_1.1mil.log")
130 core_points = get_corePoints(dataset, threshold)
131 adbc(core_points, dataset, dist,minpts,thoints,file,leng)
```

9.2 Mitigation approach

9.2.1 Bloom filter

```
1  #!/usr/bin/python3
2  import sys
3  from multiprocessing import Process, Queue, Pool
4  from netfilterqueue import NetfilterQueue
5  import socket
6  import _thread
7  import threading
8  import time
9  import queue
10 import struct
11 from collections import deque
12 from bitstring import BitArray
13 from pybloom import ScalableBloomFilter
14 from pybloom import BloomFilter
15 class Bloom:
16     def __init__(self, dataset, error, num):
17         self.f = BloomFilter(capacity=num, error_rate=error)
18         with open(dataset) as yOutput:
19             for line_binary in yOutput:
20                 if len(line_binary) > 1:
21                     self.f.add(line_binary.rstrip())
22         print("Bloom done")
23     def check(self, ip):
24         return ip in self.f
25 def initate_nfqueue(nfqueue):
26     s = socket.fromfd(nfqueue.get_fd(), socket.AF_UNIX, socket.SOCK_STREAM)
```

```
27     try:
28         nfqueue.run_socket(s)
29     except KeyboardInterrupt:
30         print("Socket failed")
31     self.s.close()
32 def packet_queue(pkt):
33     payload = pkt.get_payload()
34     ip = "{}{}{}".format((bin(payload[12])[2:]).zfill(8),(bin(payload[13])[2:]).zfill(8),(bin(payload[14])[2:]).zfill(8))
35 
36     if  bloom.check(ip) == True:
37         pkt.accept()
38     else:
39         pkt.drop()
40 if __name__ == '__main__':
41     bloom = Bloom(sys.argv[1], 0.0001, sys.argv[2])    # The bloom containing the entire data structure.
42 
43     nfqueue = NetfilterQueue()
44     nfqueue.bind(1, packet_queue, max_len=65000, range=20)
45 
46     workers=[]
47     for i in range(4):
48          w = Process(target=initate_nfqueue, args=(nfqueue,))
49          workers.append(w)
50          w.start()
51     for w in workers:
52         w.join()
53     nfqueue.unbind()
```

9.2.2 Binary tree structure

```
1  #!/usr/bin/python3
2  from multiprocessing import Process, Queue, Pool
3  from netfilterqueue import NetfilterQueue
4  import socket
5  from scapy.all import *
6  from scapy.layers import inet
7  import ipaddress
8  import _thread
9  import threading
10 import time
11 import queue
12 import struct
13 from collections import deque
14 from bitstring import BitArray
15 
16 class Node:
```

```
17    """ A node is a value in the tree and contains a reference to the next left and right node """
18    def __init__(self, value):
19        self.left = None  # left  means 0
20        self.right = None # right means 1
21 class Tree:
22    """  The tree containing the data structure."""
23    def __init__(self):
24        self.root = Node(0)
25        self.mode = True
26        self.sum = 0
27    def buildTree(self,dataset):
28        with open(dataset) as sOutput:
29            for line in sOutput: # root - 0 - 0
30                count = 0
31                n = self.root
32                for i in line:
33                    prefix=line[0:(count+1)]
34                    if prefix is not "":
35                        if i is '0':
36                            if n.left is None:
37                                n.left = Node(prefix)
38                            n = n.left
39                        elif i is '1':
40                            if n.right is None:
41                                n.right = Node(prefix)
42                            n = n.right
43                        count = count + 1
44        print("Tree finished")
45    def incr(self, ip):
46        str = ""
47        count=0
48        lth = 24 - len( ip )
49        n = self.root
50        try:
51            #Leading zeros is not acknowledged in the 32 bit binary number. A seperate for loop
52            #Will acknowledge this and go through the leading zeroes. root - 1 - 2 - 3 - 4
53            for i in range(0,lth):
54                n = n.left
55                count=count+1
56                str = str + "0"
57            #Goes through the rest of the IP address(32 Bit).
58            #Needs to be changed for lesser bits(24-31 bit) 1,2
59            for i in ip:
60                if i is '0':
61                    n = n.left
62                    str = str + "0"
63                else:
64                    n = n.right
```

```
65                  str = str + "1"
66              count=count+1
67      except:
68              return False
69      if n is not None:
70          return True
71      else:
72          return False
73  def initate_nfqueue(nfqueue):
74      s = socket.fromfd(nfqueue.get_fd(), socket.AF_UNIX, socket.SOCK_STREAM)
75      try:
76          nfqueue.run_socket(s)
77      except KeyboardInterrupt:
78          print("Socket failed")
79      self.s.close()
80  def packet_queue(pkt):
81      payload = pkt.get_payload()
82      ip = "{}{}{}".format(bin(payload[12])[2:],(bin(payload[13])[2:]).zfill(8),(bin(payload[14])[2:]).zfill(8))
83      if tree.incr(ip) == True:
84          pkt.accept()
85      else:
86          pkt.drop()
87  if __name__ == '__main__':
88      tree = Tree()    # The tree containing the entire data structure.
89      tree.buildTree(sys.argv[1]) # file with 24 bit networks
90
91      nfqueue = NetfilterQueue()
92      nfqueue.bind(1, packet_queue, max_len=65000, range=20)
93
94      workers=[]
95      for i in range(4):
96           w = Process(target=initate_nfqueue, args=(nfqueue,))
97           workers.append(w)
98           w.start()
99      for w in workers:
100         w.join()
101     nfqueue.unbind()
```

Bibliography

[1] Moheeb Abu Rajab et al. "A multifaceted approach to understanding the botnet phenomenon." In: *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM. 2006, pp. 41–52.

[2] Shikha Agrawal and Jitendra Agrawal. "Survey on Anomaly Detection using Data Mining Techniques." In: *Procedia Computer Science* 60 (2015), pp. 708–713.

[3] Marco Ajelli, R Lo Cigno, and Alberto Montresor. "Modeling botnets and epidemic malware." In: *Communications (ICC), 2010 IEEE International Conference on*. IEEE. 2010, pp. 1–5.

[4] Akamai. "Akamai's State of the Internet Report Q1 2016." In: (2016).

[5] Akamai. "s [state of the internet] / security Q1 2015 report." In: (2015).

[6] Akamai. "[state of the internet] / security Q2 2015 report." In: (2015).

[7] Akamai. "[state of the internet] / security Q3 2015 report." In: (2015).

[8] Akami. "s [state of the internet] / security Q3 2016 report." In: (2016).

[9] Akami. "s [state of the internet] / security Q4 2015 report." In: (2015).

[10] Akami. "s [state of the internet] / security Q4 2016 report." In: (2016).

[11] Nahla Ben Amor, Salem Benferhat, and Zied Elouedi. "Naive bayes vs decision trees in intrusion detection systems." In: *Proceedings of the 2004 ACM symposium on Applied computing*. ACM. 2004, pp. 420–424.

[12] Michael Armbrust et al. "A view of cloud computing." In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[13] M Bahrololum and M Khaleghi. "Anomaly intrusion detection system using hierarchical gaussian mixture model." In: *International journal of computer science and network security* 8.8 (2008), pp. 264–271.

[14] Hakem Beitollahi and Geert Deconinck. "Analyzing well-known countermeasures against distributed denial of service attacks." In: *Computer Communications* 35.11 (2012), pp. 1312–1332.

[15] Hakem Beitollahi and Geert Deconinck. "Tackling application-layer DDoS attacks." In: *Procedia Computer Science* 10 (2012), pp. 432–441.

[16] Derya Birant and Alp Kut. "ST-DBSCAN: An algorithm for clustering spatial–temporal data." In: *Data & Knowledge Engineering* 60.1 (2007), pp. 208–221.

[17] Giovanni Bottazzi and Gianluigi Me. "The botnet revenue model." In: *Proceedings of the 7th International Conference on Security of Information and Networks*. ACM. 2014, p. 459.

[18] Paul S Bradley and Usama M Fayyad. "Refining Initial Points for K-Means Clustering." In: *ICML*. Vol. 98. Citeseer. 1998, pp. 91–99.

[19] Andrei Broder and Michael Mitzenmacher. "Network applications of bloom filters: A survey." In: *Internet mathematics* 1.4 (2004), pp. 485–509.

[20] "Building a DDoS-Resilient Architecture with Amazon Web Services." In: (2014).

[21] Olivier Chapelle, Bernhard Schölkopf, Alexander Zien, et al. "Semi-supervised learning." In: (2006).

[22] Daniel Cid. "Analyzing Popular Layer 7 Application DDoS Attacks." In: *blog.sucuri.net* (2015).

[23] Angelo Comazzetto. *Botnets: The dark side of cloud computing*. Tech. rep. Technical Report, Bostan, USA, 2011.

[24] Michael Cusumano. "Cloud computing and SaaS as new computing platforms." In: *Communications of the ACM* 53.4 (2010), pp. 27–29.

[25] David Dagon, Cliff Changchun Zou, and Wenke Lee. "Modeling Botnet Propagation Using Time Zones." In: *NDSS*. Vol. 6. 2006, pp. 2–13.

[26] "Defending Against DDoS Attacks." In: (2015).

[27] Christos Douligeris and Aikaterini Mitrokotsa. "DDoS attacks and defense mechanisms: classification and state-of-the-art." In: *Computer Networks* 44.5 (2004), pp. 643–666.

[28] Levent Ertoz et al. "Detection and summarization of novel network attacks using data mining." In: *Minnesota INtrusion Detection System (MINDS) Technical Report* (2003).

[29] Martin Ester et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *Kdd*. Vol. 96. 34. 1996, pp. 226–231.

[30] Laura Feinstein et al. "Statistical approaches to DDoS attack detection and response." In: *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*. Vol. 1. IEEE. 2003, pp. 303–314.

[31] Markus Goldstein et al. "Bayes optimal ddos mitigation by adaptive history-based ip filtering." In: *Networking, 2008. ICN 2008. Seventh International Conference on*. IEEE. 2008, pp. 174–179.

[32] Markus Goldstein et al. "Server-side Prediction of Source IP Addresses using Density Estimation." In: *Availability, Reliability and Security, 2009. ARES'09. International Conference on*. IEEE. 2009, pp. 82–89.

[33] Nabil Hachem et al. "Botnets: lifecycle and taxonomy." In: *Network and Information Systems Security (SAR-SSI), 2011 Conference on*. IEEE. 2011, pp. 1–8.

[34] John A Hartigan and Manchek A Wong. "Algorithm AS 136: A k-means clustering algorithm." In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979), pp. 100–108.

[35] Ian Graham Jörg Micheel and Nevil Brownlee. "The Auckland data set: an access link observed." In: (2001).

[36] Georgios Kambourakis et al. "A fair solution to dns amplification attacks." In: *Digital Forensics and Incident Analysis, 2007. WDFIA 2007. Second International Workshop on*. IEEE. 2007, pp. 38–47.

[37] Tapas Kanungo et al. "An efficient k-means clustering algorithm: Analysis and implementation." In: *IEEE transactions on pattern analysis and machine intelligence* 24.7 (2002), pp. 881–892.

[38] James M Keller, Michael R Gray, and James A Givens. "A fuzzy k-nearest neighbor algorithm." In: *IEEE transactions on systems, man, and cybernetics* 4 (1985), pp. 580–585.

[39] Alexander Khalimonenko and Oleg Kupreev. "DDoS attacks in Q1 2017." In: *securelist.com* (2017).

[40] Alexander Khalimonenko, Jens Strohschneider, and Oleg Kupreev. "DDoS attacks in Q4 2016." In: *securelist.com* (2017).

[41] Yoohwan Kim et al. "PacketScore: a statistics-based packet filtering scheme against distributed denial-of-service attacks." In: *IEEE transactions on dependable and secure computing* 3.2 (2006), p. 141.

[42] Yoohwan Kim et al. "PacketScore: Statistics-based overload control against distributed denial-of-service attacks." In: *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*. Vol. 4. IEEE. 2004, pp. 2594–2604.

[43] Ron Kohavi. "Scaling Up the Accuracy of Naive-Bayes Classifiers: A Decision-Tree Hybrid." In: *KDD*. Vol. 96. Citeseer. 1996, pp. 202–207.

[44] Sanjeev Kumar. "Smurf-based distributed denial of service (ddos) attack amplification in internet." In: *Internet Monitoring and Protection, 2007. ICIMP 2007. Second International Conference on*. IEEE. 2007, pp. 25–25.

[45] Pat Langley, Wayne Iba, and Kevin Thompson. "An analysis of Bayesian classifiers." In: *Aaai*. Vol. 90. 1992, pp. 223–228.

[46] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *Nature* 521.7553 (2015), pp. 436–444.

[47] Jussipekka Leiwo, Thomas Aura, and Pekka Nikander. "Towards network denial of service resistant protocols." In: *Information Security for Global Information Infrastructures*. Springer, 2000, pp. 301–310.

[48] Kingsly Leung and Christopher Leckie. "Unsupervised anomaly detection in network intrusion detection using clusters." In: *Proceedings of the Twenty-eighth Australasian conference on Computer Science-Volume 38.* Australian Computer Society, Inc. 2005, pp. 333–342.

[49] Qiming Li, Ee-Chien Chang, and Mun Choon Chan. "On the effectiveness of DDoS attacks on statistical filtering." In: *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE.* Vol. 2. IEEE. 2005, pp. 1373–1383.

[50] Aristidis Likas, Nikos Vlassis, and Jakob J Verbeek. "The global k-means clustering algorithm." In: *Pattern recognition* 36.2 (2003), pp. 451–461.

[51] Danielle Liu and Frank Huebner. "Application profiling of ip traffic." In: *Local Computer Networks, 2002. Proceedings. LCN 2002. 27th Annual IEEE Conference on.* IEEE. 2002, pp. 220–229.

[52] Neil Long and Rob Thomas. "Trends in denial of service attack technology." In: *CERT Coordination Center, Summary* (2001).

[53] Wei-Zhou Lu and Shun-Zheng Yu. "An http flooding detection method based on browser behavior." In: *Computational Intelligence and Security, 2006 International Conference on.* Vol. 2. IEEE. 2006, pp. 1151–1154.

[54] Peter Mell and Tim Grance. "The NIST definition of cloud computing." In: (2011).

[55] Michael Mitzenmacher. "Compressed bloom filters." In: *IEEE/ACM transactions on networking* 10.5 (2002), pp. 604–612.

[56] Naive Bayes Model. "Naive Bayes Algorithms." In: ().

[57] David Moore et al. "Inferring internet denial-of-service activity." In: *ACM Transactions on Computer Systems (TOCS)* 24.2 (2006), pp. 115–139.

[58] Seyed Mohammad Mousavi. "Early detection of DDoS attacks in software defined networks controller." In: (2015).

[59] Gerhard Münz, Sa Li, and Georg Carle. "Traffic anomaly detection using k-means clustering." In: *GI/ITG Workshop MMBnet.* 2007.

[60] Neamen Negash and Xiangdong Che. "An Overview of Modern Botnets." In: *Information Security Journal: A Global Perspective* 24.4-6 (2015), pp. 127–132.

[61] NSFOCUS. "NSFOCUS Mid-Year DDoS Threat Report." In: (2013).

[62] Rene Paap. "The rise of multi-vector DDoS attacks." In: (2016).

[63] Mrutyunjaya Panda and Manas Ranjan Patra. "Network intrusion detection using naive bayes." In: *International journal of computer science and network security* 7.12 (2007), pp. 258–263.

[64] Leif E Peterson. "K-nearest neighbor." In: *Scholarpedia* 4.2 (2009), p. 1883.

[65] Matthew Prince. "The DDoS That Knocked Spamhaus Offline (And How We Mitigated It)." In: *Web Log Post* (2013).

[66] Xi Qin, Tongge Xu, and Chao Wang. “DDoS Attack Detection Using Flow Entropy and Clustering Technique.” In: *Computational Intelligence and Security (CIS), 2015 11th International Conference on*. IEEE. 2015, pp. 412–415.

[67] Noureddin Sadawi. “The Apriori Algorithm.” In: (2014).

[68] James Scott and Drew Spaniel. “Rise of the machines: The dyn attack was just a practice run.” In: *icitech.org* (2016).

[69] Tara Seals. “Kaspersky: Criminals make 95% profit on DDoS.” In: *infosecurity-magazine.com* (2017).

[70] “Snort: The Open Source Network Intrusion Detection System.” In: *http://www.snort.org/* ().

[71] Haoyu Song et al. “Fast hash table lookup using extended bloom filter: an aid to network processing.” In: *ACM SIGCOMM Computer Communication Review* 35.4 (2005), pp. 181–192.

[72] Stephen M Specht and Ruby B Lee. “Distributed Denial of Service: Taxonomies of Attacks, Tools, and Countermeasures.” In: *ISCA PDCS*. 2004, pp. 543–550.

[73] V Srihari and R Anitha. “DDoS detection system using wavelet features and semi-supervised learning.” In: *International Symposium on Security in Computing and Communication*. Springer. 2014, pp. 291–303.

[74] Christopher Leckie Tao Peng and Kotagiri Ramamohanarao. “Protection from Distributed Denial of Service Attack Using History-based IP Filtering.” In: (2003).

[75] Rob Thomas. “Managing the Threat of Denial-of-Service Attacks.” In: *CERT Coordination Center* 10 (2001).

[76] Maseng Torleiv. “Communication and Information theory.” In: *Communication and Information theory*. 2017, pp. 97–106.

[77] Verisign. “Verisign distributed denial of service trends report - VOLUME 3, ISSUE 4 – 4TH QUARTER 2016.” In: 4 (2016).

[78] Karan Verma, Halabi Hasbullah, and Ashok Kumar. “An efficient defense method against UDP spoofed flooding traffic of denial of service (DoS) attacks in VANET.” In: *Advance Computing Conference (IACC), 2013 IEEE 3rd International*. IEEE. 2013, pp. 550–555.

[79] Rajagopalan Vijayasarathy, Serugudi Venkataraman Raghavan, and Balaraman Ravindran. “A system approach to network modeling for DDoS detection using a Naive Bayesian classifier.” In: *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)*. IEEE. 2011, pp. 1–10.

[80] FuiFui Wong and Cheng Xiang Tan. “A survey of trends in massive DDoS attacks and cloud-based mitigations.” In: *International Journal of Network Security & Its Applications* 6.3 (2014), p. 57.

[81] Yi Xie and Shun-Zheng Yu. “A large-scale hidden semi-Markov model for anomaly detection on user browsing behaviors.” In: *IEEE/ACM Transactions on Networking (TON)* 17.1 (2009), pp. 54–65.

[82] Yi Xie and Shun-Zheng Yu. “Detecting shrew HTTP flood attacks for flash crowds.” In: *Computational Science–ICCS 2007* (2007), pp. 640–647.

[83] Yi Xie and Shun-Zheng Yu. “Monitoring the application-layer DDoS attacks for popular websites.” In: *IEEE/ACM Transactions on Networking (TON)* 17.1 (2009), pp. 15–25.

[84] Rui Xu and Donald Wunsch. “Survey of clustering algorithms.” In: *IEEE Transactions on neural networks* 16.3 (2005), pp. 645–678.

[85] Takeshi Yatagai, Takamasa Isohara, and Iwao Sasase. “Detection of HTTP-GET flood attack based on analysis of page access behavior.” In: *Communications, Computers and Signal Processing, 2007. PacRim 2007. IEEE Pacific Rim Conference on.* IEEE. 2007, pp. 232–235.

[86] Jie Yu et al. “A detection and offense mechanism to defend against application layer DDoS attacks.” In: *Networking and Services, 2007. ICNS. Third International Conference on.* IEEE. 2007, pp. 54–54.