FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER’S THESIS

Title:	Automatic parameters estimation for DDoS attacks mitigation
Student:	Bc. Filip Křesťan
Supervisor:	Ing. Tomáš Čejka, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Systems and Networks
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2019/20

Instructions

Study the current approaches to computer network monitoring based on network flows (IP Flows).
Study technologies for efficient storage of network traffic information.
Design a system (a set of SW modules) for long-term observation of traffic volume for selected IP subnets.
For each subnet, compute a profile composed of observed traffic statistics (e.g., average and maximal volume).
Using the observed profiles, the system should be able to detect an anomaly, i.e., a significant deviation from the historically observed and computed profiles.
The aim of the system is to inform operators about "normal" level of observed traffic to set up parameters of DDoS mitigation tools during an attack.
Additionally, the system should store a list of sources of "normal" traffic and to identify new sources during an attack.
Evaluate the created system using real flow data from CESNET2 network (provided by the supervisor) merged with simulated DDoS attacks.

References

Will be provided by the supervisor.

prof. Ing. Pavel Tvrdík, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 23, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Automatic Parameters Estimation for DDoS Attacks Mitigation

Bc. Filip Křesťan

Department of Computer Systems
Supervisor: Ing. Tomáš Čejka, Ph.D

May 9, 2019

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Ing. Tomáš Čejka for his patience and support throughout this thesis work. I would also like to thank CESNET a.l.e. for providing measurement data, without which this thesis would not be possible. Last but not least, I would like to thank my significant other, Eliška Vepřková, for her never-ending patience and encouragement. I could never accomplish this without her.

Declaration

In Prague on May 9, 2019 ....

Citation of this thesis

Křesťan, Filip. *Automatic Parameters Estimation for DDoS Attacks Mitigation.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Cílem této práce je poskytnout automaticky generovanou informaci potřebnou pro detekci, diagnostiku a případnou mitigaci DDoS útoků popisující normální provoz v dané počítačové síti. Výsledný síťový profil, extrahovaný z informací o síťových tocích, se skládá z efektivně zakódované množiny entit, které historicky komunikovaly s profilovanou sítí a očekávanými úrovněmi provozu procházející danou sítí.

V první části práce je do širšího kontextu metod DDoS útoků a jejich mitigace zasazena mitigační metoda History-based IP filtering, která je základem navrženého subsystému agregujícího historii komunikace na síti. Další část práce se zabývá různými možnostmi ukládání historie síťové komunikace se zaměřením na paměťovou efektivitu. Na základě získaných informací jsou pro tento účel zvoleny Bloomovy filtry. V následující části je vyhodnocena přesnost několika modelů vhodných pro predikci očekávané úrovně provozu ve sledované síti. Na základě vyhodnocení je vybrán prediktivní model Prophet.

Výsledný informační systém, detailně popsaný ve třetí části, se skládá ze dvou podsystémů, které poskytují obě složky informací o síťovém profilu: škálovatelnou implementaci metody History-based IP filtering užívající Bloomovi filtry jako jediné úložiště dat a prediktivní subsystém využívající model Prophet.

Výsledky měření výkonnosti, popsané v poslední kapitole, ukazují, že implementovaný systém je vhodný i pro nasazení v sítích komunikujících s více než sto miliony odlišnými síťovými entitami, což výrazně převyšuje původní požadavky.

Klíčová slova Mitigace DDoS, History-based IP filtering, Analýza časových řad, Probabilistické datové struktury, Bloomův filtr, Prophet

Abstract

The aim of this thesis is to provide information in a fully automated manner describing the normal operation of a computer network needed for detection, diagnosis and possible mitigation of DDoS attacks. The resulting network profile, extracted from network flow information, consists of an efficiently encoded set of entities which historically communicated with the profiled network and expected levels of traffic flowing through the network.

In the first part, History-based IP filtering, the basis of our historical information subsystem, is introduced and set into a broader context of DDoS attack and mitigation methods. The next part explores various storage options of network communication history with focus on space efficiency. Based on the obtained information, Bloom filters are chosen as the most suitable option. The focus is then shifted towards performance evaluation of forecasting models suitable for prediction of expected levels of traffic on the monitored network. The Prophet forecasting model is selected as the most suitable option due to its precision and robustness.

The resulting information system, described in the third part, is composed of two main subsystems providing the two network profile information components: a novel and scalable implementation of History-based IP filtering using Bloom filters as the sole data storage and a forecasting subsystem using the Prophet model.

The results of a performance measurement, described in the last chapter, show that the implemented system is suitable even for deployments on networks communicating with over a hundred million of distinct network entities which vastly exceeds requirements for its intended deployment.

Keywords DDoS mitigation, History-based IP filtering, Time series forecast, Probabilistic data structures, Bloom filter, Prophet

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Introduction

Distributed Denial of Service (DDoS) attacks are lead with an intent to disrupt services provided by an attack victim or otherwise prevent the normal use of the service for legitimate users by exhausting resources of the target system or even a network connecting this service to the clients.

The DDoS attacks are still among the most prevalent security issues plaguing the Internet today and even with technology advancements made in the last decade, an efficient defense against them proves to be complex and expensive. As shown by Majkowski in [2], this is mostly because there are many different types of attacks, unrelently evolving in reaction to new mitigation techniques each of which can possibly addresses only a small portion of the attack types.

The ever-changing nature and shape of the attacks represents a significant topic of interest of many security researchers and companies, however, the decentralized nature of the Internet suggests, that finding perfect method to distinguish between a legitimate and a malicious traffic is not possible. Therefore, the current approaches to the mitigation of DDoS attacks are based on heuristics.

The primary objective of this thesis is to design a system that provides information in a fully automated manner describing the normal operation of a computer network which can be used for detection, diagnosis and possible mitigation of DDoS attacks. This information, which we call *network profile*, can be split into two parts. The first part gives an information about expected levels of traffic based on its long term observations. The second part of the profile consists of set of entities which historically communicated with the profiled network.

Chapter 1 of this thesis introduces problematics of DDoS attacks and related mitigation techniques. Notably, ideas behind the History-based IP filtering mitigation method, which gives the basis and reasoning for the second part of the *network profile*, are explored in depth. At the end of the chapter, security tools developed by *CESNET a.l.e* which are essential for completing objectives of this thesis are shortly introduced.

In the second chapter, requirements for the *network profile* format and contents are specified, followed by a discussion of operational requirements for the system providing this information. We then explore various storage options of the network communication history with focus on space efficiency end perform evaluation of forecasting models suitable for prediction of expected levels of traffic on the monitored network. The proposed system design and its limitations are described at the end of the chapter.

In the third chapter, we present purpose and implementation overview of each separate module of the information system together with specifics of the designed interfaces. We also discuss some of the more interesting technical details of the solution.

The performance evaluation methodology of each of the modules designed and implemented in this thesis and its results are presented in the last chapter.

Chapter 1

Background Information

In this chapter, we provide a short introduction to a problematics of DDoS attacks and their taxonomy. We then proceed with a summary and main ideas of History-based IP Filtering, the basis of our system.

In the last part of this chapter, we introduce the Network Measurements Analysis (NEMEA) system which we heavily rely on in this thesis.

1.1 DDoS Attacks

The Denial of Service cyber-attack category is lead with an intention to disrupt provided services by an attack victim or prevent the normal use of the service for legitimate users. With the rise of the Internet, a Denial of Service attack has become a synonym for an attack lead through a computer network. A Distributed Denial of Service attack is its subcategory that is executed from many network sources at once.

According to Zargar et al. [3], the reasons for attacking and disrupting the target service are commonly financial gain or revenge, but can be also motivated by ideological or political believes. A trend occurring in the last few years is to divert attention from additional system compromise.

Given the recent attacks reported by GitHub [4] and Cloudflare [2] it is apparent that the problematics of DDoS attacks are fare from resolved and require further attention. As Mirkovic et al. noted in [5], the current state of DDoS protection merely reacts to the new attack techniques and trends as they appear, but fail to address the core reason that enables this kind of attack; the Internet design is optimized to forward packets from source to a destination on best effort basis as efficiently as possible which shifts the complexity to end hosts and exposes them to any misbehaving party. This is further supported by the decentralized nature of the Internet spanning many independent entities each with its set of policies and the resulting complicated control and accountability enforcement. Mirkovic also notes, that the proneness of the Internet to DDoS attacks is given by its overall security. This claim is supported by the relatively recent attacks originating from the Mirai botnet composed of poorly secured IoT devices [6] and Memcrashed amplification attacks using combination of weakness in *memcached* protocol and insecure service configuration [7].

As Mirkovic et al. described in [5], DDoS attack has traditionally two phases. First the attacker needs to recruit the network devices that will participate in the actual DoS attack. This is commonly done by remotely exploiting weaknesses or configuration errors and gaining code execution capabilities on the targeted devices, but can also consist of gaining other ways to manipulate otherwise secure devices to perform undesirable but legitimate action like requesting a network resource or sending a response to a request with spoofed address. The recruited devices, also called *bots*, are then formed into a *botnet* by employing, usually covert, method of communication and command relay between the *bots* and the attacker. The methods of communication are usually designed in a way that makes it exceedingly hard to find other nodes in a *botnet* in case one of the *bots* is discovered.

Once the preparations are in place the actual DDoS attack on a victim specified by the attacker is executed. This is the second stage of the attack and is commonly carried out multiple times on many victims by a single *botnet*.

The DDoS taxonomy introduced in [5] is one possible way to classify DDoS attacks based on exploited weakness, victim type, attack source address validity and its spoofing technique, communication with device in the *botnet*, attack rate dynamics and few other classes describing the *botnet*. As noted by Mirkovic et al., it is usually not sufficient to describe a DDoS attack by one class only and that the introduced taxonomy should serve as a common framework to coarsely classify an attack. However, in this thesis we focus on a flooding type DDoS attacks in wired networked systems which are further classified by Zargar et al. in [3]:

L3/L4 flooding attacks
: Network/transport level DDoS attacks have been mostly launched using Transmission Control Protocol (TCP), User Datagram Protocol (UDP) and Internet Control Message Protocol (ICMP) protocol packets and commonly focus on exhausting network bandwidth or packet processing capacity.

- *Flooding attacks*: This type of attack focuses on exhausting the network bandwidth or packet processing speed by sheer amount of packets sent (e.g. UDP and ICMP flood).
- *Protocol exploitation flooding attacks*: The aim of this type of attack is to exhaust resources by exploiting design feature or error in implementation to exhaust target resources (e.g. TCP SYN flood).
- *Reflection-based flooding attacks*: In this type of attack, attackers send a forged request using the victims source address to a known vulnerable reflector which then replies to the spoofed source. This is an attempt to hide origin of the attack. A typical example is Smurf attack which uses ICMP echo requests.
- *Amplification-based flooding attacks*: This type of attack is commonly combined with reflection flooding attacks by leveraging a services to amplify the amount of traffic generated by the service response.

L7 flooding attacks

Application-level DDoS attacks consume less bandwidth in general and are thus stealthier from a L3/L4 point of view than volumetric attacks since they are very similar to legitimate traffic, but usually have the same impact on the services since they target specific characteristics of application level protocols. However, the basic ideas remains the same as for L3/L4 attack types mentioned above only on application layer protocols. A notable additions to the types are:

- *Asymmetric attacks*: This attack consists of forging a request resulting in intensive computation on the victim side. This is typically an expensive database query or HTTP multiple VERB single request.
- *Slow request/response attacks*: This attack consumes resources on a victim side by keeping large number of application sessions open by slow updates. A prominent example is Slowloris HTTP attack.

In [5] Mirkovic et al. also classifies DDoS defense strategies. The most notable strategies listed are filtering and rate limiting which are commonly employed for L3/L4 and L7 attacks respectively. Zargar et al. in [3] further extends this taxonomy by the deployment location of a defense technique to Source, Destination, Network and Hybrid approaches respective of an attack origin. As discussed in the research, detection is usually most accurate close to a destination of an DDoS attack, however mitigation is better deployed close to the sources of an attack since it is possible that the bandwidth is depleted on a transit network even before reaching the victim given high enough attack volume.

Focusing on the defense methods deployed at the destination of DDoS attack, various mechanisms that can take place either at the edge routers or the access routers of the destination Autonomous System are further classified in [3] into multiple categories:

IP Traceback mechanisms
In this mechanism the source of a forget packet is traced back to its true origin. It usually relies on packet marking mechanisms. As Zargar notes, this method has a significant operational and deployment challenges since a non trivial number of routers that support this marking is required for this method to be effective.

Management Information Base
This method consists of building a database of packet and routing statistics which is later used to detect anomalies and help identify when a DDoS attack is occurring and provide information for mitigation system reconfiguration.

Packet marking and filtering mechanisms
These mechanisms aim to mark legitimate packets based on some heuristic. This packet marking can be later used for filtering or given higher preference in case of DDoS attack by dynamic filters installed on the victims network edge.

Packet dropping based on the level of congestion
As the name suggests, these defense mechanisms drop suspicious or undesirable packets so that desired levels of congestion on a network links are met.

The *History-based IP filtering* is one of the mechanisms classified in *Packet marking and filtering mechanisms* category and is described in detail in the next section.

1.2 History-based IP Filtering

This thesis partially builds on ideas presented in [8] and heavily relies on the published results which introduces novel and robust solution to DDoS attack mitigation which can be used on most types of current internet-facing computer networks.

The main idea of history-based IP filtering presented by Tao Peng et al. in [8] is that most IP addresses communicating with a certain network does so repeatedly and fairly frequently. The intuition behind this is that people tend to visit services present on the network on a regular basis and that this holds to a high degree (around 82.9%, [9]) even for cases of flash crowd (i.e. significantly more than usual amount of legitimate users access one particular service at the same time) as demonstrated by Jung in [9]. A prime and the most relatable examples of such a service would be a web based news outlet or a company information system.

To leverage this property, Peng proposes to store a history of entities which commonly communicate with the protected network outside of DDoS in a database together with a timestamp and exchanged amount of packets. A sliding window of most frequent IP addresses which regularly had a valid communication with the network is kept in the database and later used as a white list of entities which are allowed to communicate with the protected network during a DDoS attack. By changing a size of the window and other parameters like the required minimum number of packets exchanged in a single communication flow for it to be considered valid a balance between precision and the database size can be established to fit a specific needs of the network operators.

The proposed scheme has two modes of operation for the system: a normal state, during which new addresses are inserted into the history database and a state when the edge router becomes overloaded due to DDoS attack during which the inbound packets are being dropped according to the learned history (i.e. the database is used as a white-list) and no new records are inserted into the database.

In the experiments performed in [8], a two week sliding window was kept in a database and flows consisting of less than three packets were ignored since these were most likely network scans or a reply from victim to a request with spoofed source IP address. The tests described in the original work performed on various datasets from real networks shows that about 88-90% of IP addresses that were observed over the period of two weeks also appeared in the next week. This of course supports the idea behind the proposed filtering scheme.

While the History-based IP Filtering alone can be very precise and effective, as demonstrated in the original white paper, it can also be relatively easily circumvented since there are types of attacks for which this type of mitigation fails pathologically. The common denominator is that the source addresses are either spoofed to match that of the valid clients or the attacking addresses had historically communicated with the protected network either as a preparation for an attack or the communication originated in an actual valid need (e.g. networks with open Domain Name System (DNS) resolvers). Thus it is not recommendable to use this method as a sole DDoS mitigation strategy but rather combine it with other methods such as *RepTopN* heuristic presented in [10].

1.3 Network Flow Monitoring

Network flow monitoring is similarly to a packet capture a passive network monitoring approach. However, in flow export, packets are aggregated into flows. In RFC 7011 [12] *network flow* is defined as “a set of IP packets passing an observation point in the network during a certain time interval, such that all packets belonging to a particular flow have a set of common properties”. According to Hofstede et al. [11], these common properties usually include source and destination IP addresses and port numbers, but can be also any other packet header fields, meta-information and even packet contents.

Flow monitoring has several advantages over packet capture which makes it more suitable for deployment on high-speed networks. Most notably, because of the aggregation, flow monitoring is considered to be more scalable than traditional packet-based traffic analysis. Also, its wide deployment is much more feasible since it is commonly supported by network hardware while packet capture requires expensive equipment and substantial infrastructure for storage and analysis. The storage requirements alone can be several orders of magnitude lower compared to packet capture due to the aggregation. [11]

The Flow monitoring architecture described by Hofstede et al. consists of several stages:

- *Flow exporter*: This is usually a packet forwarding device which reads packets from a network line and aggregates them into *network flows*. Flows which are considered to be terminated are then exported to one or more flow collectors using a standardized protocol.

- *Flow collector*: Commonly a software which receives the exported flow records, optionally performs pre-processing like aggregation, filtering and data compression and stores the records for later use. Example of flow collector software is IPFIXcol [28] developed by *CESNET a.l.e.*

- *Analysis application*: Analyzes flow data stored by flow collector. This can be either manual or a fully automatic process. The NEMEA system described in the next chapter is an example of such application.

One of the most prominent protocols used to transfer flow records from exporters to a flow collectors is IP Flow Information Export (IPFIX), which is described in [12] as “a unidirectional, transport-independent protocol with flexible data representation”, however we omit its detailed description since it is not used directly in this thesis.

1.3.1 NEMEA System

One of the tools heavily used in this thesis is NEMEA:

> NEMEA (Network Measurements Analysis) system is a stream-wise, flow-based and modular detection system for network traffic analysis. It consists of many independent modules which are interconnected via communication interfaces and each of the modules has its own task. Communication between modules is done by message passing where the messages contain flow records, alerts, some statistics or preprocessed data. [13]

As per the flow monitoring architecture in previous section, NEMEA can be classified as an automated analysis application.

Multiple ingest formats and sources are supported by NEMEA, however it uses its own Unified Record (UniRec) data format for efficient communication between module instances. Each instance of a NEMEA module runs as a separate process. The communication between modules is defined via application level interfaces each being either TCP socket, UNIX socket, file or a blackhole which drops all traffic. Together, this allows forming of complex multistage record processing pipelines as shown later in this thesis.

The project is developed and maintained by CESNET a.l.e. under dual GPL-2.0 [14] or the permissive BSD 3-Clause license [15].

NEMEA is currently deployed on the CESNET2 network which interconnects main university cities of the Czech Republic and other sites and from which we use anonymized data to design and verify multiple components of this thesis.

In this thesis, we rely on several software projects and libraries, however NEMEA is the most prominent since it is used to gather and manipulate network flow records and we also contribute two new modules to the project as a part of this thesis. The contributed modules are described in chapter 3.

Chapter 2

Analysis and Design

This chapter presents a set of requirements for the designed system and the resulting *network profile*, discuss possible solutions for the storage of history about entities communicating with the protected computer network, compare several forecasting models and evaluate their performance and suitability for our problem of generating expected traffic levels flowing to the protected network.

In the last part of this chapter, we discuss the results of the evaluations and conclude with a high level design of the proposed system.

2.1 Requirements Analysis

During the design of this information system a few requirements have emerged. The common requirement for all parts of the system is for it to be capable of handling information for multiple separate networks in one single deployment (a way of multitenancy). This is mainly done in order to lower the complexity of deployment but also to save some computation resources.

In section 1.2, we have described a History-based IP Filtering scheme which uses white lists of allowed IP addresses. The implementation of this scheme leads to a problem of responding to simple set membership queries where the set is bound to a certain time range. In other words, the resulting system has to provide a way to query whether a certain IP address is within a white list which itself is limited to a certain date range.

Also, the system for gathering of historical information about network entities has a performance requirement for it to be able to digest at least 400 thousand flow records per second. This requirement originates from CESNET2 network where this is the current observed maximum during peak hours and where the prototype of this system should be deployed. Though not explicit, it is paramount that the resulting information should be possible to use for packet filtering on the edge of the protected network and thus as fast as possible.

There is only one hard requirement for the second part of the system which is that the information about expected levels of traffic includes lower and upper bounds for triggering of a packet filtering mechanism on the edge of the protected network. The other, somewhat soft and vague, requirement is for the forecast be robust.

2.2 Evaluation of Efficient Storage of Network Communication History

There is multitude of ways to store network entity identifying information (i.e. an IP address) and time when the communication happened. For example this information could be stored in a relational database, however we do not need a set of strong grantees provided by this type of database for our use-case. Limiting focus on the features required a key-value store seems like a more viable solution since it typically provides higher transactional throughput than traditional relational databases while still being flexible enough to model out data. If we further limit the required functionality to a bare minimum, a class of probabilistic data structures that in exchange for absolute certainty are fast, extremely space efficient and can answer the set membership queries in constant time emerges as an interesting alternative.

In this section, we describe and compare three such probabilistic data structures that fit our requirements.

2.2.1 Bloom Filter

Bloom filter, first introduced by Burton H. Bloom in [16], is a space-efficient probabilistic data structure that supports some of the common set operations with constant time complexity: insertion and element membership query. However the membership query can return false positives with configurable probability. The false negatives are not possible. In other words the result of membership query is either that an element is possibly in or that it is definitely not in a set.

Another aspect of Bloom filters is that, in its basic form, it does not support element retrieval or deletion. This means that it is not possible to retrieve all elements from the set unless they are from a finite field (in which case we would need to enumerate all of the field elements).

The data structure consists of a bit array of size m and a k *different* hash functions h. Each of the hash functions maps a set element to exactly one bit in the array. Positions of the k bits for an element in the array must be uniformly distributed.

Initially, when the filter is empty all bits of the array are set to 0. To insert an element, it is hashed by the k hash functions and bits corresponding to the k hashes in the bit array are set to 1. To check if an element is in the bit array a same hashing operation is performed and if all of the corresponding bits in the array are 1, the element is *likely* in the set. If at least one of the bits is 0, then the element is *definitely* not present. This process can be seen on figure 2.1.

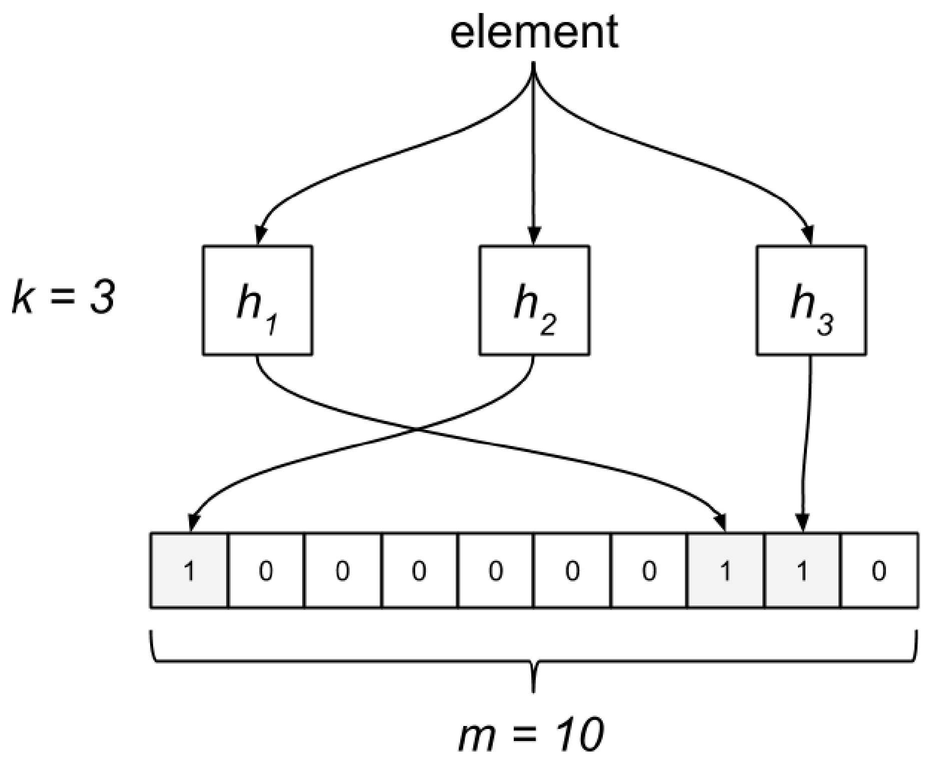

Figure 2.1: Insertion of an element to a Bloom filter (source: [17])

Regardless of the size of data being inserted, Bloom filter uses only about 10 bits per element at 1% false positive probability, as noted in [18], since the inserted element is being hashed into the bit array. This is much more efficient compared to a naive method where a full IP address has to be stored. Furthermore, this feature allows for experimentation with more complex identifiers of connection with the remote network entity (e.g. source and destination IP address tuple).

Given that the hash functions are perfectly random, the probability of a false positive in a Bloom filter where n is the number of elements encoded in the filter [19] is:

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-kn/m}\right)^{k}$$

In [19], Border et al. also derives the optimal number of hash functions:

$$k_{opt} = \ln 2 \left(\frac{m}{n}\right)$$

The optimal size of the bit array is given by Tarakoma et al. in [20]:

$$m_{opt} = -\frac{n \ln p}{(\ln 2)^2}$$

Given the formulas above, we can construct an optimal Bloom filter from just the expected number of unique IP addresses and desired false positive rate.

Another interesting property that needs to be kept in mind when working with Bloom filters is that the false positive rate grows with the number of elements inserted from 0% to the designed error rate. However, if more than the amount of elements for which the filter was designed is inserted, the false positive rate keeps growing over the desired value. This eventually reaches state where all the bits in the underlying array are set to 1 and the false positive error rate is 100%. However, as Swamidass in [21] shows, even if we don't know the actual count of inserted elements, we can approximate it (and thus the error rate) using the bit array cardinality X:

$$\hat{n} = -\frac{m}{k} \ln \left(1 - \frac{X}{m}\right)$$

The last interesting property of Bloom filters for our application is that it is possible to create an union of two Bloom filters without a loss of any information simply by computing bitwise OR of their underlying bit arrays [20]. The resulting array will be the same as if it was created by inserting elements encoded in both filters one by one into the new filter. The condition here is that the number and type of hashing functions and bit array size used are the same for the resulting Bloom filter and both of filters for which we are computing the union.

Given the long history of Bloom filters, there exists a lot of libraries implementing this data structure. We liked [22] in particular since it is extremely fast and straight to use and distributed under the 2-Clause BSD license [23]. The speed of this library is mostly because it implements ideas presented by Kirsch er al. in [24]. In summary, the most computationally expensive part of inserting an element into a Bloom filter is producing the k hashes. The proposed improvement is to use constriction with two hash functions only for the first bit position and derive the remaining $k-1$ from this by single multiplication and modulo operation which is significantly less expensive than computing even the fastest hash functions which fulfill the distribution requirements. While in theory this has negative impact on the uniformity of distribution of the resulting bit positions, the practical impact is negligible as shown in the research.

2.2.2 Cuckoo Filter

Cuckoo filter, described in [1] by Fan, Andersen, Kaminsky, and Mitzenmacher, is very similar to Bloom filter, in a way that it also space efficient probabilistic data structure which supports fast set membership testing and the result of a membership query can be also a false positive. They build on the ideas of Cuckoo hashing and while maintaining similar space complexity, Cuckoo filters improve upon the design of Bloom filter by offering deletion, limited counting, and a bounded false positive probability.

The Cuckoo filter consists of an array of buckets. Each of the buckets can hold b small f-bit fingerprints. The value of f is computed based on the required ideal false positive probability required for the usage when designing the filter.

Having bound false positive rate, means that, similarly to Bloom filters, false positive rate steadily increases with load of the data structure, but, in contrast to Bloom filters, Cuckoo filters never exceeds its designed false positive rate. A Cuckoo filter load is a proportion of non-empty slots to empty slots.

In Cuckoo hashing, each element is hashed by two different hash functions so that it can be inserted into one of two buckets. The element is placed in a first empty slot found. However, if both buckets are full a conflicting record has to be evicted to its alternative position for the insertion to finish successfully. The evictions can happen recursively, but it is typically limited to a several iterations to guarantee constant time complexity. When the last iteration has conflicting pair of elements a complete removal of one of the conflicting elements has to happen or the operation fails. However, as stated in the original publication, this happens rarely and increases only with the Cuckoo filter load. The exact algorithm for element insertion described in [1] can be seen in algorithm 1. The first part of the insert algorithm is also common for both search and delete operations.

```
$f$ = fingerprint($x$);
$i_1$ = hash($x$);
$i_2$ = $i_1 \oplus$ hash($f$);
**if** *bucket[i1] or bucket[i2] has an empty entry* **then**
    add $f$ to that bucket;
    **return** Done;
**end**
// must relocate existing items
$i$ = randomly pick $i_1$ or $i_2$;
**for** *n=0; n < MaxNumKicks; n++* **do**
    randomly select an entry $e$ from bucket[$i$];
    swap $f$ and the fingerprint stored in entry $e$;
    $i$ = $i \oplus$ hash($f$);
    **if** *bucket[i] has an empty entry* **then**
        add $f$ to bucket[$i$];
        **return** Done;
    **end**
**end**
// Hashtable is considered full
**return** Failure;
```

Algorithm 1: Element insertion into Cuckoo filter (source: [1])

As can be seen on algorithm 1, it is simple to compute the location of the other bucket. The downside of this scheme is that given an f-bit fingerprint, the second bucket is chosen from 2^f possible locations and is thus not com<pletely random for small fingerprints. Despite having theoretically much more collisions than Bloom filters, empirical analysis performed in [1] has shown, that for $f = 7$ the load factor of the Cuckoo filter mirrored that of a Cuckoo hash table with two perfectly random hash functions.

Cuckoo filter, in contrast to Bloom filter, in its basic form also supports limited entry counting. It is achieved simply by inserting the f-bit fingerprint into a multiple fields in the assigned buckets. The maximum of the counter is then decided by the number of positions the fingerprint can be in.

Deletion works by removing all of the matching fingerprints from its possible locations if counting is allowed. In case of counter decrement, only one of the fingerprints is removed.

The search and deletion time complexity for Cuckoo filters and amortized time complexity for insertion are all $\mathcal{O}(1)$ [1].

It seems to be possible to perform a set union on two Cuckoo filters constructed with the same parameters. The idea is to iterate over the fields in all of the buckets from both filters and insert the fingerprints one by one effectively skipping the initialization phase in 1. Compared to Bloom filter merging, this is significantly less straight forward and more computationally intensive. Another issue is that the union of two filters might not be simply possible in some cases due to exceeded capacity of the resulting filter.

Given that the Cuckoo filters were introduced only in 2014, there is much less libraries readily with varying degree of quality. Nevertheless, the implementation included in the NEMEA Framework project [25] is sufficient.

2.2.3 Quotient Filter

Quotient filters as Bloom or Cuckoo filters are space efficient probabilistic set with feature set comparable to that of Cuckoo filters.

An f-bit fingerprint is computed for each element and split into a r bit reminder and $q = f - r$ bit quotient. The quotient is used as an offset to an array of m slots each consisting of the r bit reminder and three bits used to signalize the state of the slot and its associate elements. Each element has a primary position given by its quotient; canonical slot. When a slot is empty the reminder is simply written to the slot and appropriate bits set. However when a canonical slot is already occupied the reminder is stored in some slot to the right. The insertion algorithm ensures that elements belonging to the same canonical slot are stored in contiguous slots called a run. It is not guaranteed that the run begins at its canonical slot. A cluster is a contiguous runs which starts at its canonical slot and is either terminated by empty slot or by a start of another cluster.

The bits attached to each slot are used to denote additional information about the elements. Whether the slot is canonical for some element in the filter, whether the slot is the first reminder on a run and whether the reminder is shifted from its canonical slot. The meaning also varies based on their combination and not all possible bit combinations are used. A set of complex rules is then applied to perform a lookup until a slot with matching reminder is found or no other possible positions remain.

The lookup and insert operations gets increasingly expensive with the growing size of the clusters. Bender et al. in [26] argues that if the hash function used to generate the fingerprint is uniformly distributed, then the length of most runs is very likely in $\mathcal{O}(logm)$.

The main downside of Quotient filter is that it requires noticeably more space compared to Bloom or Cuckoo filters. The main advantage over Bloom filters is that only one hash function needs to be computed, however, this advantage is diminished by Bloom filter performance optimizations such as [24].

2.3 Evaluation of Forecasting Models

One of the requirements for the *network profile*, is that it includes expected levels of traffic. In this section we perform a short evaluation of three forecasting models on data gathered from the NEMEA2 network.

We have chosen byte count, packet count and number of connections flowing to the protected networks as the forecasted metrics since it is readily available from network probes located in the CESNET2 network exported as IPFIX records [27] to IPFIXcol flow collector [28] which can be then further processed by the NEMEA system introduced in the first chapter of this thesis.

The main reason for using flow records is that it is significantly more compact in comparison to more traditional packet dump, thus allowing for monitoring of bigger computer networks. However, one downside of flow-based monitoring, is that in order to keep information in the flow record accurate, the flow record is produced only after the connection has ended or after a set maximum connection duration. This has an interesting drawback in that a long-lived connections are not visible in the flow records until the configured maximum connection length runs out. This is the main reason for the choice of metric and its interval used in this comparison and in the resulting network profile.

For reasons outlined above, we've decided to use hourly sum of bytes, packets and number of distinct flows for the forecast model evaluation to account for the 5 minute maximum connection duration configured on the NEMEA2 flow exporters. This of course means, that the prospective implementations of the filtering system using this network profile information will have to do appropriate approximations.

The data used in this comparison were collected from CESNET2 and aggregated using the NEMEA aggregation module. Figure 2.2 shows a diagram of the metric collection system architecture. Configuration for the NEMEA pipeline is available on the media enclosed to this thesis.

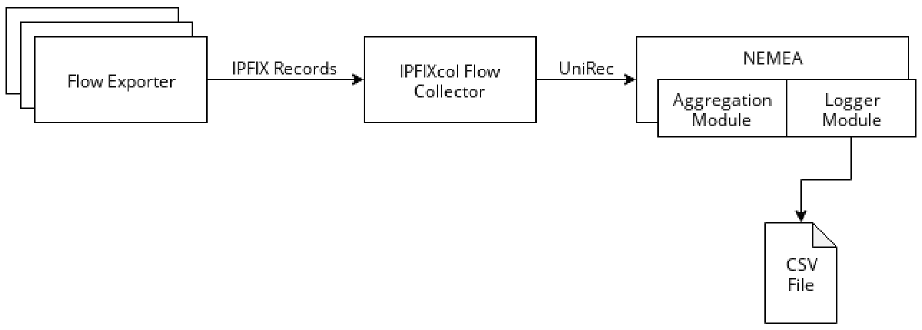

Figure 2.2: Diagram of the metric collection system architecture

2.3.1 Evaluation Method

When evaluating models, it is common to split the data to a training and test portions. The training data is used to estimate parameters of a forecasting model and the test data is then used to evaluate its accuracy to minimize any bias and over-fitting of the model on particular data. However, time series can't be split randomly to two sets of values as the values in time series are typically not independent. The spit is done at a certain point in the time where a number of data points prior to this splitting point are used as a training set and data points after the splitting point are used as a testing set. [29]

To further increase the reliability of results we use a cross-validation modified for time series evaluation described in [29], where an average over rolling series of training and test sets is computed. This is depicted on figure 2.3, where the red observations are form the test sets, blue observations are form the training sets and gray observations are not used for the current step. As can be seen in the figure, a number of the earliest observations are not considered as test sets, since it is not possible to obtain a reliable forecast based on a small training set. This method is also known as “evaluation on a rolling forecasting origin” [29].

To compare the models, we use forecast error e which is difference between observed y and forecasted value $\hat{y}$. Given training data $y_1, \ldots, y_T$ and test data $y_{T+1}, y_{T+2}, \ldots,$ the error is given by:

$$e_{T+h} = y_{T+h} - \hat{y}_{T+h|T}$$

We then measure the forecast accuracy by summarizing the errors to three metrics: mean absolute error (MAE), root mean squared error (RMSE)

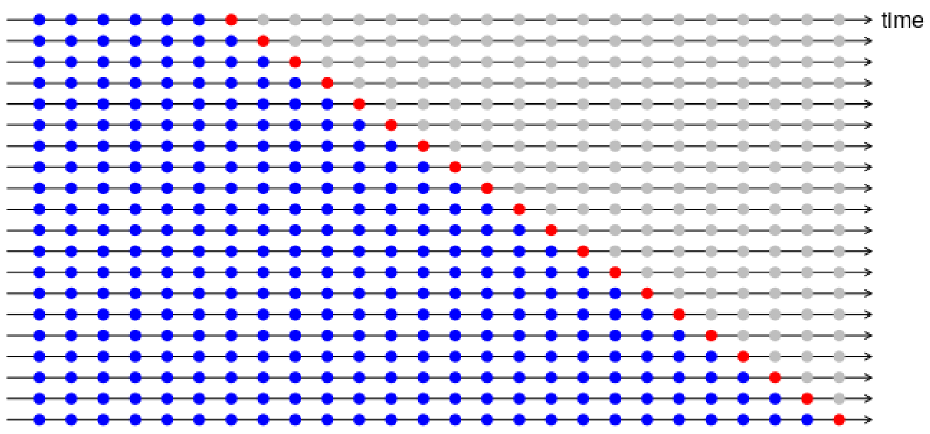

Figure 2.3: Time series cross-validation (source: [30])

and mean absolute percentage error (MAPE):

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|e_{t_i}|$$

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}e_{t_i}^2}$$

$$MAPE = \frac{100\%}{n}\sum_{i=1}^{n}\frac{e_{t_i}}{y_{t_i}}$$

We include both MAE and RMSE, because forecast method that minimises the former will lead to forecasts of the median, while minimising the later will lead to forecasts of the mean. Also, RMSE is more punitive to larger errors, which is usually desirable. On the other hand MAE is more easily understood as it is not scaled or skewed. [29]

In order to make the profile more granular, we've also decided to split the data by *Protocol* field in the IPv4 [31] header and *Next Header* in the IPv6 header [32]. This field gives information about how to interpret data contained in current packet.

The data we are using for the model evaluation were gathered over a period of four months from the CESNET2 network. Since the data are collected from an actual production network, it also contains anomalies. For example, when ICMP data is selected, it contains what appears to be a port scan of the whole network. This is depicted on fig 2.4. However, we do not want to forecast any of the anomalies. To fix this, we've manually removed the anomalous parts of the data and then filled the newly created gaps by data points approximated by a linear function.

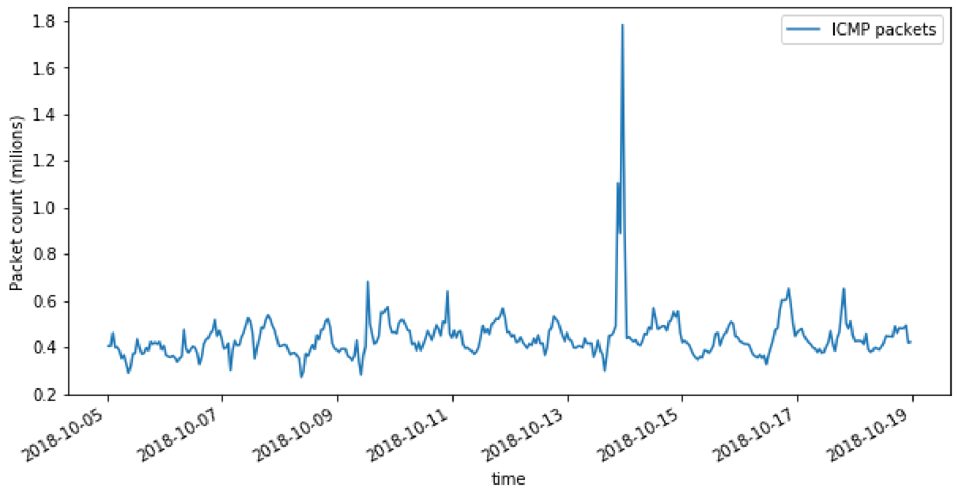

Figure 2.4: Example of anomaly in ICMP data used for model evaluation

In the requirements section, we specify that the forecast needs to be robust. To include this criteria in our model evaluation, we use the unmodified data for training and cleaned-up data for testing. This has a two-fold effect. First, any influence on a model by the anomalies will result in higher forecast error. Second, this further decreases error for models that forecast data points closer to what we consider correct.

With respect to the discussion above, we've chosen the cross validation parameters to have at least 500 data points, split the dataset into 40 folds and forecast 24 hour ahead. However, we've decided to evaluate each hour as a separate forecast.

The anonymized data and evaluation computations in form of Jupyter notebook [33] are available on the medium enclosed to this thesis.

2.3.2 Evaluation Models

For the expected levels of traffic of the network profile, we've decided to use forecast models using history of the given metric to make a forecast.

The first model, which we call Last Value, simply repeats the last seen value of the forecasted metric. We include this model since it is commonly used as baseline for comparison of more complex models.

The second model in comparison is Linear Regression. For the computation we use implementation available from scikit-learn [34], which uses

Ordinary Least Squares (OLS). OLS has a closed form solution and its computation can be a potential problem for large data sets. This was not the case on the testing data set however.

The last model we used is Prophet which is both a forecasting model presented by Tayloer et al. in [35] and a forecasting toolkit developed and distributed under the 2-Clause BSD License [23] by Facebook.

> Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well. [36]

Prophet uses composite model with three components: trend $g(t)$, seasonality $s(t)$, and holidays $h(t)$:

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t.$$

The trend function models non-periodic changes. In Prophet this can be modeled either by a logistic saturating growth model or a linear function with changepoints. The changepoints are any growth-altering events and can be either selected manually or automatically. In our case, there are strong changepoints visible in our test data at the start of academic year so we've decided to use the linear model.

Seasonality represents periodic changes. Time series, as is the case of our testing data, often have multi-period seasonality as a result of the human behavior. This means that we can observe periodic changes within the day due to normal human daily cycle, but also a changes with weekly period which are effect of the work week. Sometimes even a changes with monthly period occur. An good example is a network traffic in an accounting company where the end of a moth is typically much busier. However this is not the case with our testing data. Also, since we do not have enough gathered data, yearly seasonality wasn't used either. In Prophet, seasonality is modeled by a Fourier series. One consequence of this model is that Prophet isn't a good fit for machine generated data (as opposed to data dependent on human behavior). An example would be stock-exchange due to high frequency algorithmic trading.

Prophet also supports multiplicative and additive model of seasonality. In short the difference is whether seasonality component is added to or the seasonal effect is a factor that multiplies the trend component. We've decided to use the multiplicative model.

Holidays and other events take a special place as those occur on potentially irregular schedules. However, the impact of the holiday is often similar each year. In the model evaluation Jupyter notebook enclosed to this thesis, we provide a list of holidays relevant to our test data, but this has no real effect given that the span of the evaluation data is less than a year.

Lastly, changes which are not accommodated by the model are represented by the error term ϵ_t. A normal distribution is assumed.

2.3.3 Evaluation Results

In this part we present and discuss the results of the forecast model evaluation. Since the results were very similar for TCP, UDP and ICMP we've decided to present only TCP results and include only UDP or ICMP results where there are notable differences. The full results of the evaluation are available in the Jupyter notebook on the media enclosed to this thesis.

All of the figures in this section show a hour by hour error on the 24 hour forecast starting at the next hour forecast and ending 24-hour prediction.

On figures 2.5 and 2.6, depicting a comparison of the models on TCP byte and packet count respectively, we can clearly see the error values on Last Value metric rise until roughly 12 hour forecast and then decrease until the 24 hour forecast. This is not surprising, given that the last known value is simply repeated and given that the daily seasonality closely resembles a sine wave with period of 24 hours. However, Last Value does not account for overall trend in the data and thus the error at the end of the 24 hour forecast period is higher than at the start.

The errors of Prophet model on figures 2.5 and 2.6 are clearly smaller than the two other models. There is slight error increase as the forecasted hour is more distant from available data points. We believe, that this is caused by uncertainty in the trend model component.

The Linear Regression model error on figures 2.5 and 2.6 is more or less constant. This is also not surprising given that the model only fits an overall trend in the evaluation data.

Surprisingly, Last Value model did yield better forecast in case of flow count metric as can be seen on figure 2.7. However, looking at the raw data of the flow metric, the flow count shows much weaker seasonality as the remaining metrics. hAlso, the relatively smaller amplitude helps the Last Value model in this case. An example cut out of the evaluation data comparing the seasonality of TCP flow count and bytes is shown on figure 2.8. Similar issue also affects all ICMP metrics. We believe that, in this case however, the lack of strong seasonality is caused by the fact, that ICMP

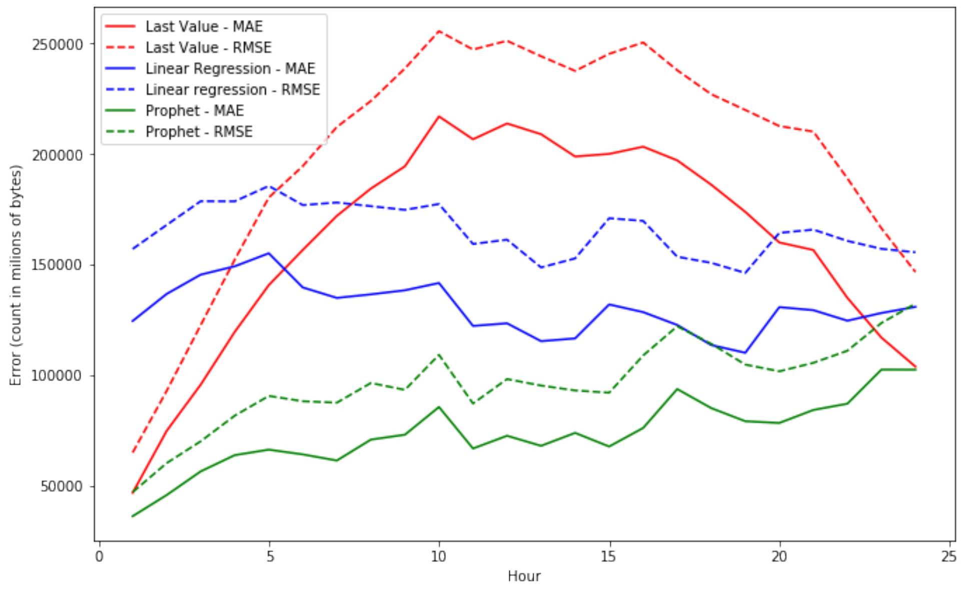

Figure 2.5: Comparison of model MAE and RMSE on 24 hour forecast using TCP byte count metric

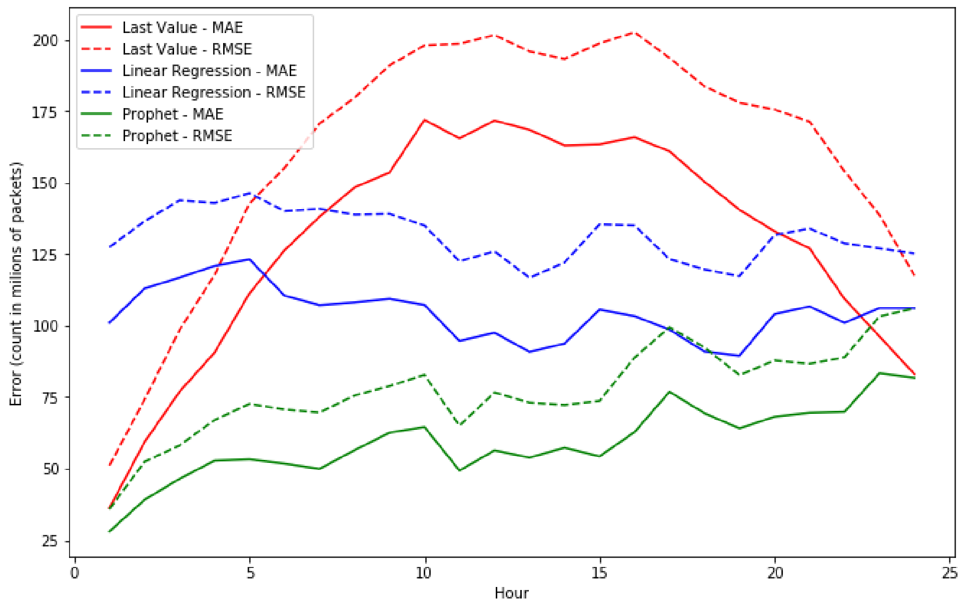

Figure 2.6: Comparison of model MAE and RMSE on 24 hour forecast using TCP packet count metric

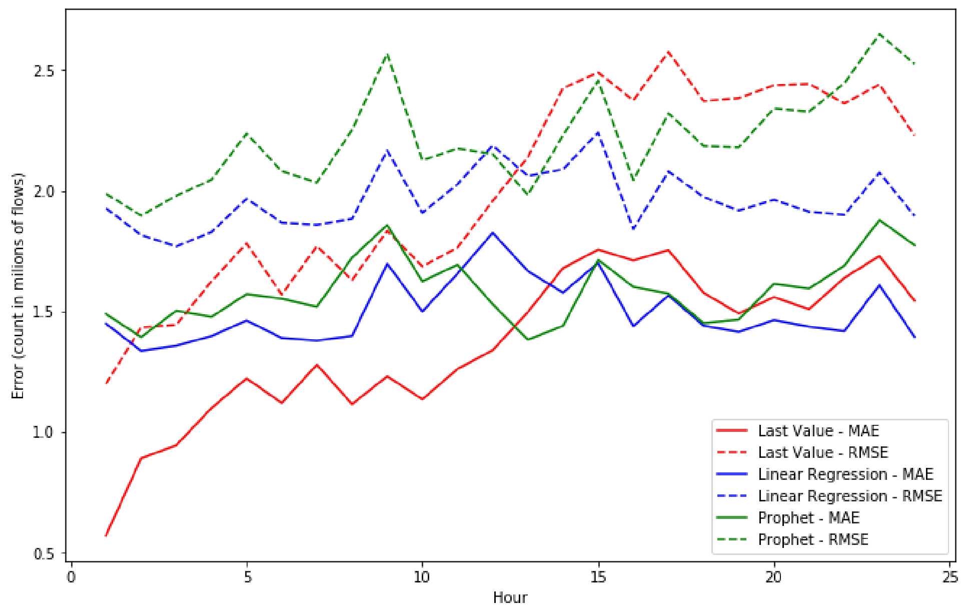

Figure 2.7: Comparison of model MAE and RMSE on 24 hour forecast using TCP flow count metric

messages are not as dependent on human activity but are rather machine generated.

Figures 2.9, 2.10 and 2.11 show a relative error in the models. Overall errors in Prophet are under 40%. Also, in comparison to the Linear Regression model, Prophet errors do not fluctuate as much and remain relatively stable. The two remaining model error even exceed 100% in some cases.

As discussed above, there are some concerns regarding the seasonality of some of the TCP flow count and all ICMP metrics. As shown on figure 2.11, the errors are actually much smaller than for the other metrics. This is also the case for ICMP.

In summary, the Prophet forecast model yields the best results for most of the evaluated metrics. In the remaining cases the errors are reasonably small for it to still be considered useful.

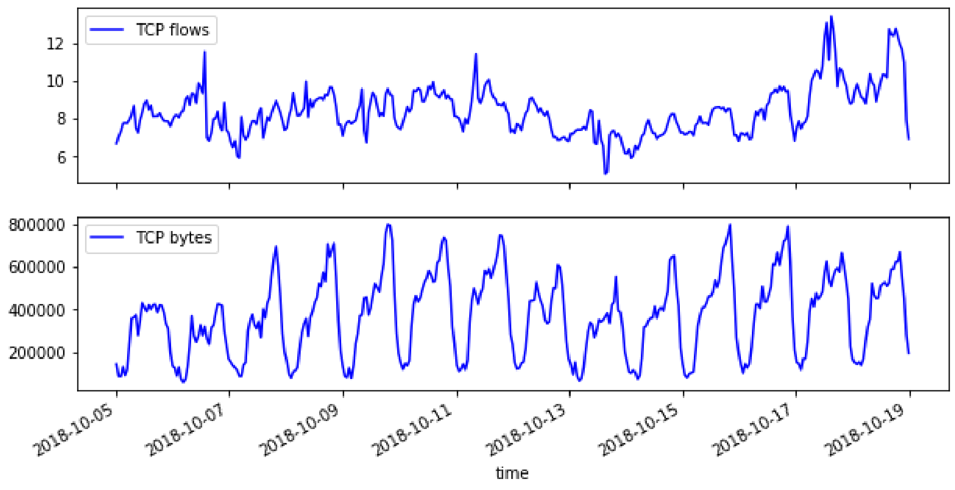

Figure 2.8: Seasonality comparison of TCP flow and bytes count metrics

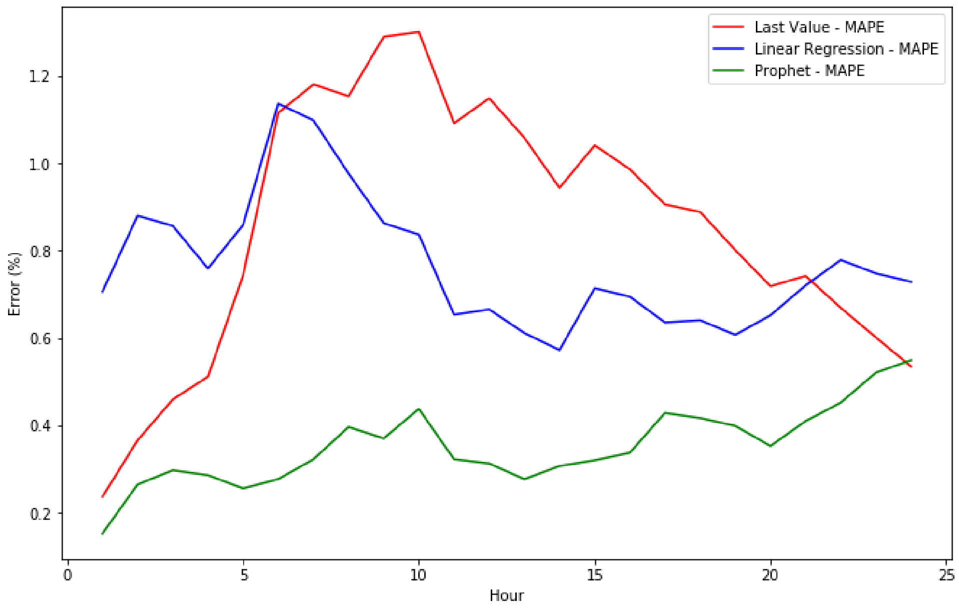

Figure 2.9: Comparison of model MAPE on 24 hour forecast using TCP byte count metric

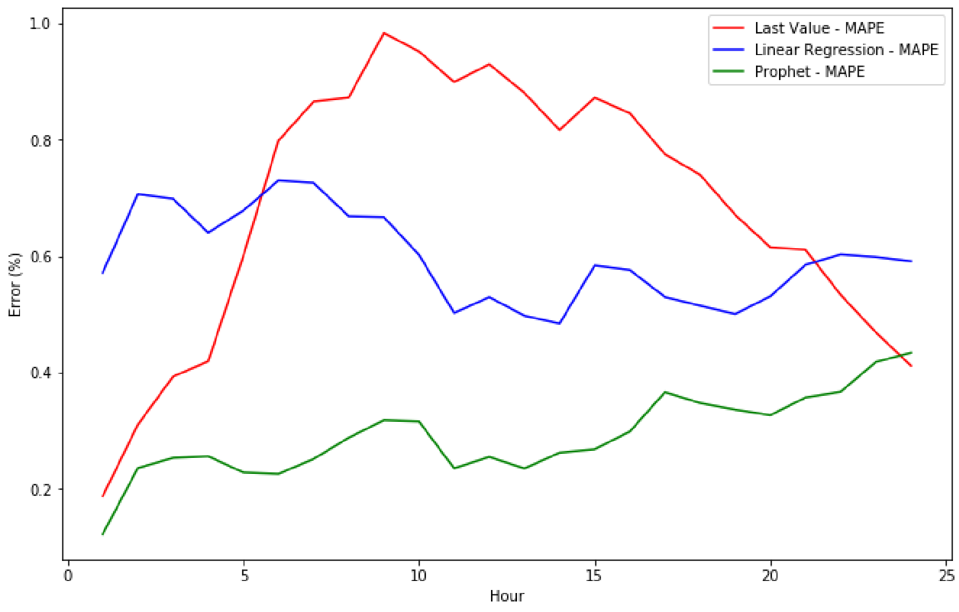

Figure 2.10: Comparison of model MAPE on 24 hour forecast using TCP packet count metric

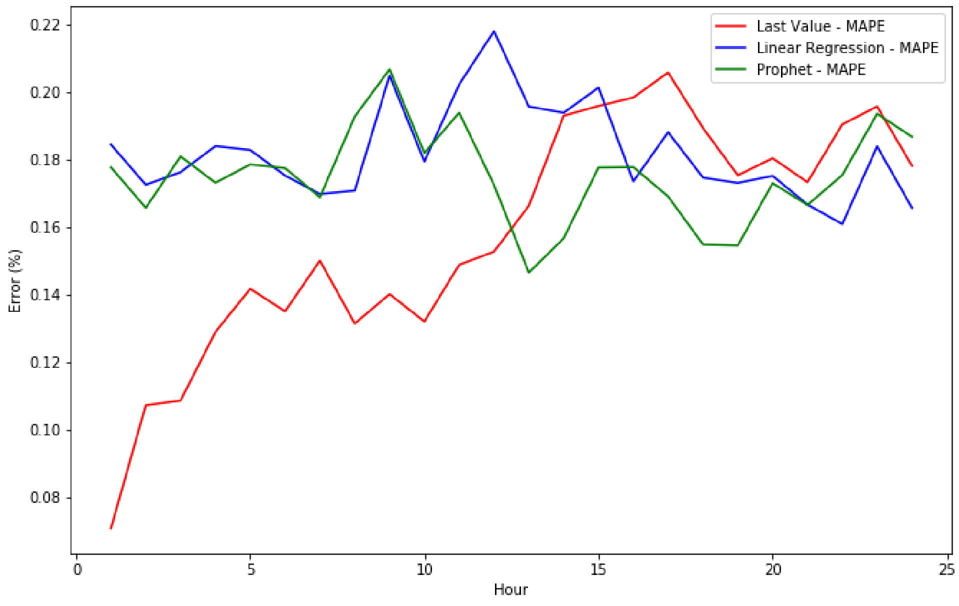

Figure 2.11: Comparison of model MAPE on 24 hour forecast using TCP flow count metric

2.4 System Design

In this section we summarize the decisions made based on the result of forecast model and efficient storage of network communication history evaluation. We also present a high level design overview of the proposed system.

The system is intended to be used for DDoS mitigation on the protected network edge or as an additional support for decision making of network administrators. To satisfy both needs, and to promote a broader adoption in the packet filtering solutions, we've decided to create modular system where each component is decoupled via a well defined application programming interface (API) so that the components can be improved upon or replaced with ease as the requirements evolve. Since we are integrating our solution into larger software project, we also focus on component reusability to maximize utility outside of this system.

The system proposed in this thesis is composed of two main subsystems each spanning multiple components. The first one is the Network Traffic Level Forecast Subsystem. Its purpose is to provide information on expected levels of traffic flowing to a protected computer networks to automatically detect anomalies and either used in semi (e.g. alerting) or fully automated DDoS mitigation system.

The second subsystem is a scalable implementation of History-based IP Filtering, which was shortly introduced in chapter 1.2 of this thesis, and is responsible for gathering and aggregating information about entities communicating with the protected computer networks. As outlined in this chapter, it is build around a probabilistic data structure as the data storage. This subsystem can also either be used in a packet filtering scheme of a DDoS scrubbing center at the edge of a protected network or as a additional source of information to help better understand the nature and source of DDoS attack for network administrator. While this subsystem can be used in a different deployment (e.g. as a host level packet filter), we designed this system with the deployment at the network edge of a protected computer network as its main purpose.

Together these two parts form a system providing information about the monitored networks that we've decided to call *network profile*.

2.4.1 Network Traffic Level Forecast Subsystem

The first part of the system generates the expected levels of traffic on the protected network and makes it available to clients over HTTP protocol. Its architecture is shown on figure 2.12.

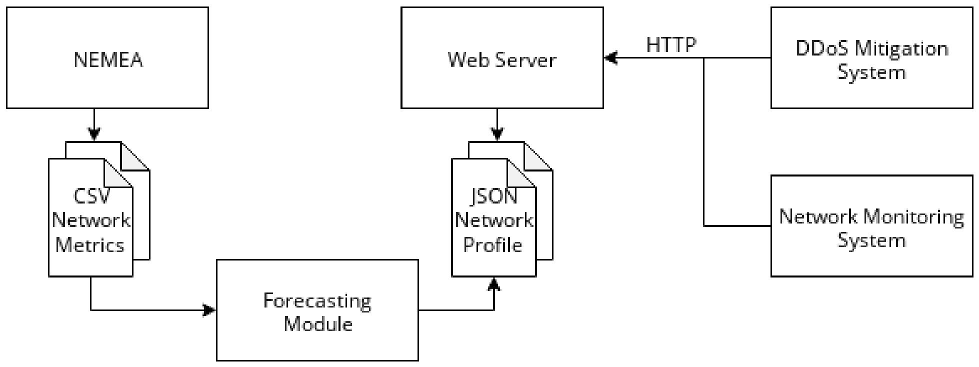

Figure 2.12: Network Traffic Level Forecast subsystem architecture

We've decided to reuse setup and NEMEA configurations described in section 2.3 for network metric data collection and input and is shown simplified on the diagram.

Data gathered by the NEMEA is periodically ingested by the Network Traffic Level Forecast module which then produces a one profile file per protected network with the expected levels of traffic. The profile files are stored on a file system in a hierarchy maintained by the forecast module split by a network on top level and by a time stamp on a sub-level. This allows inspection of the forecast history and eliminates the need for complicated caching scheme on application level since the forecast computation is a highly resource demanding task.

The static files made by the forecasting module are then served to clients by a common web server. The only requirement for the web server configuration is that the latest forecast file is available on a well known URL for simplicity of client implementation. Specifics of the forecast module and its output data formats are described in the next chapter.

For the forecasting model used in the Network Traffic Level Forecast module, we've decided to use the Prophet model described in 2.3 with the same parameters since its performance and robustness best fit our model of 24h expected levels of traffic in the network profile. We've decided to use the Prophet model only even through the Last Value model performed better in some specific cases since we believe that the uniformity and simplicity of the module far outweighs the small difference in the forecast precision.

As can be deduced from the design of this subsystem, this detection mechanism is not well suited for application level attacks since the volume required to successfully carry out a DDoS attack is usually significantly smaller than with the network/transport level attacks described in the previous chapter.

2.4.2 Network Communication History Gathering Subsystem

This subsystem is responsible for gathering and aggregating information about entities communicating with the protected computer networks. The architecture is shown on figure 2.13 with the NEMEA Bloom Filter Pipeline detailed on figure 2.14.

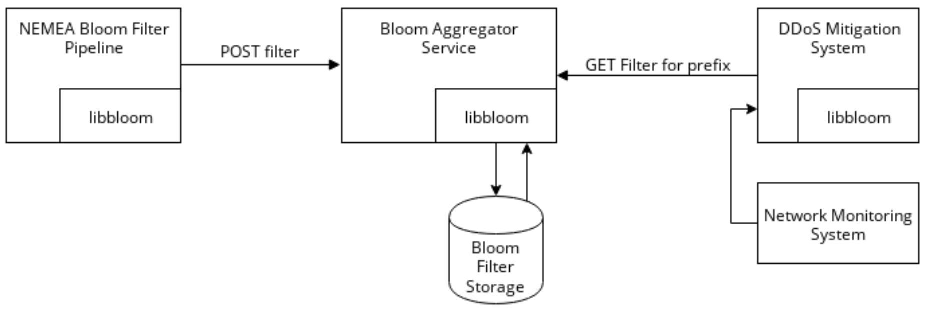

Figure 2.13: Network Communication History Gathering Subsystem

The subsystem scalability, efficient memory usage per entry and query performance is achieved by using probabilistic data structures described in section 2.2. Of the three possibilities, we've decided to use Bloom filters.

The false positives in combination with white list based filtering do not pose a serious issue for our application when the possibility of false positive is kept sufficiently small. To set it in the context of History-based IP filtering, we argue that it is much better to erroneously classify few attacking IP addresses from a massive DDoS attack as legitimate traffic since the infrastructure should be able to handle slightly increased load than block a legitimate client and possibly loose a customer.

The Bloom filters were chosen partially because of their simplicity and ease of implementation, but mainly because the speed with which the unification of two filters can be done and the mechanics of exceeding the designed capacity as discussed in section 2.2. Another benefit of Bloom filters is their wide spread usage and understanding of their limitations.

In this subsystem, the data are first ingested into the NEMEA framework. We then apply simple rules that filter out messages that, for the purpose of this system, we do not consider a valid communication with client. The examples are ICMP Echo Request messages or any unsolicited traffic that does not receive a response form the protected network.

To support a multiple network prefixes per single instance of the Bloom History NEMEA module, we've decided to decouple this functionality to a separate Prefix Tags module. The Prefix Tags module perform network prefix matching and attaches tags to a matched UniRec messages according to its configuration. Messages not belonging in any of the network prefixes listed in configuration are dropped. This lowers load and complexity of the Bloom History module.

The last part of the NEMEA Bloom Filter Pipeline is the Bloom History module. It periodically creates a clean Bloom filter per configured network. As the messages are delivered it inserts the destination IP addresses included in the message into the current Bloom filter using the network tag added by the Prefix Tags module. After a configurable period, the current filter is serialized and send, possibly over network, to the Bloom History Aggregator Service. A new clean Bloom filter is then created to take its place. The API and the serialized format of Bloom filter are described in detail in the next chapter.

While we've restricted ourselves to insert only destination IP address into the Bloom filter in this thesis, it is certainly possible to insert any data available. For example a tuple of source and destination IP addresses could be used, however this also rises the number of distinct elements.

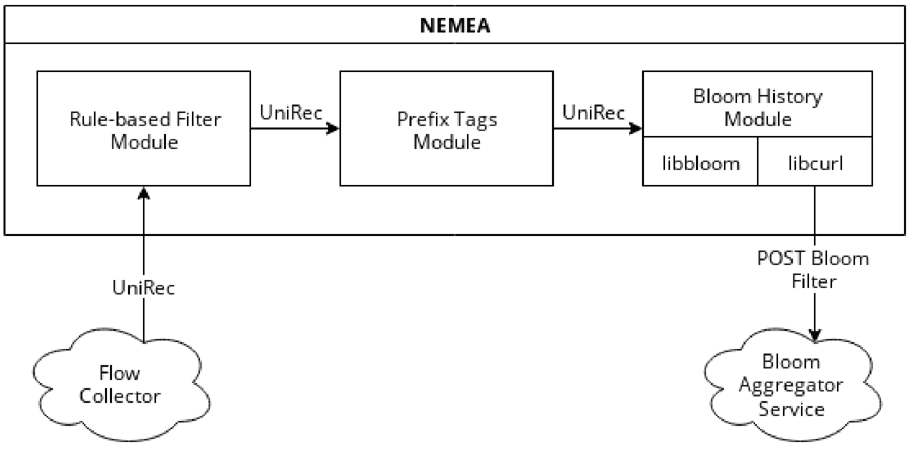

Figure 2.14: NEMEA Bloom filter pipeline architecture

Unfortunately there is a catch to storing the information using a Bloom filter. We most likely don't want to keep the information about IP addresses forever. Certainly, there is a time limit after which, if the IP address did not communicate with the protected prefix, we would like to remove the "white-listing" of such IP address. The catch is in the Bloom filters; once an element is inserted, it can not be effectively removed from a Bloom filter. On the other hand, Bloom filters can be merged together very easily and efficiently without any information loss. This is the reason we've designed the Bloom History Aggregator Service.

The Bloom History Aggregator service shown on figure 2.13 accepts the periodically uploaded Bloom filters send by the Bloom History NEMEA module and stores them in a file system hierarchy similar to the one used by network traffic level forecast module. The main difference is that the separate Bloom filters are serialized with the time range information describing the span of Bloom filter contents.

On request from a client, the Bloom History Aggregator service reads and computes union of a Bloom filters for a given network in requested time range. This results in a efficiently encoded probabilistic set of entities, which communicated with our network in a given time range and which we can consider to be legitimate users (i.e. a white list). The resulting Bloom filter can be then used for decision making support of network administrators in a form of monitoring system or, given the speed of element membership queries on Bloom filters, as a part of an actual DDoS mitigation solution.

There is only one requirement that in case of a DDoS attack on a protected network prefix, the mitigation or alerting system has to notify the Bloom Aggregator Service so that it does not store information about the communicating entities. Failure to do so would result in inclusion of the attacker IP addresses in the white list.

Another aspect of this subsystem is that the Bloom filters cannot be resized once created without having the original data. Because of this a proper planning and analysis of expected number of unique IP addresses in a time window which will be used for construction of the final Bloom filters by Bloom Aggregator Service is needed in order to keep the false positive rate within the configured bounds. Underestimation could result in a Bloom filter with higher false positive rate than expected.

In contrast to the Network Traffic Level Forecast Subsystem, the information provided by this subsystem can be used to diagnose and mitigate even application level DDoS attacks.

Chapter **3**

Implementation

In this chapter we describe APIs and configuration file formats of services implemented as a part of this thesis. We also show some of the more interesting details of the implementation and the format of serialized Bloom filters used by the system.

3.1 Forecasting Module

The forecasting module, described in section 2.4.1, is a Python [37] script periodically run by an operating system. Each run it produces a network profile file for each network data file present. It does not use any configuration file and only `data_root` file system path command line option is required.

The `data_root` option should point to a directory with a specific structure following `<data_root>/<network_id>/data.csv` pattern where the `network_id` is a positive integer used to identify the given computer network.

The `data.csv` is the provided data file in comma-separated values (CSV) format and should have at least the columns specified in table 3.1. The `PROTOCOL` column should contain the IP protocol number assigned and updated by Internet Assigned Numbers Authority (IANA) in [38]. The `BYTES`, `PACKETS` and `COUNT` should contain a sum of byte, packet and flow count observed in the time range given in `TIME_FIRST` and `TIME_LAST`. Each time range should span one hour and not overlap with other periods having the same `PROTOCOL` number. Data in this specific format can be easily obtained from NEMEA framework. An example of NEMEA configuration snippet to produce this data is available on the enclosed media.

Column	Data Type
`TIME_FIRST`	ISO8601 time
`TIME_LAST`	ISO8601 time
`PROTOCOL`	uint8
`BYTES`	uint64
`PACKETS`	uint32
`COUNT`	uint32

Table 3.1: Forecasting Module input data columns

The input data are then processed by Prophet [36] forecast model described in section 2.3 of this thesis.

The output data for each network is written to file in respective network directory following the `profile-<timestamp>.json` naming pattern, where `timestamp` is time of ingestion of the input data in ISO8601 format. In addition `<network_id>/latest.json` symbolic link is atomically updated to point to the latest profile file for each network.

The resulting profile file is in JavaScript Object Notation (JSON) format. An example of this module output is shown in listing 3.1 with field description inline. Note, that the C-style comments in listing 3.1 are not actually part of the JSON specification and we include them int this example for documentation only.

All command line options and file formats are described in the `README.md` file on the enclosed media.

The Forecasting Module is developed and released under the Apache 2.0 license [39].

```
{
  // Protocol identified by IP protocol number
  // By default ‘TCP‘, ‘UDP‘, ‘ICMP‘
  "TCP": {
    // Metric by default ‘bytes‘, ‘packets‘ and
    // ‘flow_count‘
    "bytes": {
      // Start of the one hour period in ISO8601
      // timestamp
      "2018-09-03T08:39:28.705000": {
        // Forecasted value of metric
        "yhat": 143619009181.51968,
        // Lower bound of forecasted value of metric
        "yhat_lower": 106421980435.8867,
        // Upper bound of forecasted value of metric
        "yhat_upper": 180611097587.87106
      },
      ...
    },
    ...
  },
  ...
}
```

Listing 3.1: Forecasting Module output example

3.2 Prefix Tags NEMEA Module

Prefix Tags is a NEMEA module that adds a network id tag to a UniRec messages based on a source or destination IP address according to its configuration. The network id tag can be used later in the NEMEA processing pipeline. We've split this functionality from Bloom History NEMEA module so that it does not need to be re-implemented and can be reused by other modules.

More specifically this module adds `PREFIX_TAG` field to the output UniRec messages based on `SRC_IP` and `DST_IP` fields. By default, the module matches on both source and destination ip address. This behavior can be switched to use only one of source or destination IP address using command line options. Additionally, the output UniRec template updates dynamically according to the input template without need to restart this module. The network matching itself is just a linear match on configured networks as we found it to be sufficiently fast.

```
[
  {"id": 1, "ip_prefix": "170.30.0.0/16"},
  {"id": 2, "ip_prefix": "170.31.0.0/16"},
  {"id": 3, "ip_prefix": "170.32.0.0/16"},
]
```

Listing 3.2: Prefix Tags NEMEA module configuration example

The configuration file is in JSON format. As can be seen in the example configuration on listing 3.2, it is an array of objects each describing single network prefix. The `id` key is used as `PREFIX_TAG` value in the output messages and `ip_prefix` as the network prefix. The number behind the forward slash in `ip_prefix` denotes a network mask length. The keys not used by this module inside each network configuration are ignored and will not rise any error. This is an attempt to make the configuration extensible by other modules and allows us to use single configuration file with the Bloom History NEMEA module.

All command line options and file formats are described in the `README.md` file on the enclosed media.

3.3 Libbloom

As the core part for manipulation of the Bllom filters, we've decided to use the *libbloom* library [22] since it already implements speed improvements presented by Kirsch et al. in [24]. However, a serialization and union of bloom filters, which we require for our purposes, is not supported by the upstream project. Both features were directly added to the library since it touches internals not described in its public API.

As already outlined in section 2.4.2, the union of Bloom filters is needed so that we can create a single Bloom filter spanning longer time frame. The `bloom` structure in the *libbloom* library represents the bit array simply as array of bytes and performs simple bit masking to access the individual bits. To merge the two filters, a simple bitwise OR is performed on the two byte arrays. When the compiler has enabled optimization, this results in almost fully vectorized operation of the merging function.

Also, we've decided to merge the filters in-place and modify one of the `bloom` structures instead of creating a new structure (and thus allocating new memory) for each merge operation. This, of course, improves the performance when merging many filters.

The serialization functionality is needed so that the we can persist the Bloom filters to a file system or send them over a network. We've decided to use the same binary format shown on figure 3.1 for both purposes since it greatly simplifies other parts of the Network Communication History Gathering subsystem described in section 2.4.2. The `size` and `entries` are both unsigned integers in big-endian byte order. The `size` is a total size of the serialized Bloom filter including this field and the `entries` fields denotes for how many entries was this Bloom filter constructed for. The `error` is a IEE754 floating point precision used to construct this Bloom filter. The last field is the `bloom` structure byte array itself.

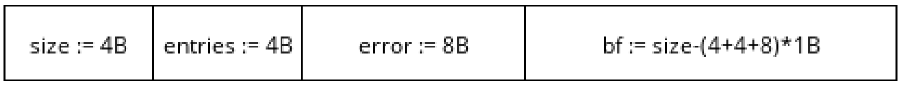

Figure 3.1: Bloom filter binary serialization format

In addition to the two operations described above, we've also added an operation to read and write the serialized `bloom` structure to a file system. This is in order to increase the speed and efficiency of the merging operation.

The sources of the extended *libbloom* library along with our changes are available on the media enclosed to this thesis. Our changes to the library are released under the 2-Clause BSD license [23] which is used by the upstream project.

3.4 Bloom History NEMEA Module

The Bloom History NEMEA module is the last module in the NEMEA pipeline described in section 2.4.2 and is responsible for creation of the Bloom filters containing the information about communicating entities on the protected computer networks and their upload to the Bloom Aggregator service or other storage.

3.4.1 Configuration

This module accepts an interval in seconds as a command line parameter which sets the period for creation and upload of the Bloom filters.

Apart from the interval command line option a configuration file in JSON format can be provided. The format is compatible with configuration

```
[
  {
    "id": 1,
    "bloom_fp_error_rate": 0.01,
    "bloom_entries": 20000000,
    "api_url": "https://localhost:8081/1"
  },
...
]
```

Listing 3.3: Bloom History NEMEA module configuration example

of Prefix Tags module described in section 3.2, but contains several more keys for each specified network. The intention of unified configuration files between the Bloom History and Prefix Tags NEMEA modules is to lower the operational complexity and thus decrease the chance of human error.

An example of configuration file for this module can be seen on listing 3.3. The `id` key is the network identifier matching `PREFIX_TAG` field in the incoming UniRec messages and is shared between the Prefix Tags and this module. The `bloom_fp_error_rate` and `bloom_entries` are both parameters of the Bloom filter corresponding to a specified network network. The former option is a decimal number between 0 and 1 and the later a positive integer. Since Bloom filter can not be resized once created, `bloom_entries` needs to be set to the expected number of distinct IP addresses communicating with the specified network prefix for the desired period of time. This period could be the either expected aggregation interval on the Aggregator Service or simply the interval command line option when no further aggregation is made. The false-positive rate will get worse than specified by `bloom_fp_error_rate` if more distinct entries than configured is inserted. The `api_url` specifies the HTTP endpoint to which the Bloom filters are sent at the end of each interval.

Similarly to the Prefix Tags module, all of the listed configuration keys are required since none has a default value and keys in the configuration file not used by this module are not considered to be an error.

3.4.2 Execution Model

On the input of this module, messages containing the `PREFIX_TAG` network identification field added by the Prefix Tags NEMEA module described in this chapter and a `DST_IP` field containing destination IP address is expected. The address in `DST_IP` field of each message is then inserted into a Bloom filter corresponding to a network id tag added by the Prefix Tags module earlier in the pipeline.

The module consists of two threads, each with a different task. The first thread is responsible for ingestion of the incoming UniRec messages and their insertion to the corresponding Bloom filters. Here we expect a sufficiently low network id tag so that it can be used as an offset to an array of Bloom filters.

Since the element insertion to a Bloom filter cannot be done in a single atomic operation, the access of the two threads to the Bloom filters is synchronized by a POSIX mutex and has to be acquired for each element insertion. As shown in the next chapter, the module does not have any performance issues even with this simple access method.

The second thread is responsible for a periodic creation of new Bloom filters and upload of the current ones to the Bloom Aggregator service. Each period a set of new Bloom filters is allocated according to their specific parameters and swapped with the current set while holding the lock. The old filters are then one by one uploaded to the `api_url` endpoint specified in their respective configurations.

A simple sleeping for the upload thread is not suitable, since when the module receives a signal to terminate we still have a set of Bloom filters possibly containing data. On the other hand waiting for the sleeping thread to wake up is also not tolerable since the termination could take in worst case scenario up to the upload interval. Given these constrains we've decided to use POSIX conditional variable with time-limited wait using the `pthread_cond_timedwait` function.

On exit, the main thread signalizes it's intent to the upload thread by using `pthread_cond_signal` function and waits for the upload thread to exit. The upload thread then uploads last set of Bloom filters and exits. The thread execution model is depicted on figure 3.2.

3.4.3 Bloom Filter Upload

The upload of the Bloom filter set after each period and at the module termination is done using HTTP protocol. Each Bloom filter is sent to their respective HTTP endpoints composed using specified `api_url` configuration key and the span of information contained in the Bloom filter following the `<api_url>/<t_start>/<t_end>` endpoint pattern. The `t_start` and `t_end` are Unix Epoch time stamps (a positive integer) of the start and end of the filter time range respectively.

The module performs a HTTP POST request to the composed endpoint with a serialized Bloom filter as a body of the request. For this, the

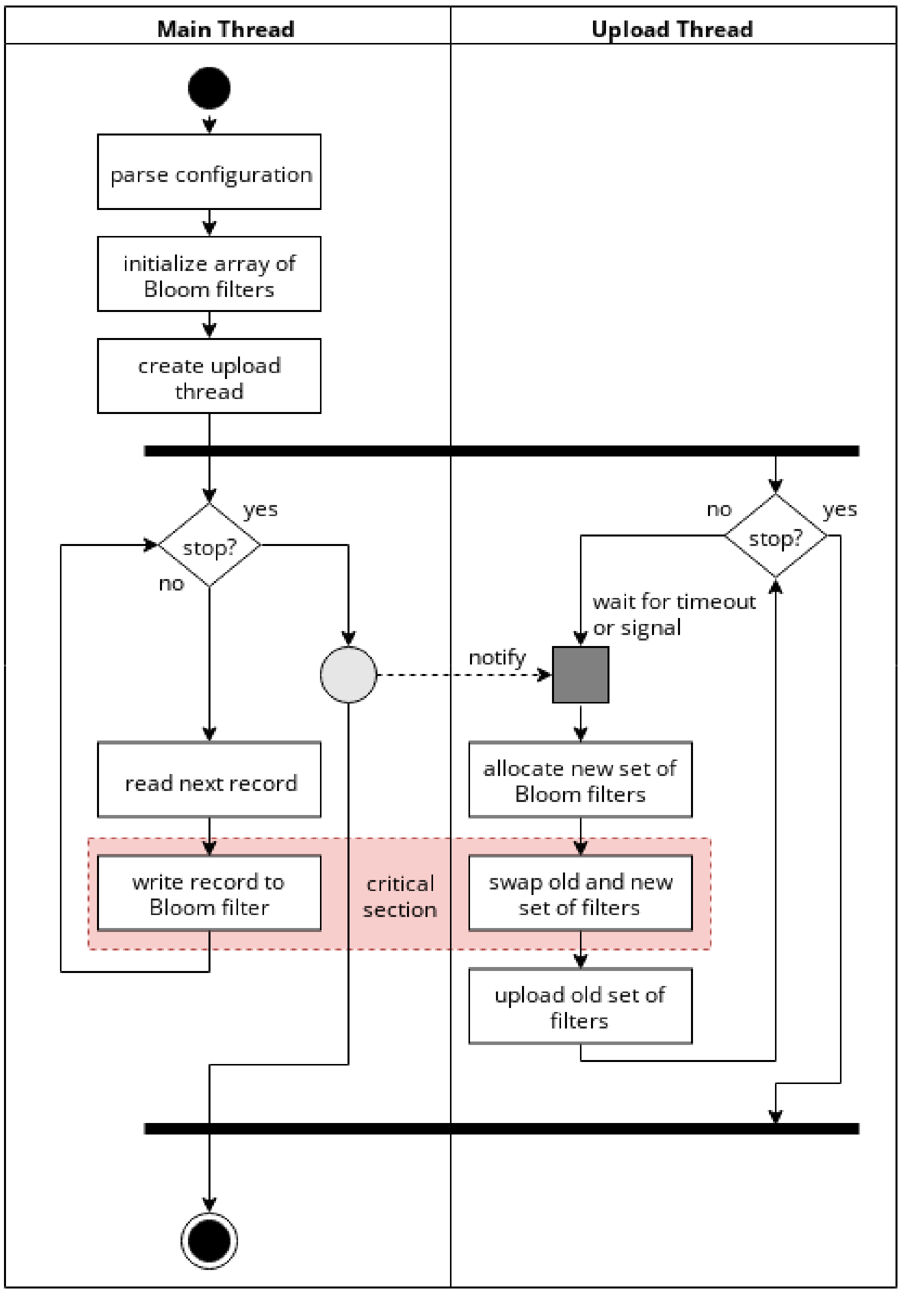
Main Thread
Upload Thread
parse configuration
initialize array of Bloom filters
create upload thread
stop?
yes
no
read next record
write record to Bloom filter
critical section
notify
no
stop?
yes
wait for timeout or signal
allocate new set of Bloom filters
swap old and new set of filters
upload old set of filters

Figure 3.2: Bloom History NEMEA module thread execution model

libbloom binary serialization format described in 3.3 is used unchanged without any additional encoding. To ensure that the binary data in the body of the request are accepted, we set `Content-Type` header of the request to `application/octet-stream`.

The HTTP communication in the Bloom History module is implemented using *libcurl* library [40] which is developed under license derived from and compatible with the MIT license [41]. Specifically, we use the easy interface which is synchronous, efficient and relatively easy to use.

For each of the requests a new connection is created since the endpoints can be different. This is done by using the `curl_easy_init` function. The parameters for the connection are all set using the `curl_easy_setopt` including the URL, HTTP headers, HTTP method and a buffer with the POST body. Notably, *libcurl* computes the length of the request body automatically using the `strlen` function. In our case this is not correct since we are sending binary data which can contain null byte and can result in *libcurl* sending only part of the specified buffer. To avoid this kind of error, `CURLOPT_POSTFIELDSIZE` option is used to set the correct buffer length.

The constructed request is then sent using the `curl_easy_perform` function. Since we are using the synchronous interface, there is no need for callback functions and the response code can be easily checked using the `curl_easy_getinfo` with `CURLINFO_RESPONSE_CODE` parameter. Apart from HTTP redirects which are handled automatically by *libcurl*, all response codes different than `200 OK` are considered to be an error.

The request can be optionally encrypted when `https://` scheme is used in the `api_url` configuration key.

3.5 Bloom History Aggregator Service

The Bloom History Aggregator Service described in section 2.4.2 is a Python application build using the Flask micro web framework [42] with bindings to *libbloom* for Bloom filter data processing and serialization. Overlooking some light logic around Bloom filter storage, the main purpose of this service is to provide a HTTP API around *libbloom*.

A simple file system structure is used for Bloom filter storage. A Flask Instance Folder is used to store the data. Its location on the file system is dependent on the style of the service installation. Under the data root path one folder per network id is created each containing files named using `<t_start>-<t_end>.bloom` pattern where `t_start` and `t_end` are Unix Epoch time stamps denoting the start and end of the contained data span respectively. The network identificator is a positive integer. The intention is to use the same network identificators across the whole system described in this thesis.

The intended setup for this service is via uWSGI [43] application server. An example uWSGI server configuration for this Flask web application is available on the enclosed media along with SystemD service file, documentation, build instructions and full list of dependencies. A test suite is also included to verify full functionality on the target system.

Since this service leverages the extended libbloom for Bloom filter manipulation, it needs to be installed on the system to be fully functional. Also, because of the bindings to libbloom, building this service requires a C compiler.

The Bloom History Aggregator Service is developed and released under the Apache 2 license [39].

3.5.1 API

Bloom History Aggregator Service HTTP API is designed to be stateless. This means that requests are independent of each other and no session context needs to be held on either server of client.

The API endpoint segments written in angle brackets are used as a named pattern. Patterns with the same name represent the same type across all endpoints.

The `id` must be unique positive integer identifying a network prefix. The `t_start` and `t_end` must be Unix time stamps (i.e. a positive integer) forming a start and end of time range respectively.

The API consists of two endpoints implementing multiple HTTP methods:

`POST /<id>/<t_start>/<t_end>`

Upload new Bloom filter. The request body must contain only serialized Bloom filter described in 3.3. and `Content-type` must be `application/octet-stream`.

The filter is written to a file in the `<id>` directory under the Flask instance path using `<t_start>-<t_end>.bloom` file name pattern.

`GET /<id>/<t_start>/<t_end>`

Get union of Bloom filters spanning the given time range. The response body contains single Bloom filter in the binary serialized form described in 3.3 merged from all filters within the range given by `t_start` and `t_end`.

Since the Bloom filters are stored as a files with the given time stamp range the resulting filter union cannot be precise. The Bloom filter time range must be fully contained in the request time range for it to be included in the aggregated Bloom Filter response. If the requested range does not include any Bloom filter file, the service returns HTTP code `404 Not Found`.

Response `Content-type` is set to `application/octet-stream`.

`DELETE /<id>/<t_start>/<t_end>`

Delete Bloom filters in time range specified by `t_start` and `t_end`. This endpoint is intended to be used for deletion of history in case a DDoS or any other malicious traffic got into the Bloom filters.

The service deletes Bloom filters filter under the `<id>` directory within the specified range.

Similarly to the `GET` method on this endpoint, the time range is exclusive and if the range is empty, HTTP code `404 Not Found` is returned.

`GET /health`

A simple health check endpoint. Responds with HTTP code `200 OK` if the API is in an operational state. This can be used for automated service readiness probe.

3.5.2 Libbloom bindings

As mentioned earlier, this service is internally using the extended *libbloom* for Bloom filter manipulation. This is achieved through C Foreign Function Interface for Python (CFFI) bindings [44] which allows to interact with almost any C code from Python.

The CFFI offers several ways of interfacing with the C libraries. We've decided to use *out-of-line API* mode which, instead of binary level, accesses a C library at the level of C. This offers the most flexibility and speed compared to the binary level mode. In this mode, C source is generated containing the functions and structures needed, which is then compiled by a C-compiler into an intermediate shared object. This object is then used by the Python process instead of directly binding to the binary interface of the target library. However, there is one downside to this mode; if we want to leverage the system installed *libbloom* instead of letting it be compiled by the CFFI, we need to provide a C functions wrapping the target library. In our case, this is a small subset of the *libbloom* functions and does not pose a serious issue. This tiny wrapper needs to be handed over to the package global `cffi.FFI` object `set_source` method along with a list of dynamic libraries for linking.

The CFFI also needs definitions of the target C functions and structures to be able to generate the Python bindings. This C definitions are parsed by a Python library. Because of this, the definitions can use only structures defined in this source, standard C data types and cannot contain `#include` directives. The definitions are then loaded to the `cffi.FFI` object by using the `cdef()` method.

The sources handed over to CFFI are then compiled by calling the `cffi.FFI` object `compile()` method. This produces the shared object bridging the *libbloom* and the Bloom History Aggregator Service. However, to be able to produce a Python binary package using its standard tool chain, the `setup.py` file needs to include `cffi_modules` option pointing to the `FFI` object with all sources and C definitions loaded.

The resulting shared object can then be imported from Python as a native extension package.

The writing of Bloom filters received from a client in a body of a HTTP POST request is handled without usage of the bindings to *libbloom*. However, to merge multiple Bloom filters we use the `bloom_file_read` function which returns only a pointer to the `bloom` structure. We then simply pass the pointers to the *libbloom* `bloom_merge` functions. This is very efficient since transfer of many large memory chunks between native Python types and types used in the C extension is avoided.

Only place where a transfer of large memory chunk from C extension to a Python type occurs is when the final Bloom filter serialized to a buffer is returned in a HTTP response to the caller. This is done by unpacking the C byte array to a Python `bytes` object. Since the C array already contains a wire format of the final Bloom filter, this is all that needs to be done to prepare the response data.

By using the CFFI bindings, we've been able to avoid duplication of Bloom filter manipulation logic and achieve almost native speeds of processing while keeping the flexibility and ease of use of the Flask web framework.

Chapter 4

Evaluation

In this chapter, we describe in detail method and show the results of performance testing of each of the separate modules described in the previous chapter.

The performance measurement of each of the modules was done on a dedicated machine with no other load than the test itself. The concrete parameters of the test environment are listed in table 4.1.

CPU:	Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz
RAM:	8064MiB
DISK:	KINGSTON SV300S3
OS:	Fedora release 28 (Twenty Eight)
Kernel:	4.17.3
CC:	gcc version 8.1.1 20180712 (Red Hat 8.1.1-5) (GCC)

Table 4.1: Test environment specification

4.1 Prefix Tags NEMEA Module

For testing of the Prefix Tags NEMEA Module, we've measured throughput of records per second based on the number of networks configured. All of the configured networks used a `/24` prefix and each configuration had exactly 30 million randomly generated records with destination address uniformly distributed over the networks. None of the source IP addresses was from the configured network ranges.

The generated records were read by the NEMEA module from a file stored on the test system disk. To make sure that the module performance is no capped by disk read bandwidth, we've used dataset that fully fits to the available system memory and preloaded the data to file system cache of the operating system. To avoid introducing additional overhead, the output of the module was dropped during execution.

The module was executed five times for each of the configurations. The measured metric is mean user space CPU time of the five test executions used by the process as reported by the `time` utility. We've also executed the module in a mode to match source only, destination only and both source and destination IP address.

The throughput of records per second based on number of configured networks with all three modes of module operations is shown on figure 4.1. As expected, there is clearly visible linear decrease in throughput with the rising number of configured network prefixes. Despite our expectations, matching on source ip address only is significantly faster than the other two modes. In our configuration and testing dataset, this means linear matching of all rules and then drop of the record since the source IP addresses were generated so that it does not match any of the configured networks. Also there is very little difference between destination only and both source and destination IP address matching. This clearly shows that the actual prefix matching is much less CPU demanding than module data input and output processing.

As shown in the results, we've exceeded the requirements set in section 2.1 by order of magnitude even with simple linear matching method. Since we currently have no use-case for large number of networks, this is sufficient.

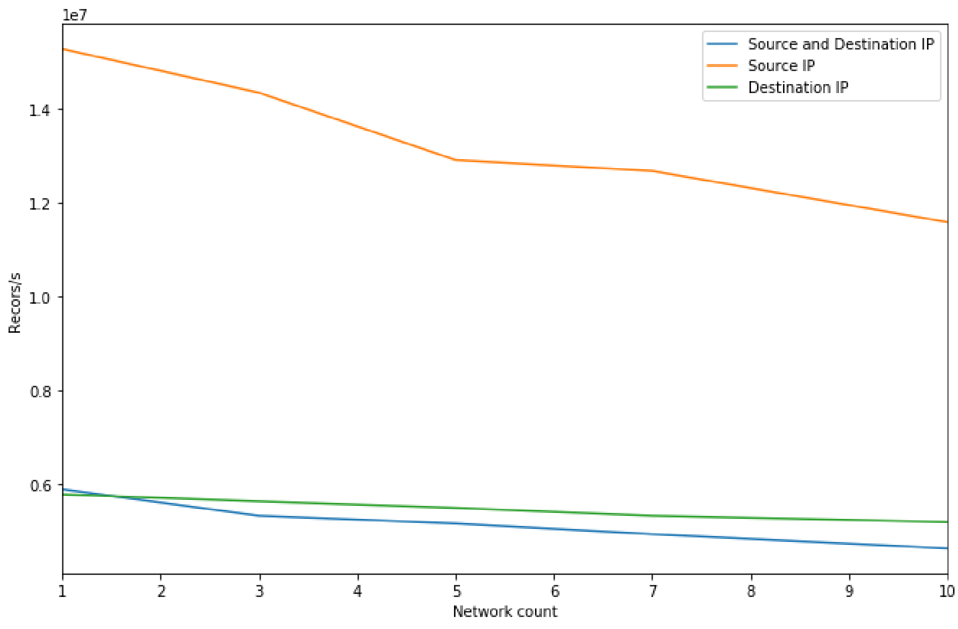

Figure 4.1: Number of records processed by Prefix Tags NEMEA Module per second based on number of configured networks

4.2 Bloom History Aggregator Service

For deployment of Bloom History Aggregator Service we are interested in latency of end-to-end request and the speed of Bloom filter merging via the CFFI interface.

Since the merging speed is dependent on the size of the Bloom filter byte array only, we keep the false positive rate constant in this experiment and change the size of a Bloom filters between given testing sets. Each of the Bloom filters used for this experiment was filled to the 80% of its capacity with random elements. The actual size of a serialized Bloom filter designed to hold given number of records at set false positive rate is shown in table 4.2.

The metric measured in this experiment is mean real time reported by the `time` utility of 10 requests since we are interested in the end-to-end latency of this service. We infer the average time taken to merge two Bloom filters together by the Flask application by subtracting the transfer overhead from the total request time.

The transfer overhead is a time taken to download a single Bloom filter. In case of our testing setup, this is composed of `curl` sending request to the service on local machine, loading and serialization of the filter to Python

Capacity	FPR	Size (MB)
100000	0.01	0.120
1000000	0.01	1.2
10000000	0.01	12
100000000	0.01	115

Table 4.2: Size of a serialized Bloom filter based on designed capacity and false positive rate

object, composition and transfer of the Flask response over the uWSGI application server Unix socket to Nginx web server and transfer from Nginx back to the `curl` client. The data path is visualized on figure 4.2. This testing setup is very similar to an expected production deployment. Only difference is that the client would send the request over a computer network instead of machine local interface.

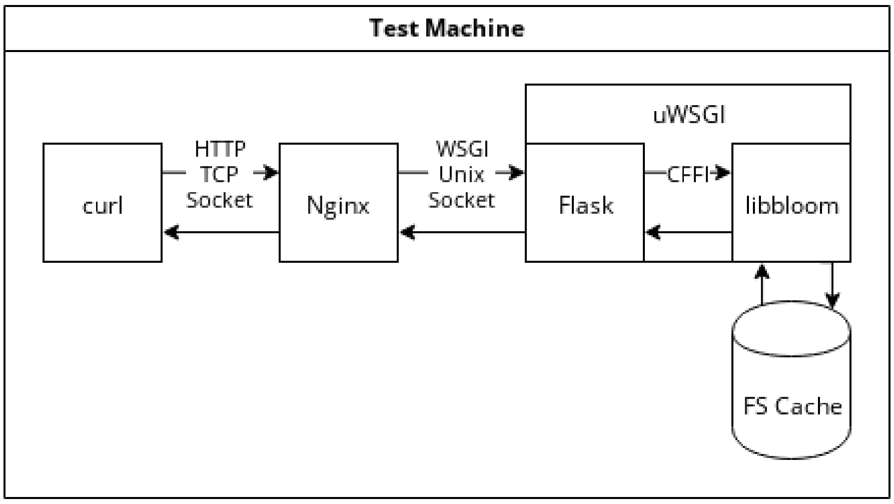

Figure 4.2: Request data path in the Bloom History Aggregator Service testing setup

During the testing, we've noticed that even the fastest persistent storage we had available was limiting the service performance. To test the actual service limits, we've decided to limit the number of Bloom filters so that they can be fully preoloaded to the file system cache of the operating system for the given test.

The time taken to merge the given number of filters along with the measured transfer overhead described above and the average time taken to merge two filters together based on the designed Bloom filter size is shown in table 4.3. As can be seen, the time to complete the request and time taken per one filter grows proportionally with the Bloom filter size.

Capacity	Merged Count	Time (s)	Overhead (s)	Time/filter (s)
100000	100	0.033333	0.021	0.000123
1000000	100	0.230667	0.051	0.001797
10000000	100	2.286500	0.298	0.019885
100000000	50	15.156833	2.895	0.245237

Table 4.3: Bloom History Aggregator service performance

Since the main limitation of the Bloom History Aggregator Service is the persistent storage, some further optimizations might be needed in case of larger deployments. For example decreasing the time range granularity of older Bloom filters by periodic server-side filter merging to a larger time ranges is possible.

4.3 Bloom History NEMEA Module

In case of Bloom History NEMEA Module, the throughput of records per second that can be ingested and inserted into a Bloom filter is critical.

We've decided to configure only one network prefix for this test scenario and measure the effects of changing Bloom filter designed capacity and false positive rate on the throughput of records per second.

To simulate the real load as close as possible a dataset created from an actual production records was used in this experiment since a generated dataset would most likely miss any hidden patterns commonly found in real network communication. For example it is reasonable to expect a time-proximity clustering of IP addresses which in turn affects access patterns to the byte array of a Bloom filter in a real network scenario. Since the memory used by a Bloom filter can be considerate, this could potentially affect performance due to cache misses. As per design, we've filtered out any flows not originating from the configured network and limit the remaining dataset to 20 million records.

The time measured is from first received record to last record processed. The data were served from a local file which was preloaded to a file system cache beforehand. Also for the purpose of this measurement, the upload thread was disabled to avoid any accidental inconsistencies across the measurements.

Results of the measurement are shown on figure 4.3. The initial steep performance decrease for small Bloom filter sizes is caused by the filter fitting into L2 and L3 CPU cache respectively. Beyond the initial rapid decrease, there is only a small performance penalty for increasing the Bloom filter size. The steep decrease of performance for small false positive rate is also not surprising since the optimal number of hash functions used by the Bloom filters has logarithmic dependency on the false positive rate.

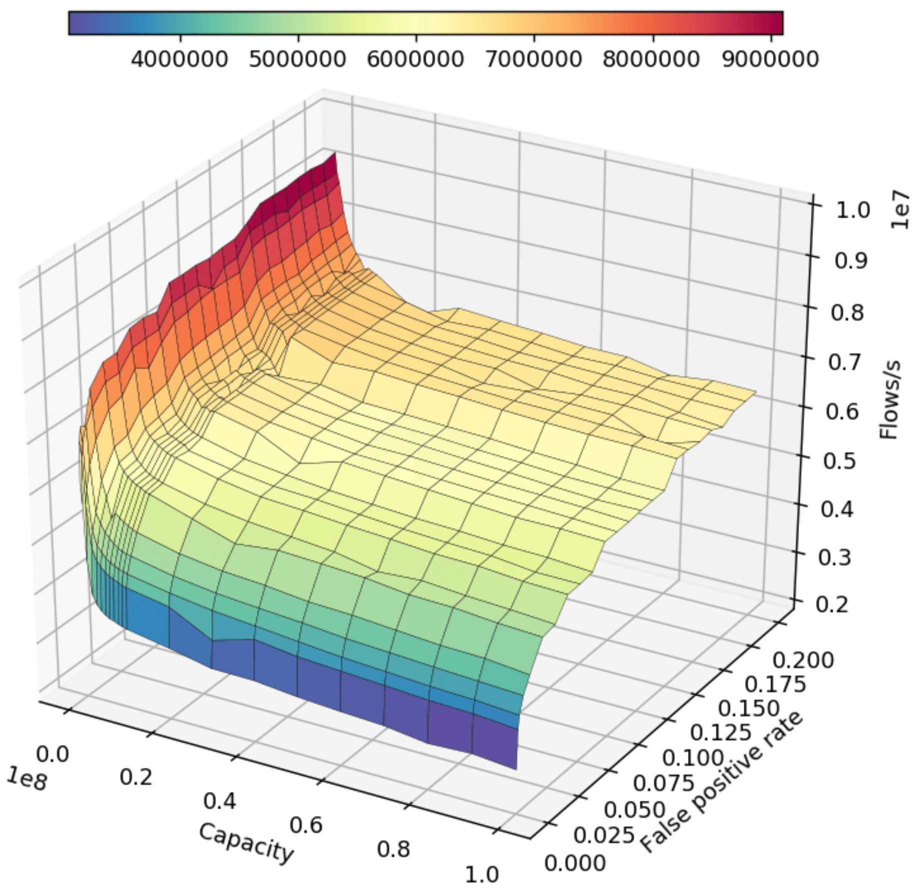

Figure 4.3: Number of records processed by Bloom History NEMEA Module per second based on Bloom filter configured capacity and false positive rate

Similarly to Prefix Tags NEMEA Module, we've exceeded the requirements set in section 2.1 almost by order of magnitude even with model using single worker thread.

Conclusion

The primary objective of this thesis was to design and implement a system that provides information in a fully automated manner describing the normal operation of a computer network which can be used for detection, diagnosis and possible mitigation of DDoS attacks. Based on the requirement analysis a *network profile* was designed.

The *network profile* is composed of two components, both of which are built using data extracted from a network flow information using the NEMEA system developed by *CESNET a.l.e.*.

The first component of the profile gives an information about expected levels of traffic based on its long term observations and is provided by the Network Traffic Level Forecast Subsystem. The results of the Prophet forecasting model evaluation performed on the data collected from CESNET2 network show, that the Network Traffic Level Forecast Subsystem has predictable error growth even in presence of a large anomalies in the training data and is kept under 40% in the 24h forecast period.

The second component of the *network profile* provides efficiently encoded set of entities which historically communicated with the protected computer networks in a given time period. This information is provided by the Network Communication History Gathering Subsystem which is a novel and scalable implementation of History-based IP filtering DDoS mitigation method that provides theoretical basis of this component.

The Network Communication History Gathering Subsystem is build around Bloom filters, a probabilistic space-efficient data structure, as the sole storage of the information. This design decision allows to fit a set of ten million distinct network entities into just 12MB of memory at 0.01 false positive rate. Together with constant element membership query complexity this makes it directly usable in a packet filtering scheme of a DDoS scrubbing center at the edge of a protected network.

The results of a performance measurement show that the implemented system is suitable even for deployments on networks communicating with over a hundred million of distinct network entities which vastly exceeds requirements for its intended deployment.

While the resulting *network profile* is designed with packet filtering scenarios on edge of the network as one of its use-cases, methods of its incorporation in broader more complex DDoS mitigation scheme needs to be explored in future research.

The implemented NEMEA modules were successfully contributed to the upstream project.

Bibliography

1. FAN, Bin; ANDERSEN, Dave G.; KAMINSKY, Michael; MITZENMACHER, Michael D. Cuckoo Filter: Practically Better Than Bloom. *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies - CoNEXT '14*. 2014. ISBN 9781450332798. Available from DOI: `10.1145/2674005.2674994`.
2. MAJKOWSKI, Marek. *The rise of multivector DDoS attacks* [online] [visited on 2019-04-20]. Available from: `https://blog.cloudflare.com/the-rise-of-multivector-amplifications/`.
3. ZARGAR, Saman Taghavi; JOSHI, James; TIPPER, David. A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks. *IEEE Communications Surveys and Tutorials*. 2013, vol. 15, no. 4, pp. 2046–2069. ISSN 1553-877X. Available from DOI: `10.1109/surv.2013.031413.00127`.
4. *February 28th DDoS Incident Report* [online] [visited on 2019-04-20]. Available from: `https://github.blog/2018-03-01-ddos-incident-report/`.
5. MIRKOVIC, Jelena; REIHER, Peter. A taxonomy of DDoS attack and DDoS defense mechanisms. *ACM SIGCOMM Computer Communication Review*. 2004, vol. 34, no. 2, pp. 39. ISSN 0146-4833. Available from DOI: `10.1145/997150.997156`.
6. *Mirai Botnet* [online] [visited on 2019-04-20]. Available from: `https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/`.
7. MAJKOWSKI, Marek. *Memcrashed - Major amplification attacks from UDP port 11211* [online] [visited on 2019-04-20]. Available from: `https:`

`//blog.cloudflare.com/memcrashed-major-amplification-attacks-from-port-11211/`.

8. PENG, Tao; LECKIE, C.; RAMAMOHANARAO, K. Protection from distributed denial of service attacks using history-based IP filtering. *IEEE International Conference on Communications, 2003. ICC '03.* ISBN 0780378024. Available from DOI: `10.1109/icc.2003.1204223`.
9. JUNG, Jaeyeon; KRISHNAMURTHY, Balachander; RABINOVICH, Michael. Flash crowds and denial of service attacks. *Proceedings of the eleventh international conference on World Wide Web - WWW '02.* 2002. ISBN 1581134495. Available from DOI: `10.1145/511446.511485`.
10. JÁNSKÝ, Tomáš. Informed DDoS mitigation based on reputation. 2018.
11. HOFSTEDE, Rick; CELEDA, Pavel; TRAMMELL, Brian; DRAGO, Idilio; SADRE, Ramin; SPEROTTO, Anna; PRAS, Aiko. Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX. *IEEE Communications Surveys and Tutorials.* 2014, vol. 16, no. 4, pp. 2037–2064. ISSN 1553-877X. Available from DOI: `10.1109/comst.2014.2321898`.
12. AITKEN, Paul; CLAISE, Benoît; TRAMMELL, Brian. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information* [RFC 7011]. RFC Editor. Request for Comments, no. 7011. Available from DOI: `10.17487/RFC7011`.
13. *NEMEA* [online] [visited on 2019-03-20]. Available from: `https://github.com/CESNET/Nemea`.
14. *GNU General Public License version 2* [online] [visited on 2019-03-20]. Available from: `https://opensource.org/licenses/gpl-2.0`.
15. *The 3-Clause BSD License* [online] [visited on 2019-03-20]. Available from: `https://opensource.org/licenses/BSD-3-Clause`.
16. BLOOM, Burton H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM.* 1970, vol. 13, no. 7, pp. 422–426. ISSN 0001-0782. Available from DOI: `10.1145/362686.362692`.
17. TREAT, Tyler. *Stream Processing and Probabilistic Methods: Data at Scale* [online] [visited on 2019-03-20]. Available from: `https://bravenewgeek.com/stream-processing-and-probabilistic-methods/`.

18. BONOMI, Flavio; MITZENMACHER, Michael; PANIGRAHY, Rina; SINGH, Sushil; VARGHESE, George. An Improved Construction for Counting Bloom Filters. *Algorithms – ESA 2006*. 2006, pp. 684–695. ISBN 9783540388760. ISSN 1611-3349. Available from DOI: `10.1007/11841036_61`.

19. BRODER, Andrei; MITZENMACHER, Michael. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*. 2004, vol. 1, no. 4, pp. 485–509. ISSN 1944-9488. Available from DOI: `10.1080/15427951.2004.10129096`.

20. TARKOMA, Sasu; ROTHENBERG, Christian Esteve; LAGERSPETZ, Eemil. 1. *IEEE Communications Surveys and Tutorials*. 2012, vol. 14, no. 1, pp. 131–155. ISSN 1553-877X. Available from DOI: `10.1109/surv.2011.031611.00024`.

21. SWAMIDASS, S. Joshua; BALDI, Pierre. Mathematical Correction for Fingerprint Similarity Measures to Improve Chemical Retrieval. *Journal of Chemical Information and Modeling*. 2007, vol. 47, no. 3, pp. 952–964. ISSN 1549-960X. Available from DOI: `10.1021/ci600526a`.

22. VIRKKI, Jyri J. *libbloom* [online] [visited on 2019-03-20]. Available from: `https://github.com/jvirkki/libbloom`.

23. *The 2-Clause BSD License* [online] [visited on 2019-03-20]. Available from: `https://opensource.org/licenses/BSD-2-Clause`.

24. KIRSCH, Adam; MITZENMACHER, Michael. Less hashing, same performance: Building a better Bloom filter. *Random Structures and Algorithms*. 2008, vol. 33, no. 2, pp. 187–218. ISSN 1098-2418. Available from DOI: `10.1002/rsa.20208`.

25. *NEMEA Framework* [online] [visited on 2019-03-20]. Available from: `https://github.com/CESNET/Nemea-Framework`.

26. BENDER, Michael A. et al. Don't thrash. *Proceedings of the VLDB Endowment*. 2012, vol. 5, no. 11, pp. 1627–1637. ISSN 2150-8097. Available from DOI: `10.14778/2350229.2350275`.

27. *Requirements for IP Flow Information Export (IPFIX)* [online] [visited on 2019-03-20]. Available from: `https://tools.ietf.org/html/rfc3917`.

28. *IPFIXcol* [online] [visited on 2019-03-20]. Available from: `https://github.com/CESNET/ipfixcol/`.

29. HYNDMAN, Rob. *Forecasting : principles and practice ; [a comprehensive indtroduction to the latest forecasting methods using R ; learn to improve your forecast accuracy using dozenss of real data examples.* Lexington, Ky: Otexts, 2018. ISBN 0987507117.

30. *Forecasting: Principles and Practice* [online] [visited on 2019-03-20]. Available from: `https://otexts.com/fpp2/accuracy.html`.

31. *DoD standard Internet Protocol* [RFC 760]. RFC Editor, 1980. No. 760. Available from DOI: `10.17487/RFC0760`.

32. DEERING, Dr. Steve E.; HINDEN, Bob. *Internet Protocol, Version 6 (IPv6) Specification* [RFC 8200]. RFC Editor, 2017. Request for Comments, no. 8200. Available from DOI: `10.17487/RFC8200`.

33. *Jupyter* [online] [visited on 2019-03-20]. Available from: `https://jupyter.org/`.

34. *scikit-learn* [online] [visited on 2019-03-20]. Available from: `https://scikit-learn.org/`.

35. TAYLOR, Sean J; LETHAM, Benjamin. Forecasting at scale. 2017. Available from DOI: `10.7287/peerj.preprints.3190v2`.

36. *Prophet* [online] [visited on 2019-03-20]. Available from: `https://facebook.github.io/prophet/`.

37. *Python* [online] [visited on 2019-03-20]. Available from: `https://www.python.org/`.

38. *Assigned Internet Protocol Numbers* [online] [visited on 2019-03-20]. Available from: `http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml`.

39. *Apache License, Version 2.0* [online] [visited on 2019-03-20]. Available from: `https://www.apache.org/licenses/LICENSE-2.0.html`.

40. *libcurl* [online] [visited on 2019-03-20]. Available from: `https://curl.haxx.se/libcurl/`.

41. *The MIT License* [online] [visited on 2019-03-20]. Available from: `https://opensource.org/licenses/MIT`.

42. *Flask Web Microframework* [online] [visited on 2019-04-20]. Available from: `http://flask.pocoo.org/`.

43. *uWSGI Application Server Container* [online] [visited on 2019-04-20]. Available from: `http://projects.unbit.it/uwsgi`.

44. *C Foreign Function Interface for Python* [online] [visited on 2019-04-20]. Available from: `https://cffi.readthedocs.io/en/latest/`.

Appendix **A**

Acronyms

API application programming interface.

CFFI C Foreign Function Interface for Python.

CSV comma-separated values.

DDoS Distributed Denial of Service.

DNS Domain Name System.

IANA Internet Assigned Numbers Authority.

ICMP Internet Control Message Protocol.

IPFIX IP Flow Information Export.

JSON JavaScript Object Notation.

MAE mean absolute error.

MAPE mean absolute percentage error.

NEMEA Network Measurements Analysis.

OLS Ordinary Least Squares.

RMSE root mean squared error.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

UniRec Unified Record.

Appendix **B**

Contents of enclosed CD

```
│ readme.txt ......................... file with CD contents description
├── src
│   ├── impl ................... the directory with implementation sources
│   └── thesis ........... the directory with LaTeXsource of the thesis text
└── text
    └── DP_Křesťan_Filip_2019.pdf ....... the thesis text in PDF format
```