



NTNU – Trondheim
Norwegian University of
Science and Technology

Senatus - Implementation and Performance Evaluation

Christian Emil Askeland
Anders Emil Salvesen
Arne Fjæren Østvold

Master of Science in Communication Technology

Submission date: June 2012

Supervisor: Yuming Jiang, ITEM

Co-supervisor: Atef Abdelkefi, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Name of Students: Anders Emil Salvesen, Christian Askeland, Arne Østvold

Traffic anomaly detection has received a lot of attention over the years. Various techniques have been proposed in this area and some of them become commercial. They have been mainly designed to flag alarms when suspicious events are happening in the network, while the root-cause analysis was left for manual inspection by network operators. In this thesis, we address this shortcoming while proposing SENATUS, a new framework for a joint anomaly detection and root cause analysis. This master project involves several main tasks:

1. Understanding network attacks and their pattern in the flow-based data collected.
2. Understanding Senatus framework that has been designed by Atef Abdelkefi.
3. Designing an automatic root-cause identification algorithm of Senatus detected anomalies
4. Senatus implementation using C++ programming language.
5. Senatus web-based graphical interface implementation using PHP.
6. Senatus performance evaluation adopting two main approaches:
 - a. Manual ground-truth construction of set of set of attacks in the collected traces
 - b. Comparison with an existing technique for traffic anomaly detection, called Histogram Based Detection.

Assignment given: January 2012

Supervisor: Atef Abdelkefi

Abstract

Traffic anomaly detection in backbone networks has received increased attention from the research community over the last years. A variety of techniques and implementations have been proposed in this area, some of which have become commercial products. However, studies have revealed that these techniques have failed to gain widespread adoption, mainly because of high false-positive rates and the fact that manual inspection of alarms is a time consuming and error prone task.

Senatus is a new technique for combined anomaly detection and root-cause analysis, originally proposed by Atef Abdelkefi. In this thesis we further enhance this approach, e.g. by providing an algorithm for automated root-cause analysis. We evaluate Senatus' performance and compare it to a similar detector by using a three week data set collected from four different routers in the GÉANT2 network. Our evaluation shows promising results, where Senatus has the overall best detection and false-positive rates.

Furthermore, we provide a complete high-performance implementation of the Senatus framework. Our implementation is capable of analyzing a 15-minute time bin and identify the root-cause of its anomalies in just a few minutes. The implementation also includes a Dashboard, which is a highly valuable tool for a network operator to monitor network traffic and events generated by Senatus. The Dashboard also offers the network administrator a number of tools to visualize network traffic, which can reveal traffic patterns that indicate anomalous behavior.

Sammendrag

Trafikkanomalideteksjon i kjerne-nettverk har fått økt oppmerksomhet fra forskningsmiljøene de siste årene. En rekke teknikker og implementasjoner har blitt foreslått på dette området, hvorav noen har blitt kommersielle produkter. Studier har imidlertid avdekket at disse teknikkene ikke har klart å oppnå bred anvendelse, hovedsakelig på grunn av en høy andel falske alarmer og det faktum at manuell inspeksjon av alarmer er en tidkrevende oppgave som er sårbar for feil.

Senatus er en ny teknikk for kombinert anomalideteksjon og årsaksanalyse, opprinnelig foreslått av Atef Abdelkefi. I denne oppgaven forbedrer vi denne teknikken ytteligere, f.eks ved å foreslå en algoritme for automatisert årsaksanalyse. Vi evaluerer Senatus' ytelse og sammenligner den med en lignende detektor ved hjelp av et tre ukers datasett samlet inn fra fire forskjellige rutere i GEANT nettverket. Vår evaluering viser lovende resultater, hvor Senatus har den beste deteksjonsraten kombinert med lavest antall falske alarmer.

Videre har vi gjennomført en komplett implementering av Senatus rammeverket. Vår implementasjon er i stand til å analysere en 15-minutters periode med datatrafikk og identifisere årsaken til angrep i løpet av noen få minutter. Implementeringen omfatter også et web-grensesnitt, som er et verdifullt verktøy for en nettoperatør for å overvåke nettverkstrafikken og hendelser generert av Senatus. Webgrensesnittet gir også nettverksadministratoren en rekke verktøy for å visualisere nettverkstrafikk, noe som kan avsløre unormale trafikkmønstre.

Preface

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) for partial fulfillment of the requirements for the degree of master of science.

This work has been performed at the Department of Telematics, NTNU, Trondheim, with Yuming Jiang as professor and with Atef Abdelkefi as supervisor. We would like to thank our supervisor Atef Abdelkefi for all his time and effort helping us getting the best results and making this master thesis possible. We would also like to thank our professor Yuming Jiang for all his helpful advices.

Finally, we want to thank our family, friends and girlfriends for their motivation and support.

Abbreviations

AS Autonomous System

API Application Programming Interface

BGP Border Gateway Protocol

DoS Denial of Service

DOM Document Object Model

DDoS Distributed Denial of Service

FP False positive

FN False negative

HBD Histogram-based Detection

H1 Senatus H1

H2 Senatus H2

ICMP Internet Control Message Protocol

IP Internet Protocol

IS-IS Intermediate System To Intermediate System

KL Kullback-Leibler

ORM Object-relational mapping

PCA Principal Component Analysis

SNMP Simple Network Management Protocol

TP True positive

TN True negative

Contents

Problem Description	i
Abstract	iii
Sammendrag	v
Preface	vii
Abbreviations	ix
Contents	xi
List of Figures	xv
List of Tables	xvii
1 Introduction	1
1.1 Problem Statement	2
1.2 Contributions	3
1.2.1 High-performance Implementation	3
1.2.2 Web Dashboard	4
1.2.3 Performance Evaluation	4
1.2.4 Algorithm for Automatic Root-Cause Analysis	4
1.3 Methodology	5
1.3.1 Performance Evaluation	5
1.3.2 Implementation	5
1.4 Structure	6
2 Background	7
2.1 Anomalies	7
2.1.1 Denial of Service Attacks	7
2.1.2 Scans	10

2.1.3	Network Experiments	10
2.2	Existing Techniques for Anomaly Detection	11
2.2.1	Volume Based Techniques	11
2.2.2	Behavior Based Techniques	13
2.3	Existing Techniques for Root-Cause Analysis	20
2.3.1	Clustering Techniques	21
2.3.2	Signature Based Techniques	22
3	Framework - Senatus	25
3.1	Overview	25
3.2	Election	26
3.2.1	Senators Election	26
3.2.2	Traffic filtering reduces the K value	28
3.3	Voting	31
3.3.1	Senators Subspace Creation	31
3.3.2	Significant Change Detection	32
3.4	Decision	34
3.4.1	Anomalous Time Bins Detection	35
3.4.2	Suspicious Flows Identification	35
3.4.3	Root-Cause Analysis	37
3.5	Automatic Root-Cause Analysis	37
3.5.1	The Structure of the Algorithm	38
3.5.2	Setting the Right Threshold	39
4	Implementation	41
4.1	Senatus	42
4.1.1	Introduction	42
4.1.2	Version 0 (Offline Implementation)	44
4.1.3	C++ Implementation Technology	46
4.1.4	Version 1 (Offline Implementation)	49
4.1.5	Version 2 (Online Implementation)	50
4.2	Dashboard	57
4.2.1	Technology	57
4.2.2	Design	63
4.2.3	Examples	65
4.3	Automatic root-cause classification	65
5	Evaluation	71
5.1	Data set	71
5.2	Ground-truth construction	72
5.3	Tuning Parameters	73

5.3.1	Traffic Filtering Heuristics	73
5.3.2	The PCP Tuning Parameter λ	74
5.3.3	Root-Cause Thresholds	75
5.4	Performance Analysis	84
5.4.1	Methodology	84
5.4.2	Detected Anomalous Time Bins	85
5.4.3	Detection rates	86
5.4.4	False Positive Rates	86
5.4.5	Detected Attacks	87
5.5	Implementation Performance Evaluation	88
5.5.1	Online versus Offline Version	89
5.5.2	Different Links	89
5.5.3	Measurement Parameters	90
5.5.4	Tuning λ	91
5.5.5	Tuning K	92
5.5.6	Tuning t	93
5.5.7	Tuning Auto and Decision Rules	93
5.5.8	Tuning Heuristics (H1,H2 and α,β)	93
6	Discussion	95
6.1	Evaluation	95
6.1.1	Detection Methods Comparison	95
6.1.2	Automatic Root-Cause Analysis	96
6.2	Implementation Performance	97
6.2.1	Execution Time as Function of Number of Average Flows	97
6.2.2	Improvement in Execution Time from Database Storage	97
6.2.3	Choice of Default Parameters	97
6.3	Geographic Distribution of Anomalies	99
6.4	Existing Commercial Products	100
6.4.1	NetReflex	102
6.4.2	StealthWatch	102
6.4.3	Peakflow SP	103
6.5	Limitations	103
6.5.1	Limitations in the Senatus Framework	103
6.5.2	Limitations of the Implementation	104
7	Conclusion	107
	Bibliography	109

List of Figures

1.1	Illustration of relationship between anomaly detection and root-cause analysis.	3
2.1	Illustration of the effect a DDoS attack can have on network infrastructure	8
2.2	Visual illustration of a DDoS attack	8
2.3	Sample DDoS attack, as seen in Flow-tools	9
2.4	Sample network scan, as seen in flow-tools	11
2.5	Illustration of difference between port scan and network scan	12
2.6	Histogram example	14
2.7	Example showing how a network scan affects a feature distribution	15
2.8	Iterative algorithm of the HBD	19
2.9	Data clustering illustration	21
3.1	Illustration of the Senatus design	26
3.2	Traffic per feature values distributions over 24 hours	27
3.3	Motivation behind the "top"- K approach	29
3.4	Impact of filtering on the K value	30
3.5	Illustration of the distribution of traffic for the senators for the destination port feature distribution over a 24 hour period, and the time series for the senator port 80	32
3.6	Scree plots (src/dst AS/ports for the four collected traces)	33
3.7	Illustration of the Decision process	36
4.1	A use-case diagram of the intended Senatus system.	41
4.2	High-level overview of the Senatus design	42
4.3	Flow diagram of version 0.	44
4.4	Value matrix for one feature.	45
4.5	Flow diagram of version 1.	49
4.6	Flow diagram of version 2.	51
4.7	Per-feature operations, the value matrix building in version 2.	52

4.8	Change in selection between v1 and v2.	53
4.9	ER-model of the Senatus-relevant parts of the database.	55
4.10	Diagram of procedure oriented part of Senatus.	58
4.11	Class diagram of classes in Senatus.	59
4.12	Google Chart Tools used for visualizing time bin info.	61
4.13	Database additions for the Dashboard.	63
4.14	Use-case for the Senatus Dashboard.	64
4.15	The event list.	66
4.16	Details of an attack.	66
4.17	Alarm info tension graph for DDoS	67
4.18	Alarm info tension graph for network scan	68
4.19	Heat graph showing a Network Scan from 195.242.166.16.	69
4.20	The configuration options for the Dashboard.	70
5.1	The GÉANT2 network [1]	72
5.2	Distribution of the average flow size	74
5.3	Detection and false-positive rates as a function of C	75
5.4	Version 1 classification rates	78
5.5	Version 1.2 classification rates	78
5.6	Version 2 classification rates	80
5.7	Version 3 classification rates	81
5.8	Illustration of the detection metrics.	85
5.9	Overview of False positive rates for the different links	87
5.10	Detected attacks on all links combined	88
5.11	Detected attacks on each link	88
5.12	Total execution time, offline vs. online version.	89
5.13	Execution time for Vienna and Copenhagen.	90
5.14	Performance as a function of λ	92
5.15	Performance as a function of K	92
5.16	Performance as a function of t	93
5.17	Performance as a function of auto and decision rules	94
5.18	Performance as a function of α and of β	94
6.1	Geographic distribution of the sources of network attacks found in our dataset.	99
6.2	Geographic distribution of the targets of DDoS attacks found in our dataset.	100
A.1	TimebinCollection and extracting senators	120
A.2	Election: Getting senator values from database	121
A.3	Voting and Decision	121

List of Tables

2.1	Table showing which feature distributions that is affected by various anomalies, when using entropy [2]	16
2.2	Examples of attributes derived from anomalies, and used to create signatures [3]	22
3.1	Anomalies definition	30
3.2	Heuristics	31
3.3	Decision rules	35
3.4	Anomaly characteristics	38
5.1	Overview of the measurement links	72
5.2	Tuning parameters	73
5.3	Automatic Root-cause Analysis Performance	83
5.4	Detected Anomalous Time Bins	86
5.5	Overall Detection Rates	86
5.6	Parameter values used for reference.	91

Chapter 1

Introduction

The Internet is continuously growing in number of end-users and traffic volume. For network operators, it is of utmost importance to maintain network performance and security to be able to provide their customers with the Quality of Service they require. Network traffic anomalies are commonplace in today's Internet, and represent a significant challenge that network operators have to deal with. An anomaly is defined as *a significant change or deviation from what is considered normal* [4], and can have implications for both network operators and end-users. From a network operator's perspective, an anomaly can cause resource exhaustion leading to significant performance degradation. Other types of anomalies do not necessarily affect network performance, but can have consequences for companies that are dependent on their online resources to generate revenue.

The foremost challenge is that anomalies can be caused by a myriad of events, including malicious attacks, worm propagation, network experiments and abrupt changes caused by legitimate network traffic. This diversity, combined with the high variability of normal Internet traffic, makes detecting and identifying network anomalies a challenging task.

1.1 Problem Statement

Detecting and identifying network anomalies have been widely studied by the research community in later years. Anomaly detection is the process of generating an alarm for a detected anomaly, possibly with some evidence (e.g. IP addresses) related to the anomaly attached to it.

The first techniques for anomaly detection were based on the assumption that anomalies could be detected by monitoring how different volume-metrics changed over time. However, since not all anomalies introduce significant changes in volume metrics, more recent techniques have focused on studying network traffic *behavior*. Behavior-based techniques facilitate a more fine-grained insight to network traffic by studying the specific meta-data, called *features*, derived from the packet header of each flow. Commonly used features are IP addresses and port numbers, and by monitoring how traffic to these features changes over time, it is possible to detect low-intensity anomalies. However, there are some challenges that need to be addressed. Features are highly dimensional objects, and working with such dimensions are not computationally feasible, taken into account that it is critical that further actions can be initiated as fast as possible. Therefore, dimensionality reduction techniques need to be applied, with the objective of removing as much as possible of the "normal" traffic, while keeping any traffic that is likely to contain anomalies. Volume- and behavior-based techniques will be further discussed in Chapter 2.

Alarms generated by an anomaly detector need further investigation, either manually by a network administrator or by machine techniques. Before initiating appropriate countermeasures, the *reason* why an alarm was raised (e.g. a Denial of Service attack) needs to be addressed. This process is known as root-cause analysis. In manual root-cause analysis, the network administrator goes through the network logs from when the alarm was generated, and searches for anomalies. This is a time-consuming task, especially if the anomaly detector does not provide useful meta-data that can reduce the set of candidate flows. Automated root-cause analysis takes this burden off the network administrator, by applying techniques that can identify the reason why the alarm was raised, and output this information to the network administrator. We introduce existing techniques for automated root-cause analysis in Section 2.3.

Figure 1.1 gives a high-level illustration of the relationship between anomaly detection and root-cause analysis. A certain root-cause will result in anomalous flows, which might cause the anomaly detector to raise an alarm. The

alarm in addition to the flow records is used in the root-cause analysis, which tries to identify the reason why the alarm was raised.

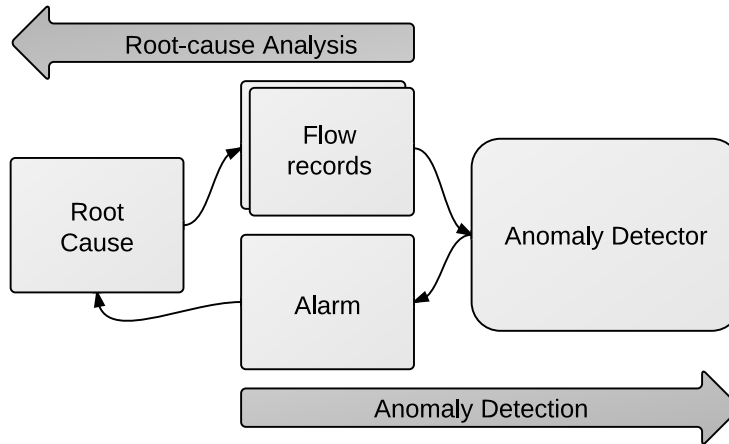


Figure 1.1: High-level illustration of the relationship between anomaly detection and root-cause analysis

1.2 Contributions

Our contribution to Senatus as a result of this thesis is the following:

- A high-performance implementation of Senatus
- Web Dashboard for Senatus
- Performance evaluation
- Algorithm for automatic root-cause analysis

1.2.1 High-performance Implementation

We have taken Senatus from a "proof-of-concept" version divided in three separate parts, to a complete high-performance anomaly detection and root-cause analysis system. The system will automatically analyze periodical network traffic data without any intervention from the network administrator. The root-cause of the anomalies will be automatically identified and stored in a database for presentation in the Senatus Web Dashboard.

1.2.2 Web Dashboard

The Web Dashboard offers easy access to important events created by the Senatus back-end, and allows a network administrator to set a number of configuration settings. For alarms where Senatus is unable to identify the root-cause, the Dashboard offers a number of different visualizations of the network traffic, which can reveal patterns that indicate anomalies. In addition, the Dashboard gives access to raw flow data, which the network administrator can use to manually label attacks that Senatus is unable to identify.

1.2.3 Performance Evaluation

To be able to perform a comprehensive performance evaluation of Senatus, we have constructed a ground truth by manually analyzing a large number of alarms. These alarms are generated by Senatus and an implementation of a well-known histogram based anomaly detection technique. Our dataset consist of flow data from 4 different links in the GEANT network. Over 8000 alarms in 1300 time bins were analyzed. A small portion of this work had already been done by other students, but since their work did not include any specific information about each anomaly, we had to redo much of their work. More specifically, to be able to measure the intensity and flow count of an anomaly, we needed the specific IP and port numbers involved. In most cases, the previous work had just marked the time bin with the type, e.g. DDoS, but not included any specific information. Therefore, we had to investigate the time bin again to find this information.

1.2.4 Algorithm for Automatic Root-Cause Analysis

Through the work in this thesis and many discussions with our supervisor, we came up with a number of ideas that have been implemented in Senatus. One of these include the algorithm for automated root-cause analysis. The original idea was to make an algorithm that could verify our results from the manual root-cause analysis, since this process can be prone to human errors. The results from the first version of the algorithm showed surprisingly similar results when compared to our manual root-cause analysis, thus we decided to further enhance the algorithm and implement it in Senatus.

1.3 Methodology

Since the work in this thesis consists of two independent parts, the methodology is split in two:

1.3.1 Performance Evaluation

Our way of evaluating the performance of Senatus was to compare the generated alarms with the alarms from another anomaly detector. We did this by manually analyzing the alarms from Senatus and a similar approach, we call it HBD, to construct a ground truth. Based on the ground truth, we were able to evaluate our anomaly detector while deriving important metrics for an anomaly detector, e.g. detection and false-positive rates, and compare the results of the evaluation to HBD. A more detailed description of the methodology can be found in Section 5.2

1.3.2 Implementation

The high-performance implementation was done in three major parts:

1. The existing implementation was studied to get a deep understanding.
2. An exact copy of the existing implementation in the selected programming language was created.
3. Senatus was changed and enhanced to make the implementation perform well in a network administration environment.

The implementation was done in incremental steps with each incrementation bringing new features, always leaving the program in a working state.

The Dashboard was designed and implemented to fully utilize the features of Senatus, and was thus changed when necessary to accommodate changes in Senatus.

1.4 Structure

The rest of the thesis is structured as follows:

Chapter 2 - Background introduces the anomalies we are interested in, and provides insight on existing techniques for anomaly detection and root-cause analysis. It also gives a more detailed explanation of the histogram based anomaly detection technique we compare to Senatus.

Chapter 3 - Framework gives a detailed explanation of the Senatus framework.

Chapter 4 - Implementation gives a thorough explanation of the implementation of Senatus and the corresponding Dashboard.

Chapter 5 - Performance Evaluation consists of a performance evaluation of Senatus and its tuning parameters, and compares the results with a similar histogram based anomaly detector.

Chapter 6 - Discussion discusses some of the limitations and drawbacks with Senatus, based on the results of the performance evaluation. We also introduce a few commercial anomaly detection products, and compare them to our implementation of Senatus. In addition to this, we discuss open problems and interesting topics for future work.

Chapter 7 - Conclusion gives a summary of the work in this thesis.

Chapter 2

Background

This chapter introduces the classes of anomalies we are interested in, and describes existing techniques for anomaly detection and root-cause analysis.

2.1 Anomalies

Network traffic anomalies are commonplace in today's Internet. In this section we introduce the two main anomaly classes we are interested in; Denial of Service attacks and scan attacks. We also introduce an anomaly class called network experiments.

2.1.1 Denial of Service Attacks

Denial of Service (DoS) attacks are a prominent threat to cyber infrastructure. The goal of a DoS attack is to prevent legitimate use of a digital resource [5]. An attack that uses several sources to obtain this goal is called a Distributed Denial of Service (DDoS) attack.

Examples of (D)DoS attacks include attempts to [6]:

- flood a network with traffic to prevent legitimate network traffic.
- disrupt connections between two machines, thereby preventing access to a service
- prevent a particular individual from accessing a service
- disrupt service to a specific system.

Figure 2.1 illustrates the effect of a high-intensity DDoS, seen in our dataset. The attack, starting around 08.30AM, causes the infrastructure to collapse, resulting in downtime of approximately one hour.



Figure 2.1: Illustration of the effect a DDoS attack can have on network infrastructure

Several well-known institutions and companies have been victims of successful denial of service attacks. Examples include the European Parliament [7], Visa/Mastercard [8] and NASDAQ [9]. DDoS attacks do not necessarily target a single resource; In September 2002 the infrastructure of the Internet was the target in a DDoS attack against 9 of 13 root servers on the Internet [10].

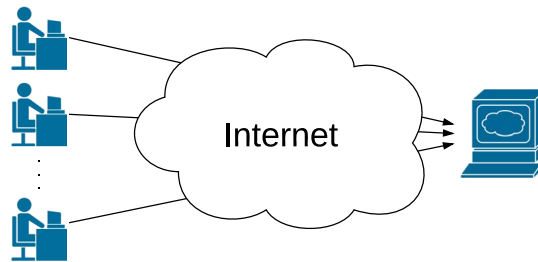


Figure 2.2: Visual illustration of a DDoS attack

Based on the degree of automation used to perform an attack, we can differentiate between *manual*, *semi-automatic* and *automatic* DDoS attacks [5]. In a manual attack, each of the participating computers is first scanned for vulnerabilities, which is exploited to install attack code. The attack code is then manually activated to perform the attack against the victim.

In a semi-automatic or automatic attack the recruit, exploit and infect phases are automated, resulting in a zombie-network of computers also known as a botnet. This process is unwillingly performed by a user e.g. by opening an attachment in an email or clicking a link on a webpage. These zombie-networks are controlled by a botnet host. In a semi-automatic attack, the computers in the zombie-network wait for information about the time, victim and attack type from a botnet host, and then perform the attack. In an automatic attack, this information is preprogrammed in the attack code. Most attacks nowadays are either semi-automatic or automatic [5].

A particular type of denial of service attack is known as a **SYN flood attack**, which exploits the TCP 3-way handshake. In a SYN flood attack, the attacker(s) attempt to overwhelm the victim with a constant stream of TCP connection requests (SYN packets). Each of these request is consuming one out of a limited number of available resources, until all the available resources are occupied [11]. The effect of this is that legitimate users will be denied access. Figure 2.3 shows a sample DDoS attack, where the host 193.140.255.1 is flooded with TCP SYN packets.

```

0623.21:35:26.546 0623.21:35:26.546 185 152.113.94.64 45623 154 193.140.255.1 80 6 2 1 48
0623.21:35:29.796 0623.21:35:29.796 185 175.106.178.19 18915 154 193.140.255.1 80 6 2 1 48
0623.21:35:29.516 0623.21:35:29.516 185 33.63.162.32 62382 154 193.140.255.1 80 6 2 1 48
0623.21:35:20.036 0623.21:35:20.036 185 163.25.124.59 59796 154 193.140.255.1 80 6 2 1 48
0623.21:35:07.591 0623.21:35:07.591 185 73.205.55.94 65319 154 193.140.255.1 80 6 2 1 48
0623.21:35:27.566 0623.21:35:27.566 185 80.8.154.54 9287 154 193.140.255.1 80 6 2 1 48
0623.21:35:24.426 0623.21:35:24.426 185 158.108.119.123 62400 154 193.140.255.1 80 6 2 1 48
0623.21:35:12.525 0623.21:35:12.525 185 94.113.33.111 50866 154 193.140.255.1 80 6 2 1 48
0623.21:35:13.475 0623.21:35:13.475 185 172.209.238.68 222 154 193.140.255.1 80 6 2 1 48
0623.21:35:12.915 0623.21:35:12.915 185 12.206.70.121 50867 154 193.140.255.1 80 6 2 1 48
0623.21:35:20.486 0623.21:35:20.486 185 71.241.45.1 8468 154 193.140.255.1 80 6 2 1 48
0623.21:35:17.396 0623.21:35:17.396 185 200.233.60.79 39917 154 193.140.255.1 80 6 2 1 48
0623.21:35:15.296 0623.21:35:15.296 185 96.60.168.12 58829 154 193.140.255.1 80 6 2 1 48
0623.21:35:12.505 0623.21:35:12.505 185 81.238.72.114 1963 154 193.140.255.1 80 6 2 1 48
0623.21:35:07.741 0623.21:35:07.741 185 199.96.251.78 38241 154 193.140.255.1 80 6 2 1 48
0623.21:35:29.136 0623.21:35:29.136 185 88.158.100.79 59843 154 193.140.255.1 80 6 2 1 48
0623.21:35:24.766 0623.21:35:24.766 185 155.236.101.51 11375 154 193.140.255.1 80 6 2 1 48
0623.21:35:26.036 0623.21:35:26.036 185 79.140.161.4 14946 154 193.140.255.1 80 6 2 1 48
0623.21:35:20.126 0623.21:35:20.126 185 191.89.72.112 19431 154 193.140.255.1 80 6 2 1 48
0623.21:35:12.065 0623.21:35:12.065 185 204.185.108.38 18935 154 193.140.255.1 80 6 2 1 48

```

Figure 2.3: Sample DDoS attack, as seen in Flow-tools

A variant of ICMP flooding, called **Smurf attack** [12], is an amplification attack where large amounts of ICMP echo messages are sent to one or more IP broadcast addresses. The ICMP messages are spoofed with the address of the victim as the source address. In most networks hosts will reply to this ICMP echo message, with the result that each ICMP message sent from the attacker(s) to the broadcast address will generate large amounts of traffic to the victim. In a Smurf attack two entities will suffer; the victim, which will be overloaded with traffic, and the target network itself, where the large amount of ICMP traffic may prevent legitimate traffic from propagating.

UDP flooding attacks exploits the UDP echo and character generator running on most computers. A forged UDP packet is used by the attacker to connect the echo service on one computer to the character generator on another computer. This attack generates large amounts of network traffic, and may consume all available network bandwidth, preventing legitimate traffic [13].

2.1.2 Scans

The main objective of scan attacks is to identify running services, and is often a precursor of other more harmful events, e.g. worm propagation. Scan attacks can be divided into two categories with slightly different objective.

2.1.2.1 Port Scans

A port scan tries to identify which services that are running on a specific host, and is performed by sending probes to a range of ports on one or more hosts [14]. A reply from the target will identify a running service on that specific port, which may be exploited by an attacker.

2.1.2.2 Network Scans

The goal of a network scan is to identify hosts that are running a specific service. It is carried out by sending probes to a specific port to a range of hosts [15]. Network scans are in some cases the result of worm activity on a infected host, where the worm searches for hosts running a vulnerable service. Figure 2.4 shows a sample network scan, where the host 219.243.47.162 is sweeping for port 1433, which is often an indication of the MS SQL worm [16].

2.1.3 Network Experiments

In our dataset, we experienced large amounts of irregular network traffic passing through certain gateways, which did not match the signature of any known attacks. By closer examination, we learned that this had its cause in various network experiments originating from Planetlab [17]. Planetlab is a research network where industrial research labs and academic institutions contribute with hardware to facilitate testing of new network services across geographic locations. More specifically, the majority of the irregular traffic was generated by the CoDeeN network [18], which is a collection of proxy servers hosted at Planetlab nodes. These proxy servers frequently

0620.11:27:02.022	0620.11:27:02.022	148	219.243.47.162	6000	140	161.116.219.166	1433	6	2	1	40
0620.11:27:00.202	0620.11:27:00.202	148	219.243.47.162	6000	140	161.116.43.204	1433	6	2	1	40
0620.11:27:00.402	0620.11:27:00.402	148	219.243.47.162	6000	140	161.116.63.107	1433	6	2	1	40
0620.11:27:01.602	0620.11:27:01.602	148	219.243.47.162	6000	140	161.116.179.16	1433	6	2	1	40
0620.11:27:00.372	0620.11:27:00.382	148	219.243.47.162	6000	140	161.116.60.133	1433	6	2	1	40
0620.11:27:01.012	0620.11:27:01.012	148	219.243.47.162	6000	140	161.116.122.183	1433	6	2	1	40
0620.11:27:01.522	0620.11:27:01.522	148	219.243.47.162	6000	140	161.116.171.198	1433	6	2	1	40
0620.11:27:02.342	0620.11:27:02.342	148	219.243.47.162	6000	140	161.116.249.76	1433	6	2	1	40
0620.11:27:01.892	0620.11:27:01.892	148	219.243.47.162	6000	140	161.116.207.2	1433	6	2	1	40
0620.11:27:01.252	0620.11:27:01.252	148	219.243.47.162	6000	140	161.116.146.2	1433	6	2	1	40
0620.11:27:02.222	0620.11:27:02.222	148	219.243.47.162	6000	140	161.116.238.15	1433	6	2	1	40
0620.11:27:00.312	0620.11:27:00.312	148	219.243.47.162	6000	140	161.116.55.38	1433	6	2	1	40
0620.11:27:01.252	0620.11:27:01.252	148	219.243.47.162	6000	140	161.116.145.243	1433	6	2	1	40
0620.11:27:00.882	0620.11:27:00.882	148	219.243.47.162	6000	140	161.116.110.14	1433	6	2	1	40
0620.11:27:01.082	0620.11:27:01.082	148	219.243.47.162	6000	140	161.116.129.105	1433	6	2	1	40
0620.11:27:01.852	0620.11:27:01.852	148	219.243.47.162	6000	140	161.116.203.53	1433	6	2	1	40
0620.11:27:01.572	0620.11:27:01.572	148	219.243.47.162	6000	140	161.116.175.241	1433	6	2	1	40
0620.11:27:01.402	0620.11:27:01.402	148	219.243.47.162	6000	140	161.116.160.123	1433	6	2	1	40
0620.11:27:00.022	0620.11:27:00.022	148	219.243.47.162	6000	140	161.116.26.81	1433	6	2	1	40
0620.11:27:01.441	0620.11:27:01.441	148	219.243.47.162	6000	140	161.116.163.138	1433	6	2	1	40
0620.11:27:00.011	0620.11:27:00.011	148	219.243.47.162	6000	140	161.116.25.160	1433	6	2	1	40

Figure 2.4: Sample network scan, as seen in flow-tools

sent UDP probes to each other to measure network performance. Network experiments are a type of anomalies that not necessarily have malicious intents, but for a network provider it may still result in negative effects like network performance degradation.

2.2 Existing Techniques for Anomaly Detection

Anomaly detection is the process of identifying unexpected events in network traffic that deviate from what is considered as normal [19]. This has been an area of extensive research in later years, where several different techniques have been proposed, some have also been implemented in commercial products. This section will divide existing techniques into two distinct groups based on their detection metrics, and give examples of existing approaches.

2.2.1 Volume Based Techniques

Volume based anomaly detection techniques use volume metrics to model the traffic, e.g. number of bytes, number of packets, number of flows etc. Volume metrics is used on the intuition that the sudden changes in network traffic caused by anomalies will be detected when monitoring these metrics over time. A number of techniques can be applied to detect abrupt variations in traffic time series. Lakhina et al. proposes a general technique to diagnose volume anomalies in [4]. Their method applies Principal Component Analysis (PCA) on observed link-based statistics. PCA is a

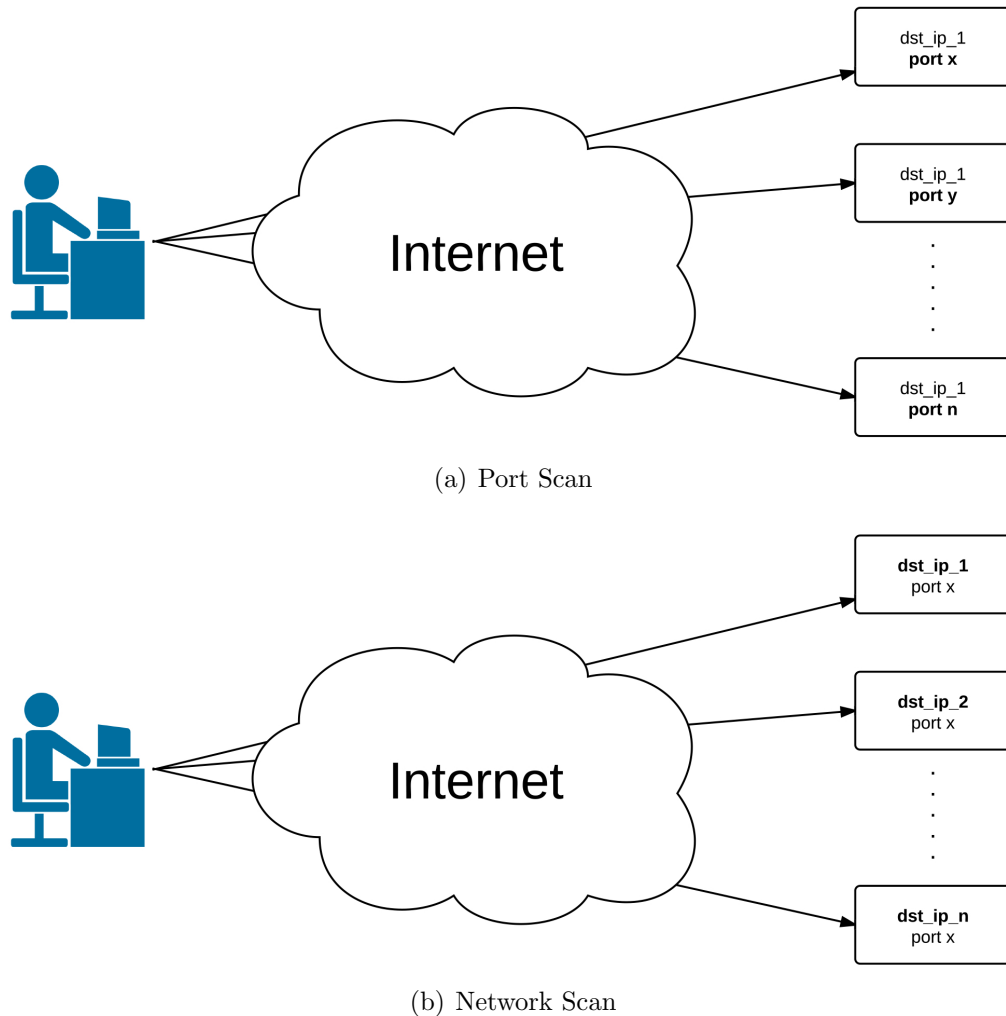


Figure 2.5: Illustration of the difference between port scan and network scan

statistical-analysis technique for detecting network traffic anomalies, and performs dimensionality-reduction to return a compact representation of a multidimensional dataset [20]. PCA is used in several anomaly detection techniques [21, 22], with the purpose of dividing normal traffic and anomalies into separate subspaces. The technique proposed by Lakhina et al. identifies the anomaly that is closest to representing the deviation from the normal subspace, and flags traffic in this subspace. Barford et al. presents another approach using volume metrics where signal processing techniques is applied to identify anomalies . More specifically, wavelet analysis is used

to remove the predictable ambient part, and then flag an anomaly if a disruption is above a fixed threshold value [23].

The drawback with techniques based on volume metrics is that not all anomalies introduce a significant change in any volume metrics, e.g. low-intensity port scans and network scans. Another drawback is that volume metrics do not provide enough information about an anomaly, e.g. IP or ports, which is needed for root-cause analysis [2].

2.2.2 Behavior Based Techniques

To overcome the shortcomings of volume based techniques, behavior based detection techniques are designed on the observation that anomalies affect the distribution of the traffic feature values that represents the anomaly. Commonly used traffic feature values are extracted from the IP header fields and includes destination/source AS, destination/source IP addresses, destination/source port numbers, flow/packet sizes and TCP flags [24]. The intuition behind using traffic features is that anomalies will affect feature distributions; During a network scan the feature distribution of source ports will be concentrated on a specific port and the feature distribution of destination addresses will be dispersed. Similarly, a DDoS attack will induce a concentration on the victim address in the destination address feature distribution, and dispersion on the source address feature distribution.

2.2.2.1 Histograms

One way of implementing behavior based detection is by using histograms. A feature histogram represents the distribution of the amount of traffic (or the number of flows) over the possible values of the chosen traffic features [25]. Figure 2.6 gives an example of a histogram representing the distribution of the destination port feature. Mathematically, a histogram can be defined as [2]: $X = \{n_i, i = 1, \dots, N\}$, meaning that feature value i occurs n_i number of times.

The main idea behind using histograms in anomaly detection is that:

1. Regular traffic patterns that represents the behavior of a network will be captured in the histograms
2. An anomaly will distort these traffic patterns, hence be visible in the histograms

Figure 2.7 shows the histograms for the destination port feature for July 2nd and July 3rd from 22:15 to 22:30. In the lower plot there is an ongoing network scan for destination port 22, which is clearly visible by comparing the bar that represents port 22 in the two histograms.

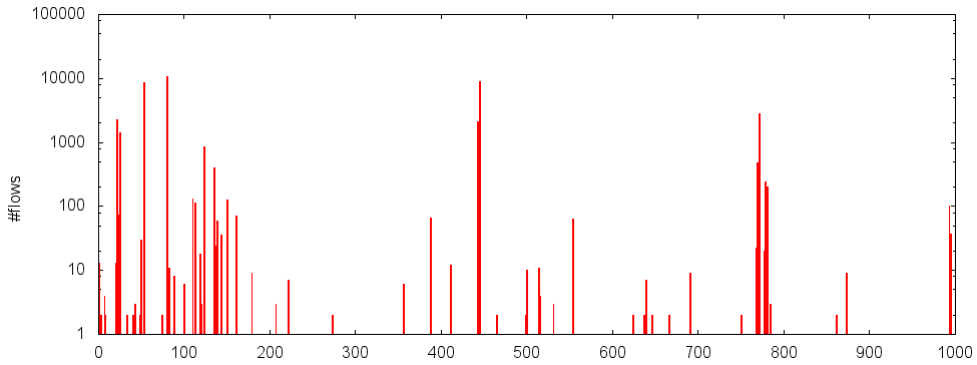


Figure 2.6: Histogram illustrating the distribution of first 1000 destination ports during a 15 minute time bin at the Copenhagen gateway the 20th June 2011

Challenges with histograms

A challenge that all implementations that make use of feature histograms have to deal with is the *curse of dimensionality* issue [25]. The problem is that commonly used features, e.g. IP addresses and ports, will result in histograms containing a high number of entries. Uhlig et al. shows that over a 15-minute measurement period, these numbers can be as high as 2^{16} ports and even higher for IP-addresses, which will result in vectors of a size that is computationally very expensive to deal with [26]. Traffic sampling, either on packet or flow basis, is commonly deployed in traffic monitoring. However, sampling does not significantly reduce the number of values in the histograms [27]. Implementations that makes use of feature histograms therefore needs to perform dimensionality reduction. The following sections will introduce commonly used dimensionality reduction techniques, and explain some of the challenges related to each of them.

2.2.2.2 Entropy

One way of performing dimensionality reduction is by summarizing a feature distribution in a single entropy value. Lakhina et al. introduces the idea of using entropy as a tool to capture the distributional changes in traffic features in [2]. Sample entropy is used as a metric because it can capture

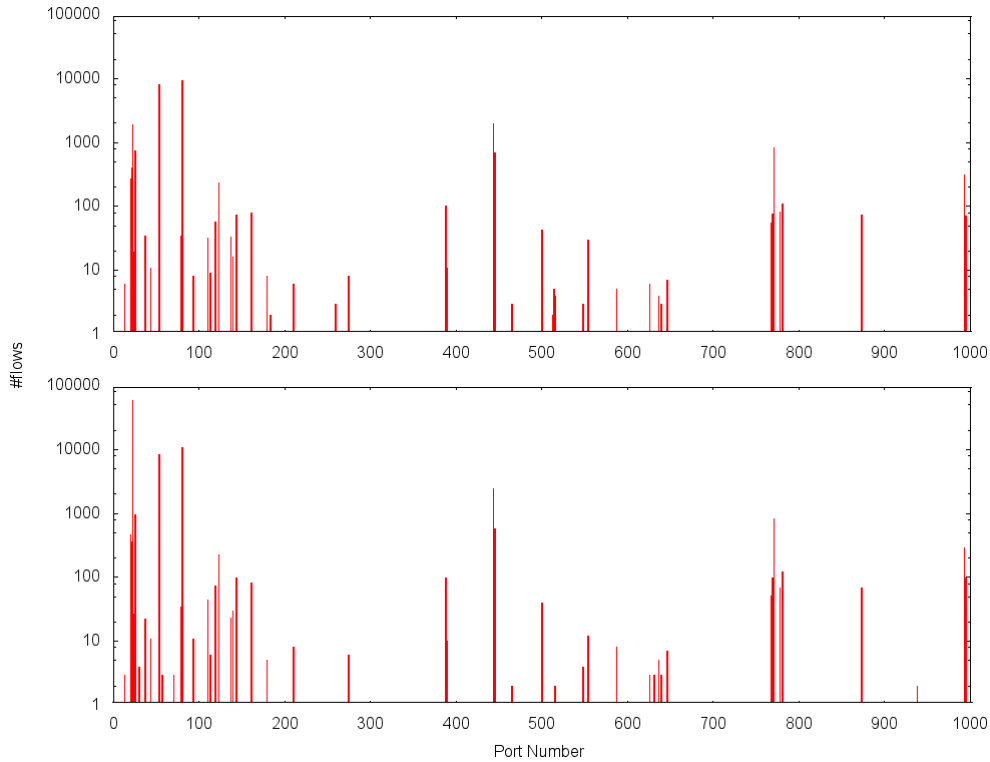


Figure 2.7: Illustration of how histograms get distorted as a result of anomalies. The two plots show the distribution of the 1000 first destination ports during the same time bin at two consecutive days. At the lower plot, there is an ongoing port sweep to destination port 22, which is clearly visible in the histogram.

both dispersal and concentration of a feature distribution. The sample entropy formula is defined as:

$$H(X) = - \sum_{i=1}^N \left(\frac{n_i}{S} \right) \log_2 \frac{n_i}{S} \quad (2.1)$$

X denotes an empirical histogram, where i occurs n_i times and

$$S = \sum_{i=1}^N n_i \quad (2.2)$$

Anomaly	Description	Effected Distribution
Alpha Flows	Unusually large volume point-to-point flow	Source address Destination address
(D)DoS	Denial of Service Attacks	Destination address Source address
Flash Crowd	Unusual burst of traffic to single destination, from a "typical" distribution of sources	Destination address Destination port
Port Scan	Probes to many destination ports on a small set of destination addresses	Destination address Destination port
Network Scan	Probes to many destination addresses on a small set of destination ports	Destination address Destination port

Table 2.1: Table showing which feature distributions that is affected by various anomalies, when using entropy [2]

is the total number of observations in the histogram [2]. Entropy values range from 0 for a maximally concentrated distribution, to $\log_2 N$ for a maximally dispersed distribution. A maximally dispersed distribution will occur if the N distinct values (i.e. IP addresses) in the distribution appear exactly 1 time, while a maximally concentrated distribution occurs when all observation is related to the same feature value. This is utilized in the detection phase, where a sudden change in the time series for entropy values indicates anomalies. Table 2.1 shows how various anomalies induce changes in different feature distributions. The observation that anomalies affect feature distributions in diverse manners can also be used for anomaly classification, although this approach will not be very precise since different anomalies can affect the same feature distributions.

A drawback with techniques that uses entropy to capture variations on feature distributions is that it lacks the ability to output specific feature values involved in an anomaly, since the entire feature is represented by *one* value. Furthermore, [28] indicates that the choice of address and port as features may not be the best option in entropy based techniques, arguing that more complementary features, like behavioral distributions (number of distinct IP addresses that a host communicates with) and the flow size distributions, may result in increased anomaly detection capabilities.

2.2.2.3 Relative uncertainty

An extension of sample entropy is proposed by Xu et al., where relative uncertainty (RU) is used for dimensionality reduction [29]. RU is defined as:

$$RU(X) = \frac{H(X)}{H_{max}(X)} \quad (2.3)$$

where $H(X)$ is defined in equation 2.1, and the maximum entropy $H_{max}(X) \equiv \log \min(N, S)$. An important property of RU is when $RU(X) \approx 1$, which implies no variations among the non-zero subset of feature values x_i . This is utilized by finding the number of feature values, K , which can approximately represent the entire feature distribution, thus reducing the dimensionality. To find this K value, the relative uncertainty is repeatedly calculated for $(x'_{K+1}, x'_{K+2}, \dots, x'_n) = R(K)$, starting from $K = 1$. This process is repeated with increasing K until $R(K)$ reaches a threshold value close to 1, meaning that the tail of the distribution is uniformly distributed. The challenge with this approach is that the K value will vary significantly between features as well as successive days, which may lead to a dramatic difference in approximation error when using the obtained K coefficients to represent the histograms [30].

2.2.2.4 Hashing / Aggregation

Previous work has adopted various aggregation strategies to decrease the set of dimensions, e.g. by aggregating flows into Origin-Destination flows [2]. However, aggregation strategies only identifies the aggregated anomalous flows, not the exact flow(s) responsible for the anomaly.

Hashing in general is done by using a hash function which maps data of any length into a fixed-length hash value [31]. Hashing can be utilized for dimensionality reduction by hashing a possibly large set of feature values into smaller sized bins. Although this can provide loss-less compression of histograms, it requires a mapping between the original histogram and the hashing function, which adds a significant processing overhead.

2.2.2.5 Histogram Based Detector (HBD)

The histogram based implementation used to compare against Senatus in this thesis is a simplified version based on the work of D. Brauckhoff et al. [32], where they propose an aggregation strategy using hashing for reducing dimensions of the histograms.

The main design ideas behind the implementation can be divided into the following three steps:

1. Histogram cloning
2. Voting
3. Flow pre-filtering

Histogram cloning and Detection

A histogram clone representing a feature consists of \mathbf{m} different bins, and applies a hash function to randomly place each feature value in one of the \mathbf{m} bins. Each feature has K different clones, created by independent hashing functions. The idea behind this is to obtain additional views of network traffic.

After the histogram clones is created, the Kullback-Leibler (KL) distance between the current distribution p and the previous distribution q is computed. The KL distance measures the similarity between two discrete distributions and is defined as [33]:

$$D(p||q) = \sum_{i=0}^m p_i \log(p_i/q_i).$$

Identical distributions will have a KL distance of 0 (since $\log(X/X) = 0$), while deviations will result in higher KL distance values. An alert is generated if

$$\Delta_t D(p||q) \geq 3\hat{\sigma}$$

where $\hat{\sigma}$ is a robust estimate of the median absolute deviation. When an alarm is generated, a set of affected histogram bins, B_k , and the corresponding set of feature values, V_k , need to be identified. An iterative algorithm that simulates removal of suspicious flows is applied to find the histogram bins that contribute to generating the alarm. The algorithm removes suspicious flows, starting with the bin with the largest absolute distance between the histogram of the previous and the current interval, until $\Delta_t D(p||q)$ falls below the detection threshold $3\hat{\sigma}$. This process is illustrated in Figure 2.8. The feature values V_k is obtained by keeping a map between the identified anomalous bins B_K and the feature values.

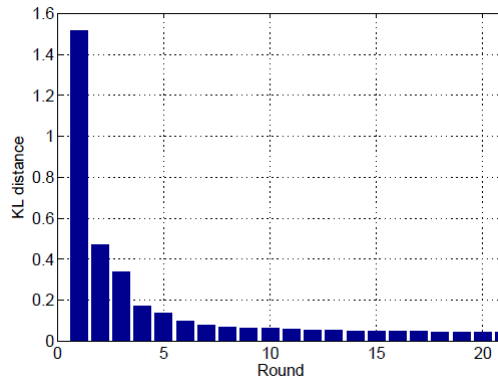


Figure 2.8: The iterative algorithm removes one bin in each round until the KL distance is below the detection threshold [32]

Voting and Meta-data Generation

In the voting process, a particular feature value is kept if it is selected by at least l out of the k clones. The parameter l can be used to manually adjust the ratio between false-positives and false-negatives. The voting process creates meta-data M_j , that contains both normal and anomalous traffic.

Flow Pre-filtering

The goal of the pre-filtering process is to remove as much of the benign traffic as possible, which is necessary to reduce the processing time in the next steps of the algorithm, and to minimize the false-positive item sets. Pre-filtering is done by using the union set of meta-data M_j to create a set of suspicious flows.

Association Rule Mining

An association rule is an expression $X \Rightarrow Y$ (where X and Y are item sets), saying that if a transaction T (containing a set of items) contains X , it is likely to also include Y [34]. Association rule mining can be employed in a variety of business applications, one example might be that 85% of customers that buys product A also buys product B, this information might be useful to increase profit by placing those two products next to each other in the store.

Association rule mining can also be used to solve the problem of anomaly extraction. The reason for this is the observation that anomalies results in a large number of flows with similar characteristics. For example a port sweep, where a certain source IP/AS and destination port will appear in a large portion of the flows. Brauckhoff et al. [32] decomposes the problem of

discovering all association rules in a dataset into two sub-parts: (i) Given a user-defined minimum support (*minsupport*), discover all item-sets that have a frequency above this value and (ii) based on these item-sets, derive association rules. The *minsupport* parameter is the minimum number of flows that contain all elements in the set X . A transaction in this context is a flow record and the items e_i , consisting of the 7 flow features srcIP, dstIP, srcPort, distort, protocol, #packets, #bytes. An item set $X = \{e_1, \dots, e_7\}$, is a combination of those 7 flow features, and the largest possible set is a 7-item set that contains a value pair for each flow feature. Each flow feature can be represented maximum one time.

For discovering frequent item-sets, the **Apriori algorithm** [35] is utilized, where the *minsupport* parameter is used as input. The choice of this parameter affects the false-positive and false-negative rates; a too low value will produce many item sets containing non-anomalous flows (FP), while a too high value might result in ignoring item sets that includes anomalies (FN). Apriori iterates over the dataset at most h times, and for each round l , where ($l = 1, \dots, h$), the l -item-sets with frequency over *minsupport* are selected. The algorithm stops when no l -item-sets are above the given min-support. To reduce the number of item-sets needed to be analyzed by a network administrator, Brauckhoff et al. slightly modifies the Apriori algorithm to output only the l -item-sets that are not a subset of a more specific $l+1$ -item-set.

Challenges

We experienced that association rule mining was a time-consuming process without giving substantially improved results, and was therefore not included in the implementation used in this thesis. In the rest of this thesis, we will refer to this implementation as HBD (Histogram Based Detector).

2.3 Existing Techniques for Root-Cause Analysis

Anomaly detection systems have the ability to generate alarms for events that might be important for a network administrator. Root-cause analysis is the process of identifying the *reason* why an alarm was raised [36]. This process is usually done by a network administrator, which uses his knowledge to examine the network logs where the alarm was flagged in search for events that need further action. Anomaly detection systems where a high percentage of the alarms is false-positives may be deemed as unusable by a network administrator, given the time needed to identify the root cause of

an alarm. Therefore there arises a need for techniques that automates the traffic anomaly root-cause analysis.

2.3.1 Clustering Techniques

The objective of clustering is to divide observations into homogeneous and distinct groups, based on a measure of similarity [37]. Clustering has a variety of applications, including data mining, automated root-cause analysis, and pattern classification. The idea behind using clustering in automated root-cause analysis is that anomalies with similar characteristics will be clustered into the same group, where each group is labeled with a root-cause.

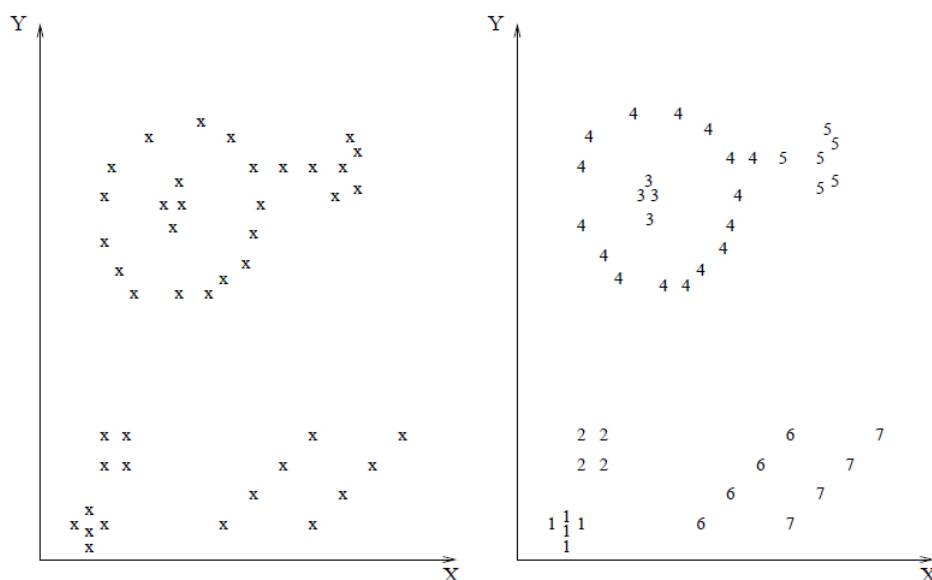


Figure 2.9: Example of data clustering. The right plot shows which distinct group each data element belongs to as a result of applying a clustering algorithm [38]

Silveira et al. propose a technique called Unsupervised Root Cause Analysis (URCA) [39], operating in two steps. The first step uses input from the anomaly detector to iteratively remove flows that exhibit normal traffic patterns until it ends up with the root-cause traffic. This traffic is used to model the behavior of the anomaly, which is used to classify the anomaly by using hierarchical clustering based on manually identified anomalies. Since the clustering is based on the behavior of manually identified anomalies,

there is no need for threshold values. However, if a new anomaly has a behavior that is different from the previously identified anomalies, this approach will fail to classify the new anomaly.

In general, an advantage with techniques based on clustering is that they do not require any threshold values, since the clustering is based on existing anomalies. However, this is also a disadvantage, since previously unseen anomalies will require a training period before they are correctly clustered.

2.3.2 Signature Based Techniques

Another type of automated root-cause analysis is based on signatures. A signature can be seen as well known "pattern" for how a specific anomaly behaves. Fernandes et al. demonstrate an approach based on the idea of using the identified anomalous flows to create the meta-data that characterizes the anomaly [3]. Information in the meta-data is used to create *attributes* representing the anomaly. Examples of attributes are shown in Table 2.2.

Attribute	Description
#respdest	Number of responsible destinations.
#rsrc / #rdst	Ratio of responsible sources to responsible destinations.
bpprop	Average packet size (only packets of the anomaly).
spprop	Ratio of number of syn to number of packets of the anomaly.
oneportpred	If only one destination port dominated.
invprotopred	If packets using invalid protocol numbers or types dominated.

Table 2.2: Examples of attributes derived from anomalies, and used to create signatures [3]

The main idea behind building signatures with attributes is that it offers the possibility for more fine-grained identification, e.g. identify the various types of DDoS attacks seen in today's Internet. The attributes can take either numerical or boolean values. An anomaly is classified by matching the attributes that represent an anomaly against the defined signatures. E.g. a network scan might be identified by a signature consisting of the attributes $\#respdest > 200$ and $samesrcpred$, which says that the anomaly contains over 200 unique destination addresses in that time bin, and that one source is responsible for most of these flows.

As with other techniques based on signatures, this technique requires signatures that are always up to date to perform satisfactory. Also, signature-based approaches might classify anomalies that do not match any of the defined signatures as false-positives. Thus, new anomalies without a well-known signature might be classified as false-positives, and pass by unnoticed.

Chapter 3

Framework - Senatus

Senatus is a new framework for joint anomaly detection and root cause analysis, originally proposed by Atef Abdelkefi [40]. The technique is inspired by a political body called a Senate (Senatus), which is a constellation of the eldest and wisest members of the society. The Senatus has the decision-power and decides the matters of the state.

3.1 Overview

Senatus is divided into three main distinct components. These components are predicated on the extraction of the top traffic feature values namely *senators*. The time series of the traffic per feature value are organized in a subspace called *senators subspace* and analyzed using sparse and low rank matrix decomposition (PCP) to detect significant changes. Detected abrupt variations called *votes* are thus collected and interpreted in the decision process to identify anomalous time bins and the set of responsible anomalous flows.

Practically, the three components are summarized as follows:

1. Senators election.
2. Senators votes.
3. Decision for a joint anomaly detection and root cause analysis.

A visual representation of Senatus framework can be seen in Figure 3.1.

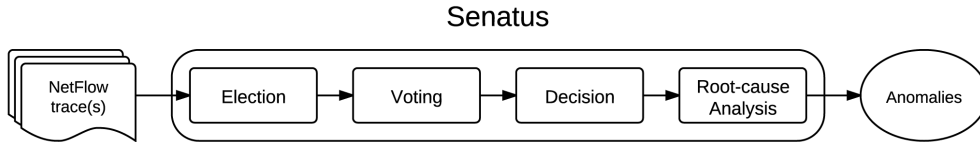


Figure 3.1: Illustration of the Senatus design

3.2 Election

As described in Chapter 2, traffic histograms have been widely adopted for traffic anomaly detection. The reason behind this is that traffic anomalies generally distort the normal pattern of traffic histograms. For example DDoS attacks may change the pattern of the histogram of traffic per source AS, while scan attacks may modify the histograms of traffic per destination ports. However, the main challenge dealing with traffic histograms is the *curse of dimensionality* issue described in Section 2.2.2.1.

Figure 3.2 illustrates the distribution of the reordered amount of flows per source port values over 24 hours for the collected traces. The figure shows that the histograms of traffic over src and dst port values are highly dimensional: around 2^{16} source and destination port values and tens of source and destination AS values, at each of the 15 min time bins. Dealing with a structure of such dimensions weakens the scalability of the analysis significantly. However, reducing such a high dimensionality using aggregation strategies or entropy, described in Section 2.2.2, results in a coarse-grained summary of the histograms characteristics [24], leading to an inaccurate analysis.

Our way out of this impasse, is based on the *compressibility* [30] of traffic histograms which we will discuss in details in this section.

3.2.1 Senators Election

It has been recently shown that traffic histograms are highly *compressible* structures due to the *power-law* decay when sorted [30]. Figure 3.2 verifies this observation in our collected data set. It shows that the amount of traffic per feature values exhibits a linear tendency in the log-log scale with a rapid decay when sorted. More formally, a compressible signal X

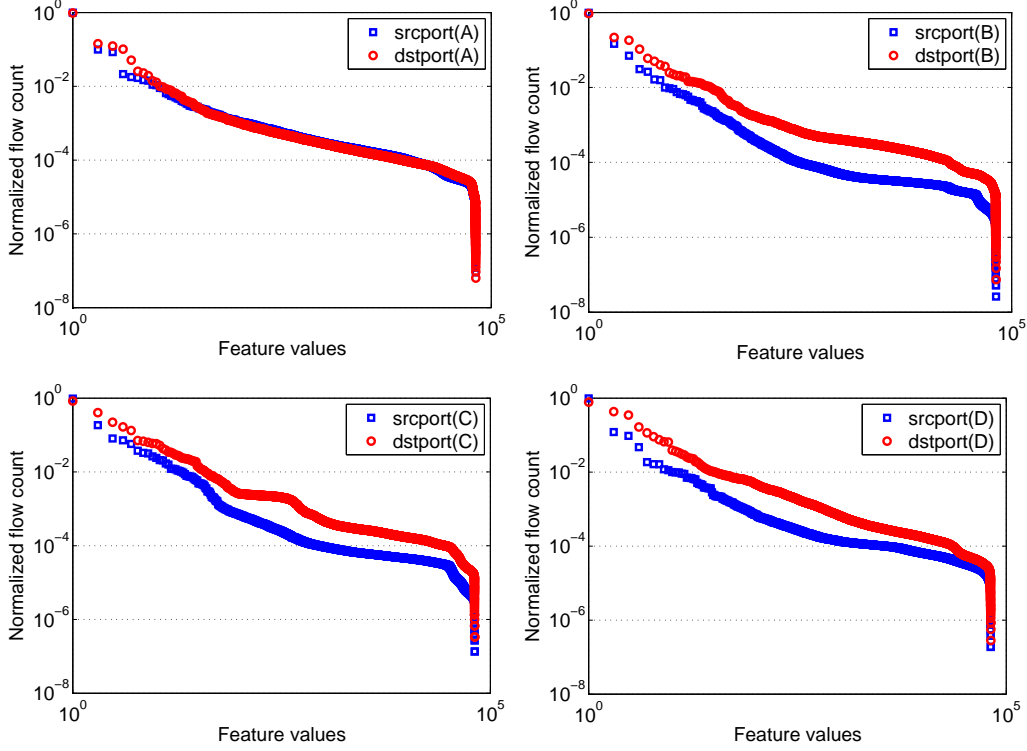


Figure 3.2: Traffic per feature values distributions over 24 hours

is a signal whose coefficients, when sorted in order of decaying magnitude $X' \equiv (x'_{(1)}, x'_{(2)}, \dots, x'_{(n)})$, $(x'_{(1)} \geq x'_{(2)} \geq \dots, x'_{(n)})$, decay according to the power law:

$$x'_{(i)} \leq Ri^{(\frac{-1}{p})}; i = 1..n, \quad (3.1)$$

where R is a normalization constant and $0 < p \leq 1$ is a scaling parameter. Note that p controls the rate of the decay of the coefficients of x , i.e, smaller p implies faster decay.

Thanks to the rapid decay of its coefficients, X can be closely approximated by the few first- K : “top”- K coefficients $(x'_{(i)}, i \in [1..K])$ (keeping the largest K coefficients and setting the remaining to zero), resulting in a **compressed representation** called the **best K -sparse approximation** X_K such that $K \ll N$.

Particularly, the *best K sparse approximation* has an error term [30]:

$$\sigma_K = \|X - X_K\|_2 \leq (ps)^{(-1/2)} RK^{(-s)}; s = \frac{1}{p} - \frac{1}{2}. \quad (3.2)$$

As such, accurate approximation of X can be constructed using a few number of coefficients.

This motivates our approach for senators election. More specifically, senators election consists on the extraction of the “top”- K feature values in each of the measurement time bins. The elected set of senators are, thus, feature values which appear among the “top”- K for at least one time bin during the observation window.

The extracted top components, which surprisingly achieve a low approximation error [30], do not only reduce traffic histograms dimension but are of inherent interest in our approach for anomaly detection and root-cause analysis. Figure 3.3 introduces the intuition behind adopting the top- K approach for traffic anomaly detection. It shows that traffic anomalies such as DoS/DDoS and scans tend to push the feature values carrying anomalous traffic toward the top feature values of the reordered histogram. This is of high interest since one can, on one hand, reduce traffic histograms dimension and on the other hand keep “most” of the traffic anomalies. However, since the number of senators increases proportionally to the value of K , it is crucial to choose an “small” value of K which minimizes the total number of senators while achieving a low approximation error. Unfortunately, choosing a “small” K value is notably challenging. First because decreasing the value of K increases the probability of missing many encountered anomalies and second because the compressibility of the traffic histograms varies over time and over data set [30]. For example the top-91 destination ports achieves an approximation error in order of 15% in trace D , while only the top-5 source ports are required to achieve the same approximation error in trace C [30].

3.2.2 Traffic filtering reduces the K value

Given the ability to approximate the traffic per source port and destination port histograms from the first few K coefficient, a main question prompts: does a low value K of “top” components carry the maximum amount of anomalies? In this section we tackle this question.

Figure 3.4 shows the rank of a destination port subject to a network scan (anomalous destination port) in both the histogram of original traffic and filtered traffic (small size flow traffic having number of packets per flow ≤ 3), per destination port values. Clearly, the rank of the anomalous port decreases and climbs toward the first “top”- K components when the traffic is filtered.

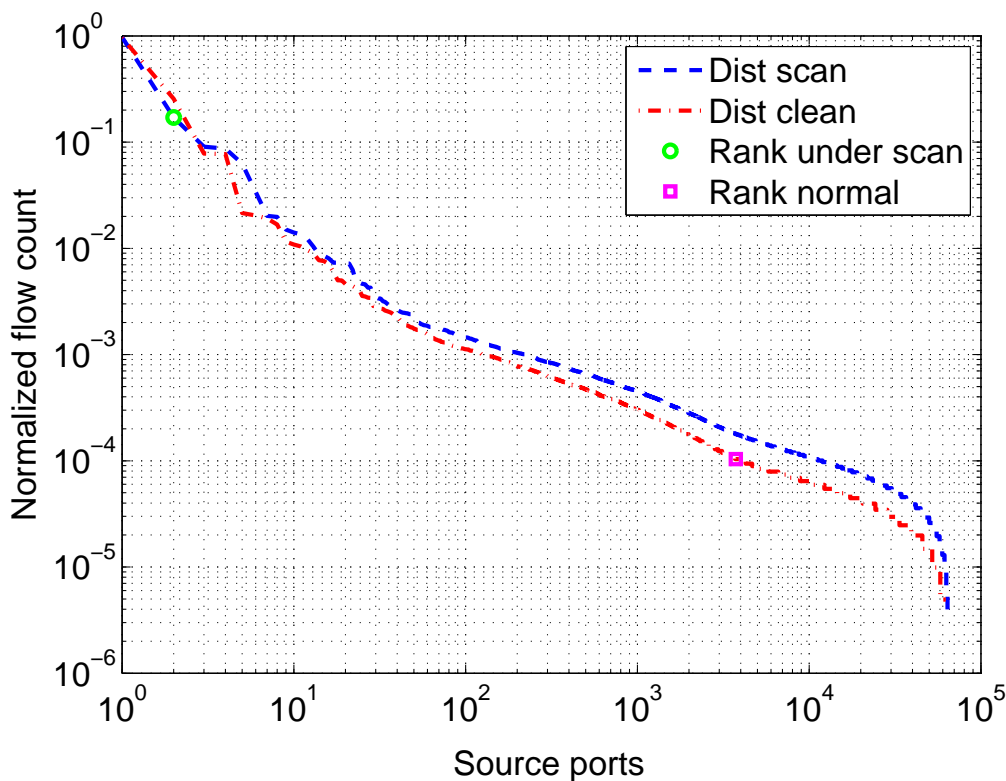
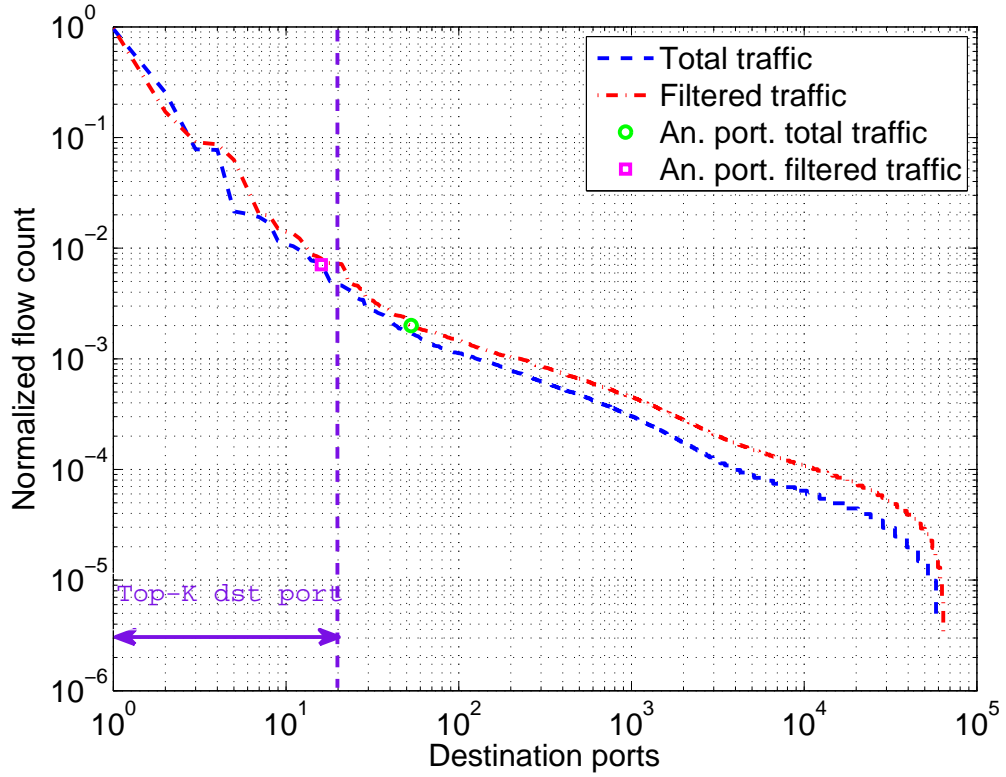


Figure 3.3: Motivation behind the "top"- K approach

We refer in our work to *population* as the filtered traffic which the top- K components are suspicious to carry the maximum amount of target anomalies. We establish the traffic filtering rules based on a set of heuristics. These heuristics are revealed following the description of target anomalies as illustrated in Table 3.1. Ultimately, table 3.1 encompasses the idea that the target anomalies such as DoS/DDoS or scans might be mostly carried by the "top"- K feature values carrying *small size flows*. Even though, some DoS attacks implicating large flow size will be missed using our approach, focusing on small flow size will reduce the risk of false positives. This is mainly because large flow size transfer frequently involves benign activities such as bandwidth tests, large transfers, high-volume P2P activity and data streaming [41].

We simply define *small size flows* as flows which have *small packets* or *low byte count*. We summarize the two heuristics we used to construct the "population" in Table 3.2, while a more detailed investigation of the threshold values α and β is presented in Chapter 5.

Figure 3.4: Impact of filtering on the K value

Anomaly	Description and Impact on traffic features
DoS	Small or large sized flows sent from one source AS via one or multiple source ports to one destination AS on one or multiple destination ports
DDoS	Many small sized flows sent from one or many source AS via one or multiple source ports to one destination AS on one or multiple destination ports
Network scan	Many small sized flows sent from one source AS via one source port to one or many destination AS on one destination ports
Port scan	Many small sized flows sent from one source AS via one source port to one or many destination AS on multiple destination ports

Table 3.1: Anomalies definition

3.3 Voting

Anomalies tend to distort the underlying traffic per feature values distributions [2]. Some feature values in the tail of the distribution tend to be among the top- K , as shown in Figure 3.4, while others tend to suddenly appear during anomalous time bins. Ultimately, our targeted anomalies, e.g. DoS/DDoS and scans, generally induce positive abrupt variations in the traffic per feature value time series.

To detect such variations we introduce the *low-rank and sparse matrix decomposition* technique PCP and show how it can be involved in the senators vote process.

Heuristic	Definition
H1	Small packet count per flow: packets $\leq \alpha$
H2	Small byte count per flow: bytes $\leq \beta$

Table 3.2: Heuristics

3.3.1 Senators Subspace Creation

We introduce senators subspace in order to address the above-mentioned problem. Abrupt variations in flow count time series are potential indicators of anomalies in the network [22]. However, since we are additionally focusing on pinpointing the root-cause of the detected anomalies, we further track the amount of traffic per feature values over time and pinpoint variations as potential indicators of anomalies. Due to the high dimensionality of such a structure, we exclusively focus on the set of elected senators. The voting process is thus summarized in the change detection of the time series of flow count per senator feature value over time. For this end we define the senators subspace X .

Let X denote a three dimension data matrix. X is composed of flow count time series per senator feature value over time and across the 4-tuple flows: $X(t, i, j)$ denotes the flow count at time t ($t \in [1, N]$) for senator feature i ($i \in [1, K]$) and of the flow-aggregation level j ($j \in [1, 4]$). Along traffic features dimension four matrices are constructed: $X(srcPort)$, $X(dstPort)$, $X(srcAs)$ and $X(dstAs)$. They contain the flow count for each senator feature over time. Figure 3.5 illustrates a graphic representation of senators subspace.

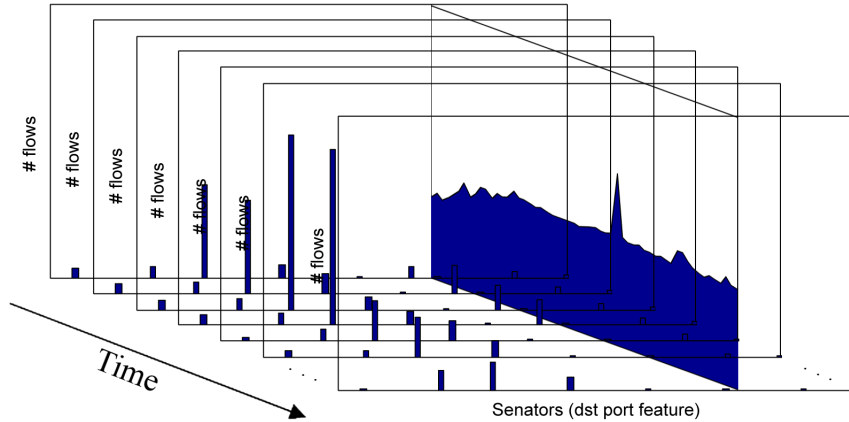


Figure 3.5: Illustration of the distribution of traffic for the senators for the destination port feature distribution over a 24 hour period, and the time series for the senator port 80

3.3.2 Significant Change Detection

Our goal in this section is to identify significant changes induced by network anomalies in the flow count per senator feature time series within the senators subspace.

There are a wide range of time series anomaly detection methods in the literature, ranging from the time series forecasting techniques, i.e, EWMA, ARIMA, to the frequency and wavelet domain analysis. Structural analysis such as Principal Component Analysis (PCA) [42, 4] and its extensions e.g, Kalman-Loeve expansion [43] was shown to be more accurate [19, 32] while better suited for a parallel decomposition of the set of time series under analysis [24], which is required due to the scale of our application. Briefly, PCA constructs the baseline behavior by projecting the time series under analysis into a normal subspace while flagging the deviation from the baseline into anomalous behavior. Unfortunately, it was recently shown that PCA in addition to its extension are vulnerable to the *low-rank subspace poisoning phenomenon* due to the outliers induced by some high intensity anomalies [42, 44]. To address this shortcoming, we propose a more robust technique called Principal Component Pursuit (PCP) that can detect changes even when the time series records large outliers. PCP is based for flow-count time series analysis, on two main assumptions which we explore in the next sections.

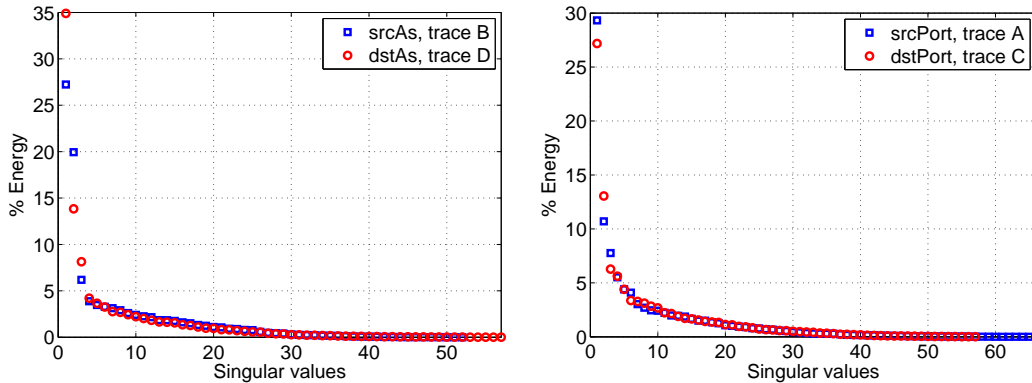


Figure 3.6: Scree plots (src/dst AS/ports for the four collected traces)

Low-rank senators subspace: In this section we observe the structure of the senators subspace. Particularly we show that the senators subspace is *low-rank*.

A matrix X that is low-rank or having an *effective rank* r ($r \ll m$) is a matrix that is well approximated with a rank r matrix, X_{appr} . A well known technique to find X_{appr} , is to apply the singular value decomposition SVD on the matrix X . SVD decomposes the matrix X into three matrices such that $X = U\Sigma V^t$, where V and U represent respectively the right and left singular vectors, while Σ represents the singular values of X , then extract the first r singular values from the diagonal of Σ . In this case, the remaining singular values beyond the first r are all small, which induces a suitably small approximation error.

Figure 3.6 illustrates the scree plots applying SVD on the matrices of the senators subspace ($X(srcPort)$, $X(dstPort)$, $X(srcAs)$ and $X(dstAs)$) for the four collected traces. The figure shows that most of the energy (variability) in the traffic time series within the senators subspace is contributed by the first singular values while the contribution to the total energy experiences an early sharp knee as the number of singular values increases. We thus conclude that senators subspace matrices $X(t, i)$ have low intrinsic dimensionality relative to the original dimensionality, i.e, low-rank.

Sparse anomalies: The low rank structure of senators subspace is seldom met due to traffic anomalies. Traffic anomalies generally induce abrupt variations in the amount of traffic for the feature which carries that anomalies. We observe that these anomalies can be either spikes or a set of pulses. As they occupy a short duration within the total observation window (a day of

measurement), we assume that they are temporary localized, thus **sparse** in time.

Principal Component Pursuit: Assuming that normal senators subspace N is low-rank while anomalies A are sparse in time, we attempt to find the matrix A such that the matrix $N=X(t,i)-A$ has the lowest possible rank. More formally we tempt to solve the following problem:

$$\min_{N,A} \|A\|_0, \text{ subject to } X(t, i) = N + A, \text{ rank}(N) \leq k \quad (3.3)$$

where $rank()$ denotes the rank of a matrix, $\| \cdot \|_0$, denotes the ℓ_0 -norm : the cardinality of the non-zero elements.

This optimization problem is NP-hard [45]. However, based on the recent advances in convex optimization theory, it has been proven that the nuclear norm, i.e, the sum of singular values, exactly recovers the low rank component [45] while the ℓ_1 norm, i.e, the sum of absolute values, exactly recovers the sparse component *with a remarkable robustness to the outliers in comparison to the ℓ_2 norm* [46]. Accordingly, Equation 3.3 (low-rank and sparse recovery) can be solved using the *Principal Component Pursuit* [45] defined as:

$$\min_{N,A} \|N\|_* + \lambda \|A\|_1, \text{ subject to } X(t, i) = N + A, \quad (3.4)$$

where $X(t, i)$ denotes the matrix of flow count per senator time series for a particular traffic feature $X(t, i) \in \mathbb{R}^{N \times K}$, $\| \cdot \|_*$ denotes the nuclear norm, i.e., the sum of the singular values of the normal delay matrix N , $\|N\|_* = \sum_{i=1}^{\min(N,K)} \eta_i = \text{trace}(\sqrt{N^T N})$, $\| \cdot \|_1$ denotes the ℓ_1 -norm of the anomalous events matrix A ie $\|A\|_1 = \sum_{i=1}^{N \times K} |A_i|$ and $\lambda > 0$ is a weighting parameter.

To solve such a convex problem, different solvers have been proposed, ranging from the Alternate Direction Method (ADM) [47] to the Singular Value Thresholding (SVT) and the Dual Method [48]. We opt for the one which scale for large matrices using the inexact version of the Augmented Lagrange Multiplier (IALM) solver [49].

3.4 Decision

The decision is the last step in our proposed technique. This is fundamental for two main reasons: it detects anomalous time bins and it identifies the subset of anomalous flows among the set of candidate anomalous flows identified by the senators' votes.

3.4.1 Anomalous Time Bins Detection

To detect anomalous time bins we proceed with the analysis of the senators' votes. More formally, we collect in each time bin the set of senators votes which potentially trigger suspicious events and logically combine (AND, OR) the collected votes based on a set of predefined decision rules. Various decision rules might be chosen predicated on the collected senator's vote to flag a time bin as anomalous, we propose two which are summarized in Table 3.3. Using R1, an anomaly is flagged when both source AS, destination AS and at least a source or a destination port are flagged. While this may detect most of the anomalies including DoS/DDoS targeting or originating random ports, it generates a high false-positive rate due to the voting algorithm artifact or due to a normal user behavior inducing a sudden increase in the amount of flows per feature value. To make the decision process more robust against this, an anomalous time bin is only flagged when a senator vote is recorded for the 4-tuple aggregation levels. We adopted in this thesis the decision rule R2.

Rule	Definition
R1	$(srcAS \wedge dstAS) \wedge (srcPort \vee dstPort)$
R2	$(srcAS \wedge dstAS \wedge srcPort \wedge dstPort)$

Table 3.3: Decision rules

3.4.2 Suspicious Flows Identification

Once detected, anomalous time bins will serve to identify the set of anomalous flows responsible for the attack in the network. From each of the anomalous bins, we select all flows that match the union of source and destination addresses and services flagged by a vote in that particular time bin, i.e. all possible flows that might be originated from any suspicious source at any suspicious port and targeting any suspicious destination at any suspicious port, that are flagged by senators' votes.

Figure 3.7 illustrates anomalous time bin detection and flows identification based on senators' votes. It shows the amount of flow time series per srcAs, dstAs, srcPort and dstPort senators. These time series were extracted, organized within the senators subspace and processed during the voting. As previously discussed, senators votes simply consists on the detection of abrupt variation in senators time series using PCP. The figure shows that PCP detects variations at more than one senator time series in each of the 4-tuple aggregation levels at time bin 60. Based on our heuristic, this time

bin is confirmed to be anomalous. Unlike time bin 60, the previous time bin is only flagged as anomalous by a srcPort senator (number 58643). In this case the vote is considered erroneous and the time bin is ignored. Once anomalous time bins exposed, the flows responsible for the anomaly are simply identified as the set of flows matching the senators' votes at that particular time bin. The figure illustrates two main flows responsible for the anomaly detected as time bin 60 (flow 1: {1299,2107,2000,6667} and flow2: {1299,2107,58643,6668}).

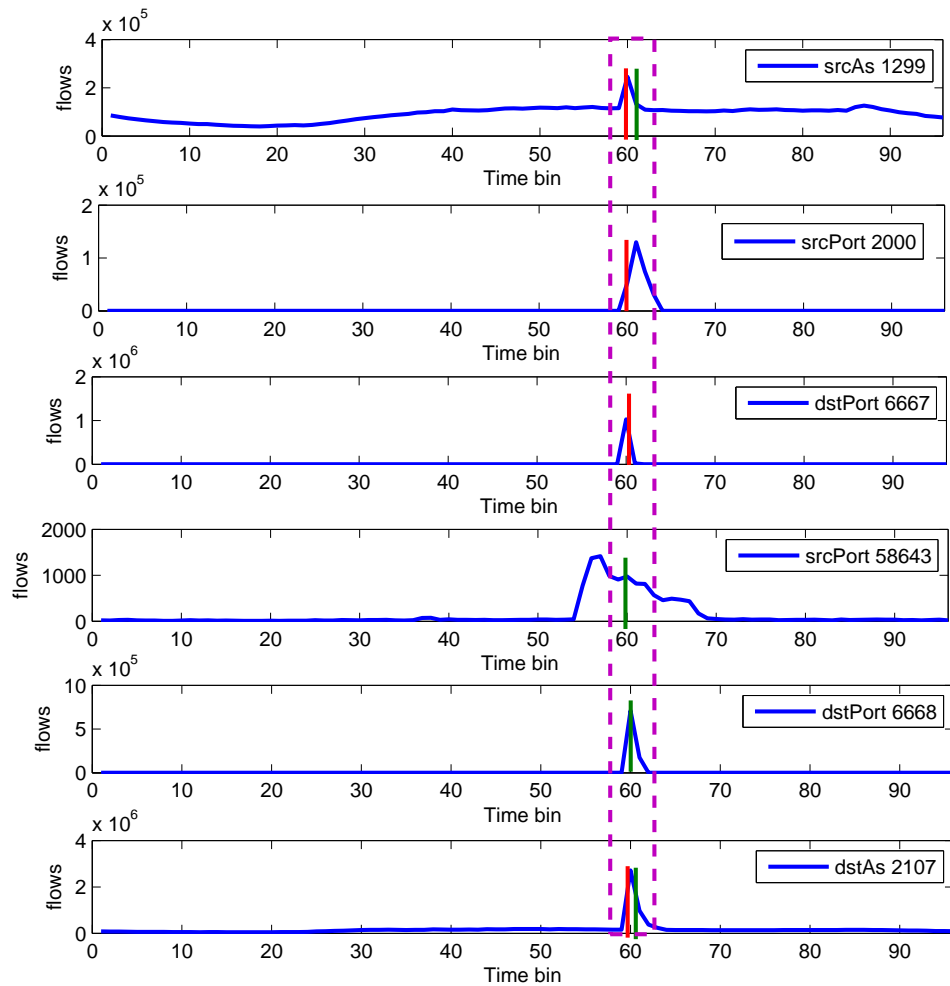


Figure 3.7: Illustration of the Decision process

3.4.3 Root-Cause Analysis

After identifying the anomalous flows, we have to infer the event that caused it. The flagged anomalous flows from Senatus gives us 4 feature values which makes the anomalies classification and root cause analysis far simpler than analyzing the entire NetFlow file, since filtering on the feature values will narrow down the number of candidate flows dramatically. Even though Senatus makes this task simpler, manual analysis is still time-consuming. In the next section we discuss a proposed automatic root-cause analysis algorithm.

3.5 Automatic Root-Cause Analysis

During the manual analysis we developed a script that made the analysis more automatic. The script looped through the flagged anomalous flows from Senatus and read NetFlow files with specified filters applied based on the feature values. We could control which filters that would be applied to be able to look for different types of attacks. We still had to find the attack manually, but it saved us a lot of time and we were able to analyze the results faster. We saw that the script could be developed further and it became the first draft for our proposed automatic root-cause analysis algorithm.

One of the challenges with an automatic analysis of the output from Senatus is that the flagged anomalous flows from Sentus are "samples" of the set of flows carried by anomalies, which can be much bigger and might involve hundreds to thousands of flows. In example, a flagged anomalous flow which involves the source AS A, the destination AS B, the source port C and the destination port D may be a sample of a network scan of the destination port D to multiple IP addresses within the destination AS B. If the attacker use multiple source ports, we will miss these flows, since they are filtered out. To overcome this challenge, we propose the following algorithm. We relax the aggregation level reducing the number of tuples within each of the anomalous flows to the minimum, i.e. src AS or dst AS, in the flagged anomalous flows and estimate the number of anomalous flows resulting from the relaxation procedure. We compute the estimate number of anomalous flows as the maximum number of flows within the anomalous time bin involving flow tuples under investigation.

The choice of the flow aggregation level and number of tuples is our algorithm is motivated by traffic anomalies definition and their fingerprints provided in Table 3.4 (based on anomaly definitions from Chapter 2).

Anomaly	Description	Features
DDoS	Attack from multiple sources to one destination	dstIP
DoS	Attack from a single source to a single destination	srcIP, dstIP
Network scan	Attack from a single source to multiple destinations, but on the same port number	srcIP, dstPort

Table 3.4: Anomaly characteristics

3.5.1 The Structure of the Algorithm

In Algorithm 1 we see the pseudo code of the algorithm which is mainly divided into three parts or statements that each detect one type of attack; DDoS, DoS and network scan. The algorithm is using if/else statements to prioritize the attacks such that if we find a DDoS, we will return that attack and not continue looking for other attacks in that specific alarm. Instead we start analyzing the next alarm, and this is done to decrease runtime and increase performance.

Part 1 - DDoS Classification DDoS is an attack from multiple sources to one destination, so to detect this attack we focus on unusual behavior at the destination. To detect DDoS attacks we count the number of occurrences of each destination IP and find the maximum, which will be the IP with most incoming flows. We then check the value against a predefined threshold θ_1 and if it is above this value we flag the suspicious flows as a DDoS attack.

Part 2 - DoS Classification DoS is an attack from a single source to a single destination. To find DoS attacks we pair up the source-IP with the destination-IP from each flow and find the maximum, which will be the IP-pair that generates most flows. We then check the value against a predefined threshold θ_2 and if it is above this value we flag the suspicious flows as a DoS attack.

Part 3 - Network Scan Classification Network scan is an attack from a single source to multiple destinations, but on the same port number. To find network scan attacks we filter out all the traffic to the flagged destination port and then count the source-IP from each flow and find the maximum, which will be the source IP that sends the most traffic to the

Algorithm 1 Root-cause($\mathcal{F}, \theta_1, \theta_2, \theta_3$)

Input: Suspicious flows $\mathcal{F} = \{f_1, \dots, f_i, \dots, f_n\}$

 threshold $\theta_1, \theta_2, \theta_3$
 \mathcal{A}_x set of flows of feature value x
Output: The root cause

```

1: for  $f_i \in \mathcal{F}$  do
2:    $x \leftarrow dstIP$ 
3:   if  $(max(\mathcal{A}_x) | x \in f_i(dstAs) > \theta_1)$  then
4:     return DDOS,  $dstIP_{max}$ 
5:   else
6:      $x \leftarrow \{srcIP, dstIP\}$ 
7:     if  $(max(\mathcal{A}_x) | x \in f_i(\{srcAs, dstAs\})) > \theta_2$  then
8:       //  $x$  is the pair of  $srcIP$  and  $dstIP$  within the  $srcAs$  and  $dstAs$ 
       // flagged in the anomalous flow  $f_i$ 
9:       return DOS,  $\{srcIP, dstIP\}_{max}$ 
10:    end if
11:   else
12:      $x \leftarrow \{srcIP, dstPort\}$ 
13:     if  $(max(\mathcal{A}_x) | x \in f_i(\{srcAs, dstPort\})) > \theta_3$  then
14:       return Network Scan,  $\{srcIP, dstPort\}_{max}$ 
15:     end if
16:   else
17:     False Positive
18:   end if
19: end for

```

flagged destination port. We then check the value against a predefined threshold θ_3 and if it is above this value we flag the suspicious flows as a network scan attack.

3.5.2 Setting the Right Threshold

We highlight that the threshold θ is a key parameter for the automatic root cause analysis. However, the choice of this threshold is notoriously difficult. The discussion about our approach to set the best threshold θ can be found in Chapter 5.

Chapter 4

Implementation

This chapter gives a thorough description of the system developed during this thesis. Figure 4.1 shows the intended architecture of the system as a whole.

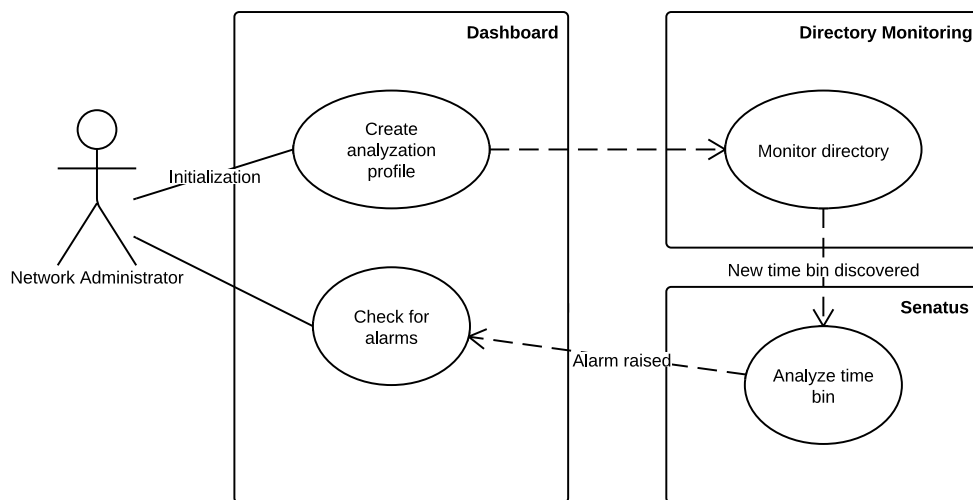


Figure 4.1: A use-case diagram of the intended Senatus system.

The different, separate parts which needs to be implemented are Senatus, the Senatus Dashboard and the automatic root-cause analysis algorithm:

- An implementation of Senatus as back-end in the system.
- Senatus Dashboard, the user interface to the system used for event handling, monitoring and configuration of the back-end.

- Script for automatic root-cause analysis.

System Call The term "system call" in this report is referred to when a program or web page issues commands to the underlying system, e.g. Linux, for utilizing tools that are only available on the system, and not to the source code itself. The "system call" will normally return the result of the system call as a string to the program or web page. This will be equivalent to running a command in the terminal and returning the output to our program.

4.1 Senatus

4.1.1 Introduction

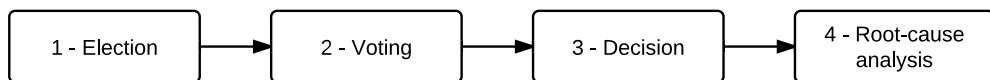


Figure 4.2: High-level overview of the Senatus design

Senatus is split into four parts; election, voting, decision and root-cause analysis, as illustrated in Figure 4.2. A quick summary can be given as this:

1. Election: A set of senators are selected for each of the following features: source AS, source port, destination AS and destination port. A matrix for each feature is constructed with the number of occurrences of these senators for the last 24 hours.
2. Voting: A PCP-algorithm, described in Section 4.1.2, performs matrix operations on the four matrices to find combinations of senators and time bins that shows abnormal traffic patterns based on previous values for the given senator. A list of suspicious flows is constructed by combining all possible tuples of the four features.
3. Decision: The list of suspicious flows is iterated over. Each flow is filtered in its specific time bin. If the number of resulting lines from the filtering is greater than zero, this suspicious flow is marked as an alarm.

4. Root-cause analysis: The suspicious flows marked as alarms will be automatically checked for its root-cause.

The back-end of the Senatus system is the heart of the implementation part of this thesis. The objective is to develop a stand-alone program in either Java or C++ that will run continuously and analyze time bins in real-time, or online.

An existing implementation of Senatus is already developed, we refer to it as version 0. Version 0, described in Section 4.1.2, consists of several bash- and MatLab-scripts. The three parts: election, voting and decision are not linked together and each part needs to be run separately, so we need to address the best and most efficient solution to link these parts together.

To develop the program, we will learn from the existing implementation, and as we progress, improve on our own version. The development cycle was split into the subsequent sub-tasks:

1. Study Senatus version 0 and understand it in detail.
2. Understand which features that are needed for the program, and based on these features decide on the programming language and development environment.
3. Do a direct one-to-one translation of version 0, we call it version 1. This is to verify that the same input gives the same output in both versions.
4. Improve version 1 to become a fast-performance, real-time, online analysis application, version 2.

The automatic root-cause analysis algorithm was not introduced before the development of version 2, and is thus only present in that implementation. Also, automatic root-cause analysis is an optional part of version 2.

Online and offline In this context, an offline implementation means that the system back-end analyzes the data too slow to present it to the user in real-time. This means that the back-end cannot be used with the Dashboard (the monitoring user interface). An online implementation, on the other hand, delivers the data real-time or near real time, and meets these requirements.

4.1.2 Version 0 (Offline Implementation)

Version 0 is the first, proof-of-concept, implementation of Senatus, written by A. Abdelkefi with the purpose of testing Senatus and perform parameter tuning.

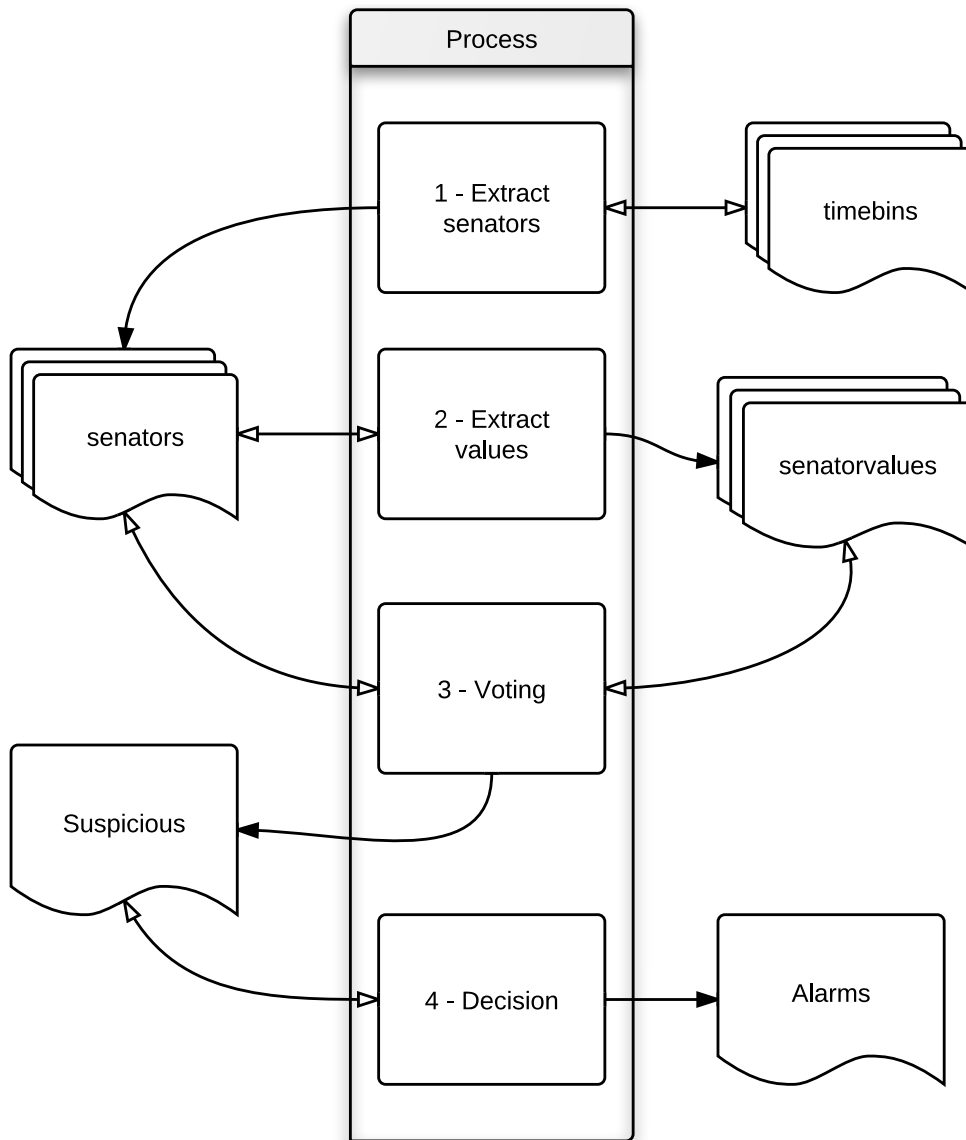


Figure 4.3: Flow diagram of version 0.

Figure 4.3 shows how election, voting and decision is implemented in version

0. It is worth mentioning that the election consists of both step 1 and 2 and that all passing of data between the different steps are done by saving and opening text files from the hard drive.

The following list gives a step-by-step description of Figure 4.3:

1. *Extract senators* reads time bins and creates one file for each feature consisting of all their senators.
2. *Extract values* reads the files saved in the previous step, runs flow-tools-filters for all these senators for the entire timespan=24 hours, and saves the values per feature and senator as matrices in files formatted to be readable by Matlab. An example of such a matrix is seen in Figure 4.4, where we have two hour time bins, and 7 senators.
3. *Voting* reads senator values from the previous step, and run these through the PCP algorithm to get time bin-senator pairs that are abnormal. All combinations of pairs are combined to get full srcas, sreport, dstas, dstport-combinations called suspicious flows and saved as a matrix text file.
4. *Decision* loads the suspicious flows-matrix from the previous step and run them through flow-tools or netflow. If the number of flows returned from the filter is bigger than zero, it is marked as an alarm. All alarms are saved to a file.

	Senator 1	Senator 2	Senator 3	Senator 4	Senator 5	Senator 6	Senator 7
0:00	806	3447	221	21	115927	1819	39414
2:00	829	3351	222	34	138396	1855	43592
4:00	1042	3423	212	45	140780	1995	47198
6:00	983	3654	237	26	146784	1972	49053
8:00	947	3773	231	28	157691	1900	51382
10:00	1053	3750	211	47	159139	3871	52975
12:00	1049	3823	224	33	168590	7099	57234
14:00	1093	4029	233	54	174226	6586	57652
16:00	1088	3988	233	40	180195	7053	57988
18:00	1125	3949	245	35	184299	6842	58471
20:00	1159	4058	210	46	188086	6757	62633
22:00	1168	5298	207	29	193152	7289	64552

Figure 4.4: Value matrix for one feature.

PCP PCP is the method for matrix decomposition used in Senatus. The method used is called Inexact ALM RPCA, developed by Lin et al., described in Section 3.3.2.

4.1.2.1 Limitations of Version 0

Version 0 has some limitations and weaknesses.

- The entire analysis takes approximately 24 hours.
- Configuration and paths needs to be hard coded in the source code.
- Each part must be run separately and started manually after the previous part finishes.
- It operates on chunks of 24 hours network traffic data only. This implies a significant delay between the time of an attack and time of detection.

4.1.3 C++ Implementation Technology

This section focuses on the choice of technology for the high-performance implementation.

4.1.3.1 Choice of Programming Language and Matrix Library

The two main criteria for the choice of programming language were:

- Expected execution time performance.
- The availability of a well-performing linear mathematics library.

Which programming language to implement Senatus in was a choice between Java and C++. Both are well-known, supported and solid programming languages, and we believed that both could be used with a satisfactory performance.

Availability of a linear mathematics library is important, as we needed to replicate the Matlab parts of version 0. The performance of this library is also important, as research showed this could vary[50].

Linear mathematics library Both Java and C++ have libraries for matrix manipulation. Java has JBLAS[51], Jama[52], Colt[53] and more.

C++ has some old libraries: LAPACK++[54], Atlas[55] and CBlas[56], which in our opinion had a difficult syntax. It also has got some new libraries: Armadillo [57, 58], OpenCV[59] and Eigen[60].

The newest libraries seemed to have a simpler interface for us to get used to. Armadillo uses bindings to Lapack for advanced matrix operations to profit on Lapack's performance.

Armadillo because it had a close resemblance to Matlab and seemed to be the fastest[50].

Execution Time Performance Research indicates that C++ has superior performance compared to Java[61]. There are certain memory management issues with C++, issues not found in Java because it has built-in memory management[62]. We believe however that the memory management issue is possible to work around, and that it is not grave enough to disregard the gain in performance from using C++.

Based on the aforementioned requirements, we conclude that C++ is the best suited programming language, with Armadillo as the Linear Algebra library.

4.1.3.2 Dependencies and Tools

After the decision of using C++ and Armadillo was taken, the architecture of version 1 and 2 could be decided, as well as the dependencies inferred by the design.

The dependencies and tools are:

- Version 1 and version 2:
 - nfdump
 - flow-tools
 - AWK
 - head
 - Armadillo
- Only version 2:

- MySQLC++Conn
- SQLite3

Senatus was developed in a Linux-environment, and uses functions specific to POSIX-based operating systems.

nfdump and flow-tools nfdump[63] and flow-tools[64] are software packages for collecting and processing NetFlow-data from routers. NetFlow is a network protocol from Cisco to record network performance data[65].

Senatus is able to process network traffic flow data in both these formats by running C++ system calls to nfdump and flow-tools on the machine. Thus, either flow-tools, NetFlow or both needs to be installed on the system running Senatus, depending on the files that are to be analyzed.

AWK AWK[66] is a powerful text-processing language that operates on each line of an input. We use it in the system calls to extract the necessary information from the output of nfdump and flow-tools, and return it in a uniform way to Senatus.

head head[67] is a tool for displaying only the top n lines of its input. We use head in the system calls to limit the number of records to return to Senatus, for example when getting the top k senators.

Armadillo Armadillo is a linear algebra library for C++ which is open source, and according to the developers "aiming towards a good balance between speed and ease of use" [58].

The version of Armadillo used in Senatus is version 2.4.2, also called "Loco Lounge Lizard." This was the latest stable version when implementation of Senatus begun.

Armadillo depends on LAPACK, Blas and Atlas for matrix decompositions and Boost[68] for standard tasks.

Armadillo is licensed under the LGPL-license[69], making it possible to publish a non-derivative work like Senatus under a license different from LGPL.

Database libraries C++ has no built-in database capabilities, so libraries are needed for both SQLite and MySQL.

SQLite was used in the first part of development, because of its agility and simplicity. However, as Senatus became more complex, we needed concurrent writing[70]) and closer integration with the Dashboard. We therefore changed to MySQL, which fully replaced SQLite in Senatus version 2.

The library used for SQLite was libsqlite3-dev[71], while the one used for MySQL was libmysqlcppconn-dev[72] from Oracle.

4.1.3.3 Development Environment

IDE As IDE we used Eclipse CDT[73] configured to use g++ for compiling. Eclipse is an IDE configurable for a variety of programming languages[74], and CDT is the plugin for programming in C++.

Version Control System Version control is a useful tool for cooperative system development, and for keeping different revisions of a project. The use of version control systems in development projects will give complete control over what changes that have been done, and by whom. Mercurial[75] was used as a version control system for the implementation of Senatus, and Bitbucket[76] for hosting of the source code.

4.1.4 Version 1 (Offline Implementation)

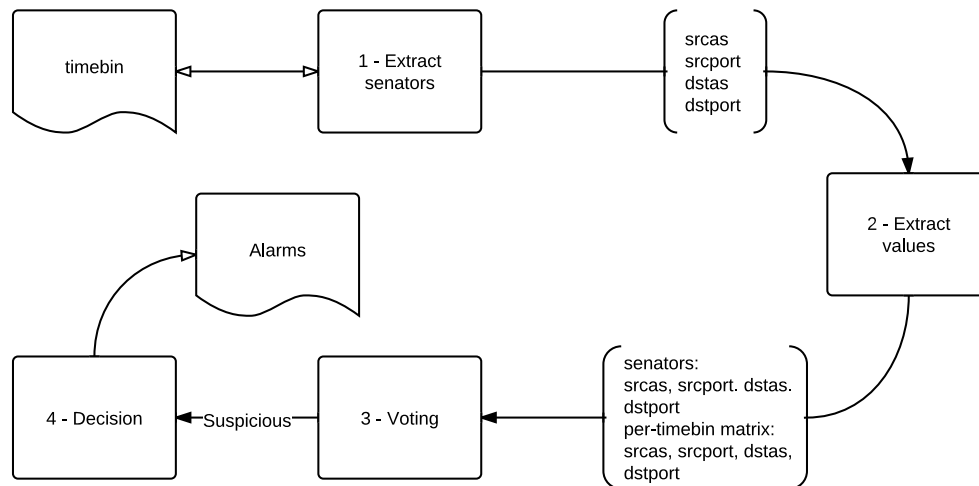


Figure 4.5: Flow diagram of version 1.

Senatus is a complex system, thus we need to verify that the output from our implementation of Senatus is the same as from version 0, given the

same input. Therefore, before making our enhancements to the program, we needed to create a replica in C++. We refer to this implementation as Senatus version 1.

The main difference in design between version 0 and 1, besides from being written in C++, is that version 1 operates as a whole, and that all files that version 0 saves now are stored as matrices in-memory, illustrated in Figure 4.5.

Figure 4.5 is described below. This description focuses on the differences between version 0 and 1.

1. Senators are extracted as in version 0, but are now kept in memory. The usage of memory is shown by the exclusion of the files that were present in Figure 4.3.
2. The extracted values are saved as Armadillo-matrices in memory.
3. The translated-to-Armadillo version of the PCP-algorithm is used in the voting part. The suspicious flows are still kept in memory as an Armadillo-matrix.
4. The matrix of suspicious flows is iterated over and each object is filtered. Alarms are written to a file.

The translation introduce some challenges, specifically the PCP-algorithm is time-consuming to translate because of its complexity.

4.1.5 Version 2 (Online Implementation)

Version 1 of Senatus, as its predecessor, analyzes flow data and detects anomalies for a 24-hour period. This requires system calls for every senator, for every feature, for all time bins.

Instead of detecting anomalies for a 24 hour period, Senatus should now detect anomalies only for the current time bin by comparing traffic for that time bin with traffic from the previous 24 hours. This will still require analyzing the same data in election, but voting and decision will operate differently.

This change makes it possible for Senatus to analyze each time bin without doing system calls to get the feature values for the previous n time bins. We accomplish this by saving previous senator and feature values

to a database. As a result, we only need to run flow-tools on the current time bin and on previous time bins where this senator has not been represented before. The additional values needed are fetched from the database. This significantly reduces the number of system calls required, resulting in a dramatic improvement in terms of performance.

Figure 4.6 shows the flow diagram of version 2, and the now replaced step 2, which has become more complicated because of the database look ups. Figure 4.7, explained in the list below, describes this in detail.

1. Senators are extracted as in version 1.
2. Value matrices are constructed as in Figure 4.7.
3. The first part of Voting has the same input and output, but only anomalies from the current time bin are combined to suspicious flows.
4. Only the suspicious flows in the current time bin are iterated over and filtered, and we only get alarms from these. If the automatic root-cause analysis algorithm is running, the execution is stopped when a root-cause is identified.

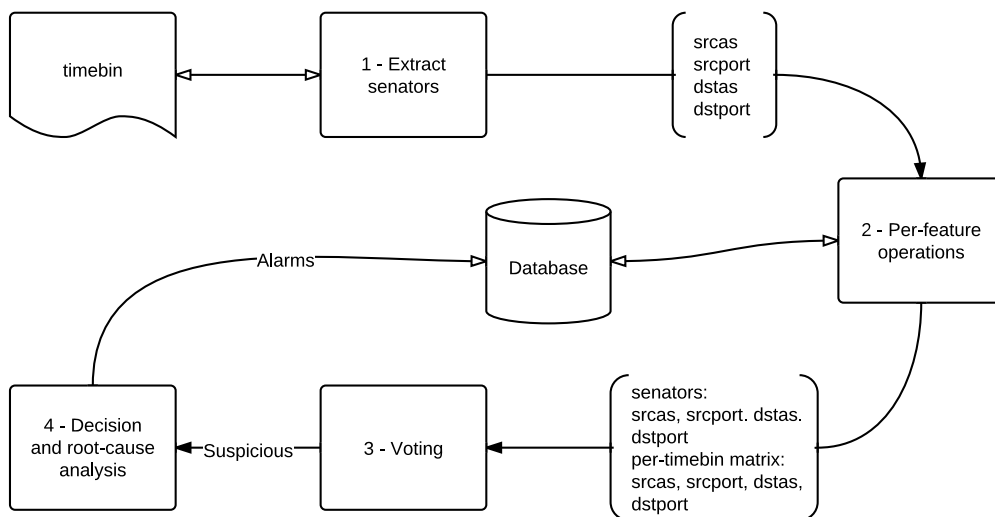


Figure 4.6: Flow diagram of version 2.

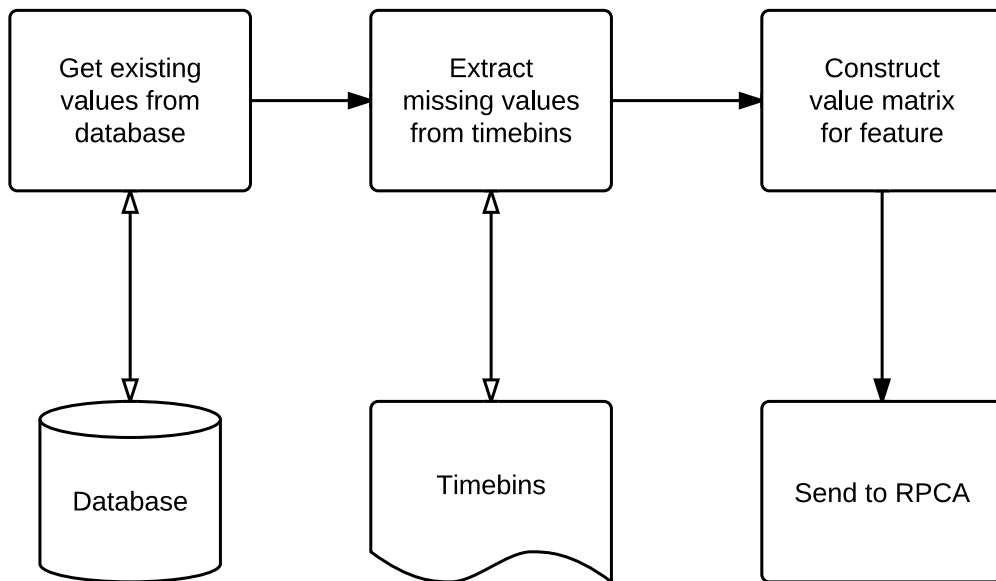


Figure 4.7: Per-feature operations, the value matrix building in version 2.

4.1.5.1 Enhancements

As described above, there are two major differences between version 1 and 2. The first change is the saving and retrieving senator values from the database, causing a big improvement on the time of execution.

The other change, in the selection, is that we now only care about anomalies in the latest time bin, thus providing up-to-date anomalies. However, we still need the information from senator values from previous time bins to detect abrupt changes.

Figure 4.8 shows how the entire value matrix for one feature is processed in the voting part. Both version 1 and 2 have the same first two steps, only differing in step 3 and step 4.

In step 3, version 1 searches for values greater than zero in the entire matrix, while version 2 only does this for the last time bin, 22:00, only choosing 1.49 and 445.77 to pass on.

Suspicious flows-objects are then generated according to the decision rules, and put in a list that is used in the decision part.

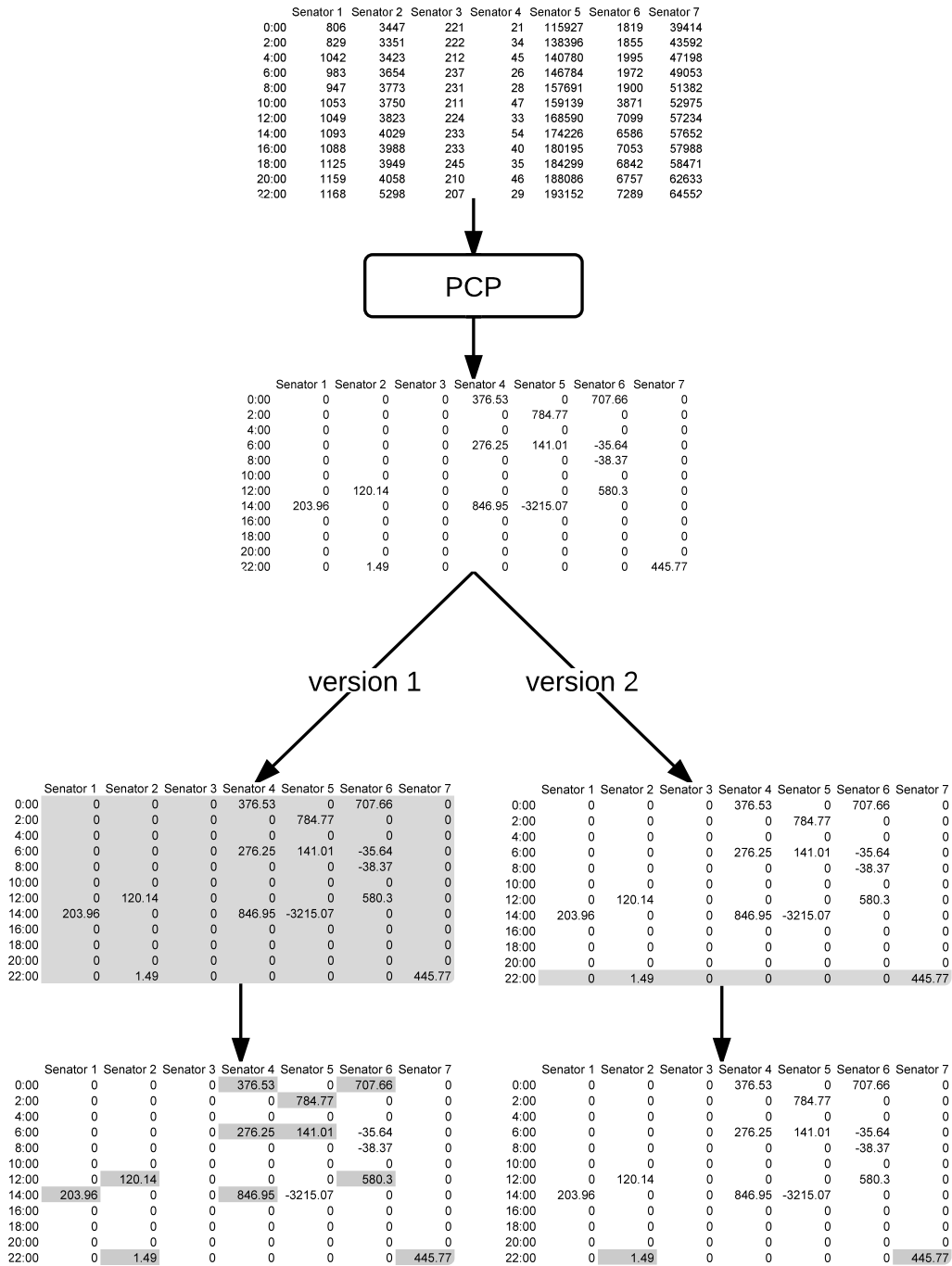


Figure 4.8: Change in selection between v1 and v2.

4.1.5.2 Parameters

Senatus has a number of parameters that are possible to tune to maximize the performance. In this case performance is measured in execution time and detection rate. These parameters are:

λ The weighting parameter for PCP.

K The number of senators for the port-features. The default is 20 senators.

t Timespan, the number of hours of data input to use in the analysis.

Hi Heuristics to be used for senator selection. These heuristics are defined in Table 3.2.

auto Auto root-cause analysis. Whether to use automatic root-cause analysis or not. When used, Senatus will stop the execution when an attack is found and identified. The default option is to stop when a network scan is found, but it is also possible to ignore network scans and continue until a DoS or DDoS is found.

Decision rules How to combine the suspicious flows. These two rules are shown in Table 3.3. R1 requires three features to have flagged values, while R2 requires all four.

The different performance characteristics of these parameters are shown in the implementation performance evaluation in Section 5.5.

4.1.5.3 Database structure

Figure 4.9 shows the structure of the Senatus database. Each database table is described below.

Time bin A time bin's path is saved for the application. The time bin belongs to a gateway, and starts at a specific date. The field *tbnr* is saved to know which time bin of the day this is. Flow, packet and byte count is saved for the Dashboard.

Senator For each time bin and feature there are senator and feature values. Type identifies what filter is used to extract the given senator; H1 or H2.

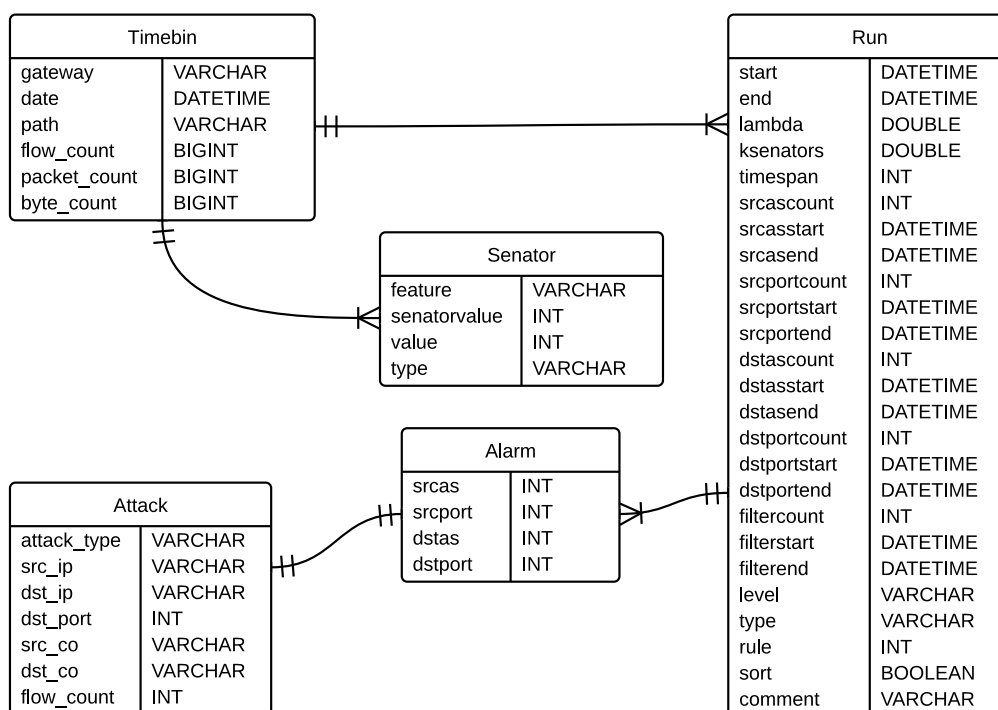


Figure 4.9: ER-model of the Senatus-relevant parts of the database.

Alarm If there is an alarm, its feature values are stored here, referencing the time bin.

Attack Alarms are checked with the automatic root-cause analysis script, or manually, and the identified attack type is stored here.

Run All executions are stored in the run table, with all required datetime-fields and variables for performance analysis.

4.1.5.4 UML Diagrams

This section will provide Class diagrams for the C++-objects defined. This, together with the sequence diagrams in Appendix A should give a thorough and detailed description of the online version of Senatus.

Procedure Diagram Figure 4.10 shows the source code files that are procedure oriented. These are used intertwined with the classes from Figure 4.11. The source code files are described below:

run This is where the main method is located, and where the initialization of each execution happens. All command-line arguments are parsed, setting the global parameters like the value of λ , k , the timespan etc. The main method controls the entire execution flow.

common This source file contains all functions that should be globally accessible, for example the `exec`-function which executes a system call and returns the result as a string.

extract Extract consists of the system call functions of the election-part. The `senators`-function gathers all senators for a feature, `valueForSenatorTimebin` gets the value for a senator in a given time bin.

voting Voting consists of the Armadillo-specific functions, where `vote` is the main function calling `filterFlows` and `Inexact ALM RPCA`.

auto This is the C++-translation of the Automatic root-cause analysis script initially developed in Python, shown in 4.3.

filter The filter source is where the decision-part takes place, where it is decided whether to use the auto script and where alarms are written to the database.

Class Diagram Figure 4.11 shows entities that are represented as classes in Senatus. Only the parts of the code where it would give an advantage were object-oriented. The classes are described below:

Feature The feature-class represents the connection between a list of senators and their corresponding values. This class is responsible for collecting existing senator values from the database, and for fetching the senator values that are not in the database from the time bin on disk with a system call.

Timebin A time bin is represented in this class, with its path, date and flow count as the primary parameters.

TimebinCollection A `TimebinCollection` constructs the list of all `Timebin`-instances that are a part of the current Senatus execution. Subsequently, existing `Timebin`-data is retrieved from the database, and missing data is retrieved by running a system call.

SuspiciousFlow Every suspicious flow constructed in the Voting-part is created as a SuspiciousFlow-object, containing important characteristics such as time bin ID, the source AS, source port, destination AS and destination port of the flow, as well as the attack-characteristics if the flow is identified as an attack. The `getWeight()`-function returns the probability of an attack in this flow based on the feature rates, so that the list of SuspiciousFlows can be sorted accordingly.

DbConn DbConn is an abstract virtual class for defining database connections for different database solutions. Subclasses of this, such as MySQLConn, are the only classes interacting with the database.

DoDbThings DoDbThings is an utility class for other classes and a way to prevent circular dependencies[77].

LogObject The LogObject-class is a class for saving info about each execution, mainly for later analysis. This includes setting the current date and time.

SenatorValue SenatorValue is a simple class, only containing a combination of time bin, senator and senator value for use in the feature-class.

4.2 Dashboard

The Dashboard is the network administrators' primary interface to Senatus, offering an intuitive and powerful way of controlling day-to-day monitoring and use of Senatus' features.

Senatus and the Dashboard are closely coupled, sharing a database, with both doing system calls, each for their own purposes.

4.2.1 Technology

The choice of technology for the web application is limited to the programming language. The database solution is already locked to MySQL because of the Senatus back-end.

In addition to programming language and database, different libraries are used to increase the user friendliness of manual root-cause analysis.

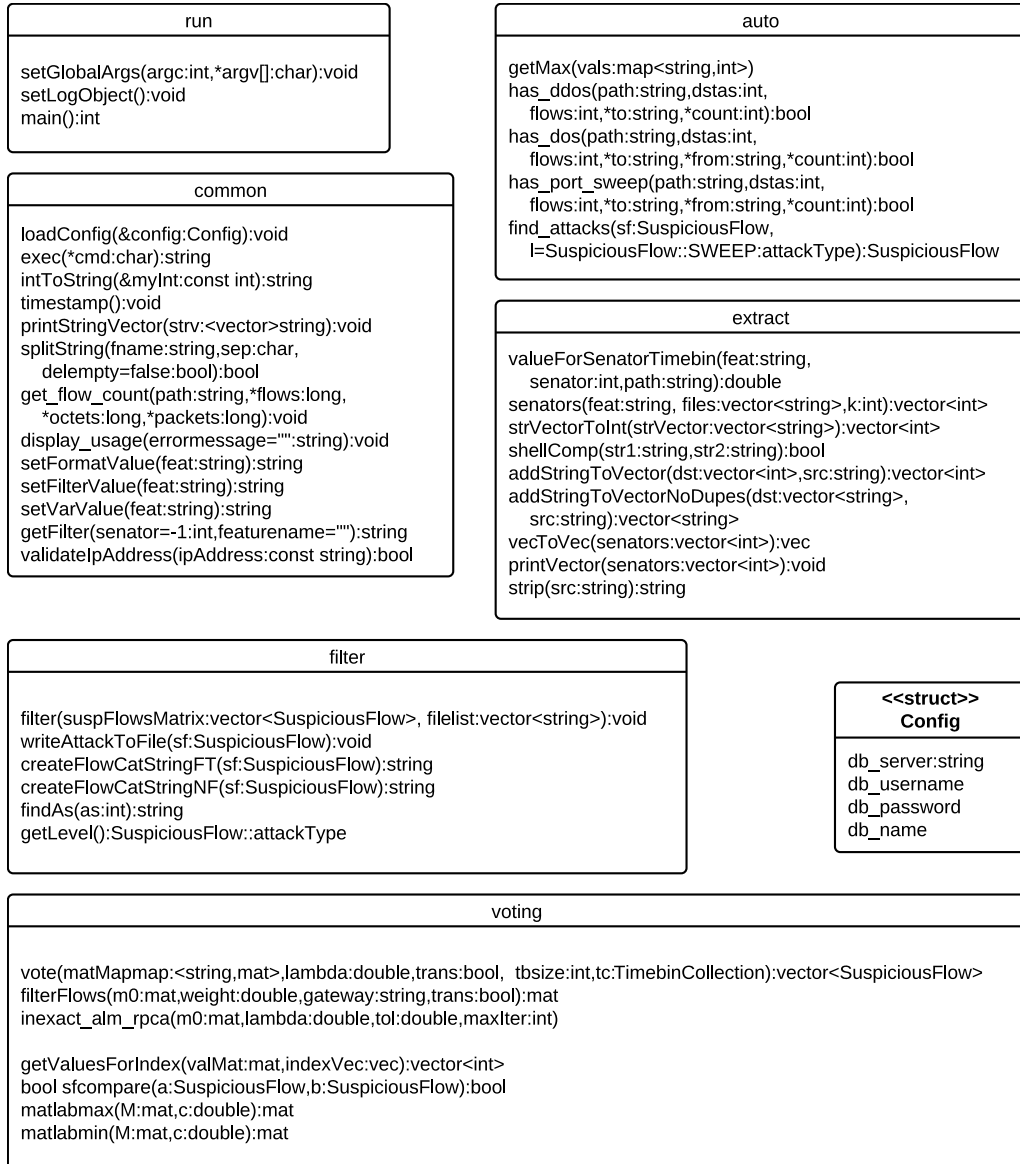


Figure 4.10: Diagram of procedure oriented part of Senatus.

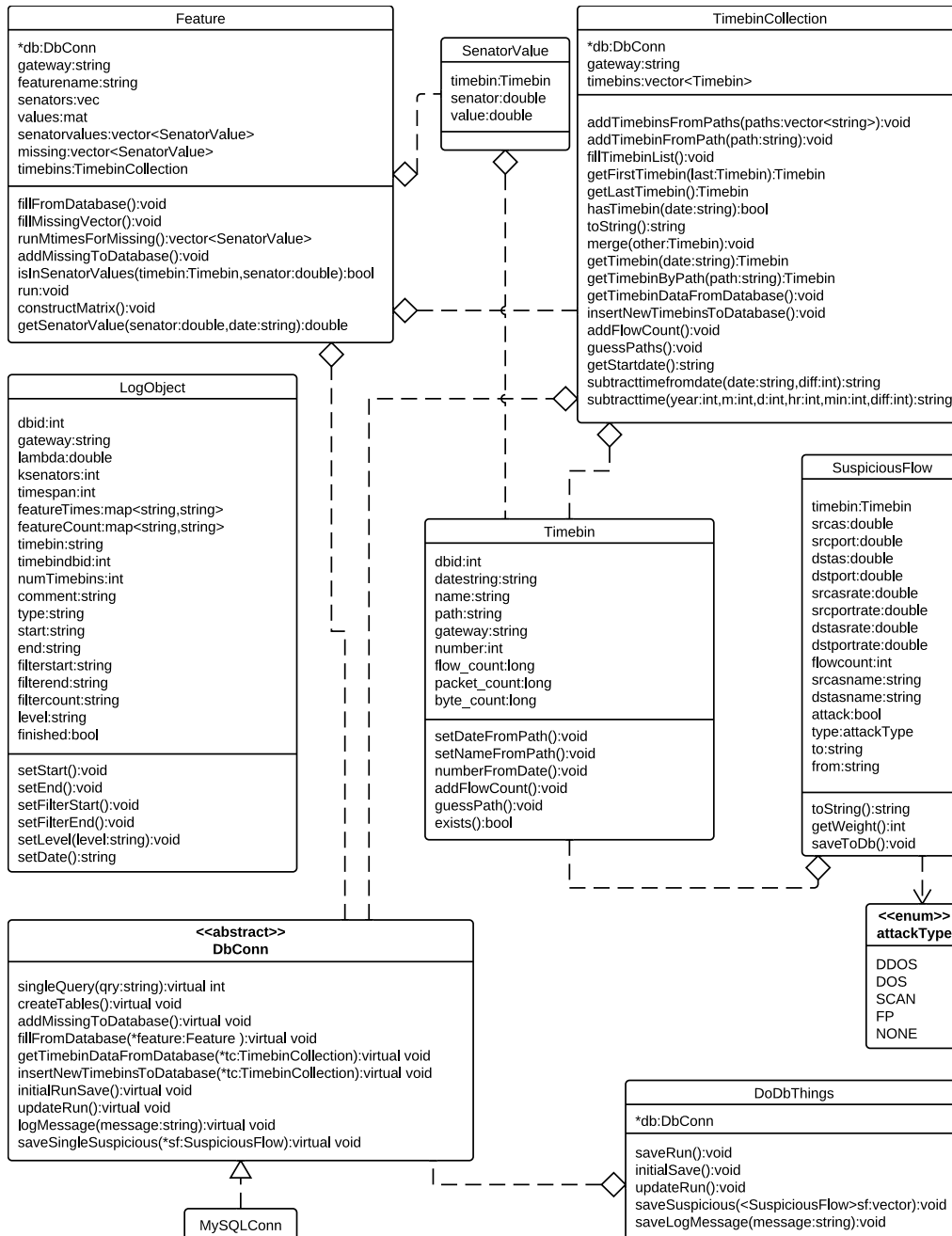


Figure 4.11: Class diagram of classes in Senatus.

4.2.1.1 Programming Language

As programming language for the Dashboard, we decided to limit the options to languages and web-frameworks that were widely used, and that at least one person in the group had experience with. We ended up with these alternatives:

- Django[78], Python-based web-framework.
- Play[79], Java-based web-framework.
- PHP[80], Plain PHP scripted solution

Of these three, Django and Play were excluded because of their Object-relational mapping (ORM)-solutions making it difficult to comply with a different database schema such as ours.

This left PHP as the choice of programming language for the Dashboard.

4.2.1.2 Development Environment

Web Server The combination of Apache and PHP for Apache was well known for the group members, making that solution the preferred one.

IDE Eclipse, being a versatile IDE, also has a plugin for PHP, called PHP Development Tools[81]. Utilizing Eclipse for both Senatus and the Dashboard is beneficial in the way that the group members will only need to accustom themselves to one IDE.

4.2.1.3 Additional Libraries

Four JavaScript[82] libraries and one library used from PHP, IPInfoDB, was used for data visualization, general UI improvements and IP address lookup.

jQuery jQuery[83] is an all-purpose JavaScript-library for easy manipulation of the HTML-Document Object Model (DOM)[84]. In the Dashboard, jQuery is used for a number of things: collapsible tables, setting text in a separate website-element when another element is clicked and for making forms prettier.

Google Maps API The Google Maps API[85] is a JavaScript-Application Programming Interface (API) for implementing Google Maps-maps in websites. This makes it easy to show the location of events, such as the source of a port- or network scan, or the target of a (D)DoS-attack.

Google Chart Tools Google Chart Tools[86] is a JavaScript-library for representing structured data in different charts. It is utilized in the Dashboard to visualize flow-, packet-, and byte-count for each time bin, sorted by date, as shown in Figure 4.12.

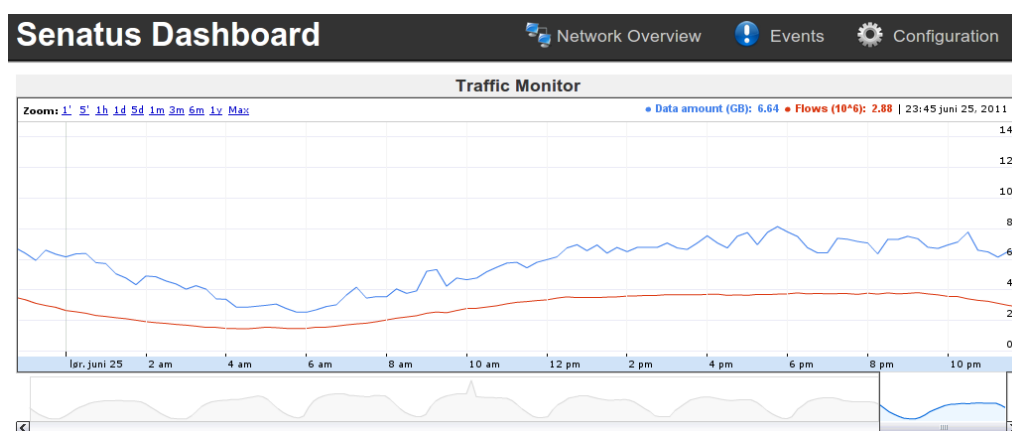


Figure 4.12: Google Chart Tools used for visualizing time bin info.

D3.js D3.js[87], Data-Driven Documents is, like Google Chart Tools, a JavaScript-library for representing data in charts, but is more versatile when it comes to the visualization part than Chart Tools. D3.js is used in the Dashboard to help with manual root-cause analysis of alarms.

IPInfoDB IPInfoDB[88] is an API for IP address lookup, giving information about the country, city, latitude, longitude and host name of the IP address. The Dashboard used IPInfoDB to provide location info per identified attack to use in the map.

4.2.1.4 Database Additions

Two extra tables are needed for the Dashboard to function. Figure 4.13 shows the tables profile and log.

Profile The profile-table controls the gateway currently being analyzed. It is possible to change between different profiles, and in each profile define a number of parameters that Senatus should be run with. Section 4.2.2.1 describes this process in detail, and the following list gives a brief description of each row entry.

- Gateway - the name of the gateway.
- Watchfolder - what directory to monitor.
- Dumpapp - whether to use flow-tools or netflow.
- Timebinsize - The duration in minutes of each time bin.
- Senatuspath - The path to the Senatus executable
- Lambda - What value of lambda to use
- Ksenators - What value of K to use
- Auto - Whether or not to do automatic classification.
- Mode - Whether to use H1 or H2.
- Timespan - How many hours back in time to base PCP on.
- Rule - Which of the different rules for combining suspicious flows should be used.
- Filename format - How is the date formatted in the filename.
- Path format - How is the folder structure formatted based on date.
- Comment - Write a comment for this series of executions.
- DDoS-, DoS- and SCAN-threshold. What threshold to use for automatic classification.

Log The log-table is for error messages and various other status messages created by Senatus. Log messages were previously written to a text-file, but this table is used instead for the administrators' convenience.

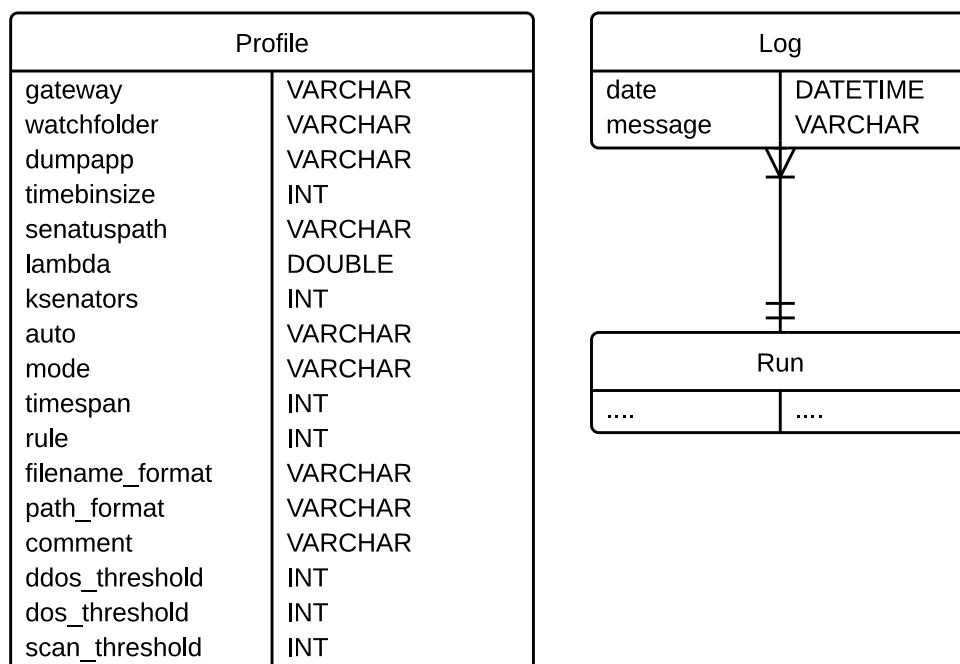


Figure 4.13: Database additions for the Dashboard.

4.2.2 Design

The Dashboard should offer four main interfaces to the network administrator:

- Front page for getting the initial overview of the network, the amount of traffic, and the latest events.
- List of events, identified attacks or unidentified alarms.
- Attack info, all available info on an identified attack, also a link to the corresponding alarm for further analysis of the time bin.
- Alarm info and tools for manual root-cause analysis of unidentified alarms.

Figure 4.14 illustrates the cooperation between the different interfaces.

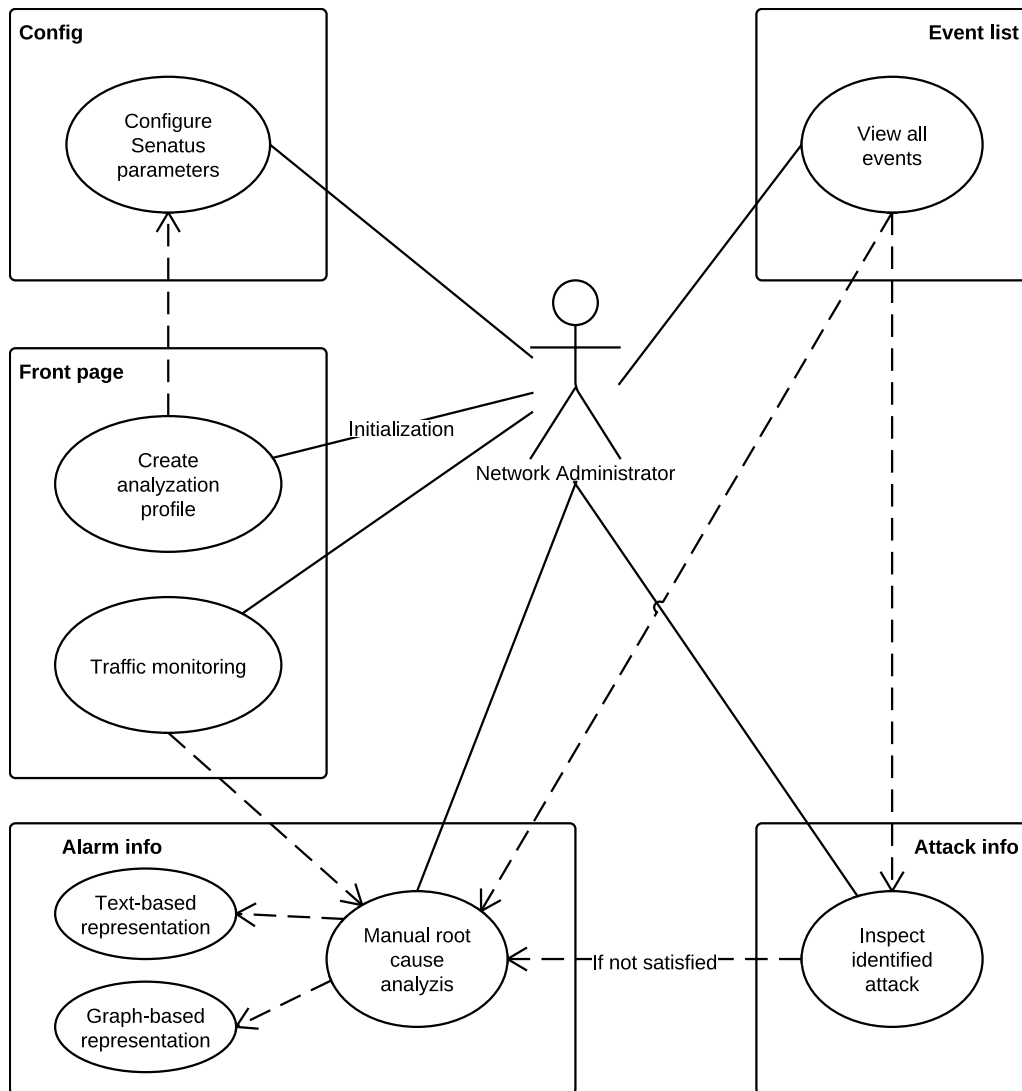


Figure 4.14: Use-case for the Senatus Dashboard.

4.2.2.1 Directory Monitoring

To improve the analysis performance we monitored the directory containing the network data and started the analysis when a new time bin was added.

To manage this, we created a daemon in C++ using inotify[89]. Inotify is a system for notifying when a folder is changed, a file is added to the folder, etc. The application accessed the profile-table directly to get parameters.

This solution was not working perfectly due to permission problems arising

when the daemon did a system call to Senatus that required system calls.

An alternative solution was then developed based on a cron job.¹ Whenever the profile is edited, the Dashboard writes a bash-script to disk.

When the cron job runs this bash-script, the given directory is checked for new time bins. If a new time bin is found, Senatus is run for the time bin with the parameters stored in the profile.

4.2.3 Examples

4.2.3.1 Front page

A portion of the front page is shown above in Figure 4.12. It will show the traffic over time, and also the latest anomalies.

4.2.3.2 Event list

Figure 4.15 shows a list of events sorted by date. Identified attacks are on the top, and alarms are on the bottom. From this particular view the DDoS against 131.154.130.49 is most notable.

4.2.3.3 Attack Details

Details of an identified attack can be seen in this interface, shown in Figure 4.16. The destination IP is marked on the map, the number of flows to this IP, and the flow-cat filtered on this destination IP is printed below the details.

4.2.3.4 Alarm info

This section of the Dashboard provides different tools for manually finding the root cause of an alarm. Figures 4.17 and 4.18 shows a Tension graph developed with d3js to show relations between the top n IPs of an alarm. Figure 4.19 shows a heat graph, also from d3js. These graphs give an intuitive way of identifying attacks by illustrating flows from source to destination.

4.3 Automatic root-cause classification

The automatic root-cause classification described in Section 3.5 was originally implemented in Python. The script take a netflow filename and the

¹A long running process that executes commands at specific dates and times[90].

Senatus Dashboard						
Identified Attacks						
Time:	Cause:	Source IP:	Destination IP:	Destination Port:	Show more:	
2011-06-26 23:00:00	DDOS	-	212.118.142.151	-	More info	
2011-06-26 22:00:00	DDOS	-	193.198.16.211	-	More info	
2011-06-26 21:45:00	DDOS	-	193.198.16.211	-	More info	
2011-06-26 21:30:00	DDOS	-	46.252.192.17	-	More info	
2011-06-26 21:15:00	DDOS	-	46.4.94.180	-	More info	
2011-06-26 20:45:00	DDOS	-	193.170.140.78	-	More info	
2011-06-26 19:00:00	DDOS	-	193.198.16.211	-	More info	
2011-06-26 17:45:00	DDOS	-	139.91.70.48	-	More info	
2011-06-26 07:30:00	DDOS	-	131.154.130.49	-	More info	
2011-06-26 06:15:00	DDOS	-	131.154.130.49	-	More info	
2011-06-26 05:00:00	DDOS	-	131.154.130.49	-	More info	
2011-06-26 04:45:00	DDOS	-	131.154.130.49	-	More info	
2011-06-26 04:30:00	DDOS	-	131.154.130.49	-	More info	
2011-06-26 04:15:00	DDOS	-	131.154.130.49	-	More info	
2011-06-26 04:00:00	DDOS	-	131.154.130.49	-	More info	
2011-06-26 03:45:00	DDOS	-	131.154.130.49	-	More info	
2011-06-26 03:30:00	DDOS	-	131.154.130.49	-	More info	
2011-06-26 03:15:00	DDOS	-	131.154.130.49	-	More info	
2011-06-26 03:00:00	DDOS	-	131.154.130.49	-	More info	
2011-06-26 02:45:00	DDOS	-	131.154.130.49	-	More info	
2011-06-26 02:30:00	DDOS	-	131.154.130.49	-	More info	
2011-06-26 02:15:00	DDOS	-	131.154.130.49	-	More info	
2011-06-26 02:00:00	DDOS	-	131.154.130.49	-	More info	
2011-06-25 23:00:00	DDOS	-	139.91.70.48	-	More info	
2011-06-25 23:00:00	DDOS	-	139.91.70.48	-	More info	


[View all attacks](#)

All alarms						
Time:	Source AS:	Source Port:	Destination AS:	Destination Port:	Show more:	Identified attacks:
2011-06-26 23:15:00	2108	-	6802	-	More info	-
2011-06-26 23:15:00	2108	-	1955	-	More info	-
2011-06-26 23:15:00	20965	-	1853	-	More info	-
2011-06-26 23:15:00	2108	-	1955	-	More info	-
2011-06-26 23:15:00	20965	-	20940	-	More info	-
2011-06-26 23:15:00	2108	-	20940	-	More info	-
2011-06-26 23:15:00	2108	-	20940	-	More info	-
2011-06-26 23:15:00	2108	-	1853	-	More info	-
2011-06-26 23:15:00	2108	-	20940	-	More info	-
2011-06-26 23:15:00	20965	-	20940	-	More info	-
2011-06-26 23:15:00	20965	-	20940	-	More info	-
2011-06-26 23:15:00	20965	-	20940	-	More info	-
2011-06-26 23:15:00	20965	-	20940	-	More info	-
2011-06-26 23:00:00	20965	-	20940	-	More info	-
2011-06-26 23:00:00	20965	-	1299	-	More info	DDOS
2011-06-26 22:30:00	2107	-	12046	-	More info	-
2011-06-26 22:30:00	2107	-	1213	-	More info	-
2011-06-26 22:30:00	2107	-	1955	-	More info	-
2011-06-26 22:30:00	36040	-	1955	-	More info	-
2011-06-26 22:15:00	2107	-	11537	-	More info	-
2011-06-26 22:00:00	6509	-	2108	-	More info	DDOS
2011-06-26 21:45:00	2107	-	22822	-	More info	-
2011-06-26 21:45:00	2107	-	2108	-	More info	DDOS
2011-06-26 21:30:00	2108	-	3549	-	More info	DDOS
2011-06-26 21:15:00	2108	-	1299	-	More info	DDOS
2011-06-26 21:00:00	20940	-	2614	-	More info	-

[View all alarms](#)

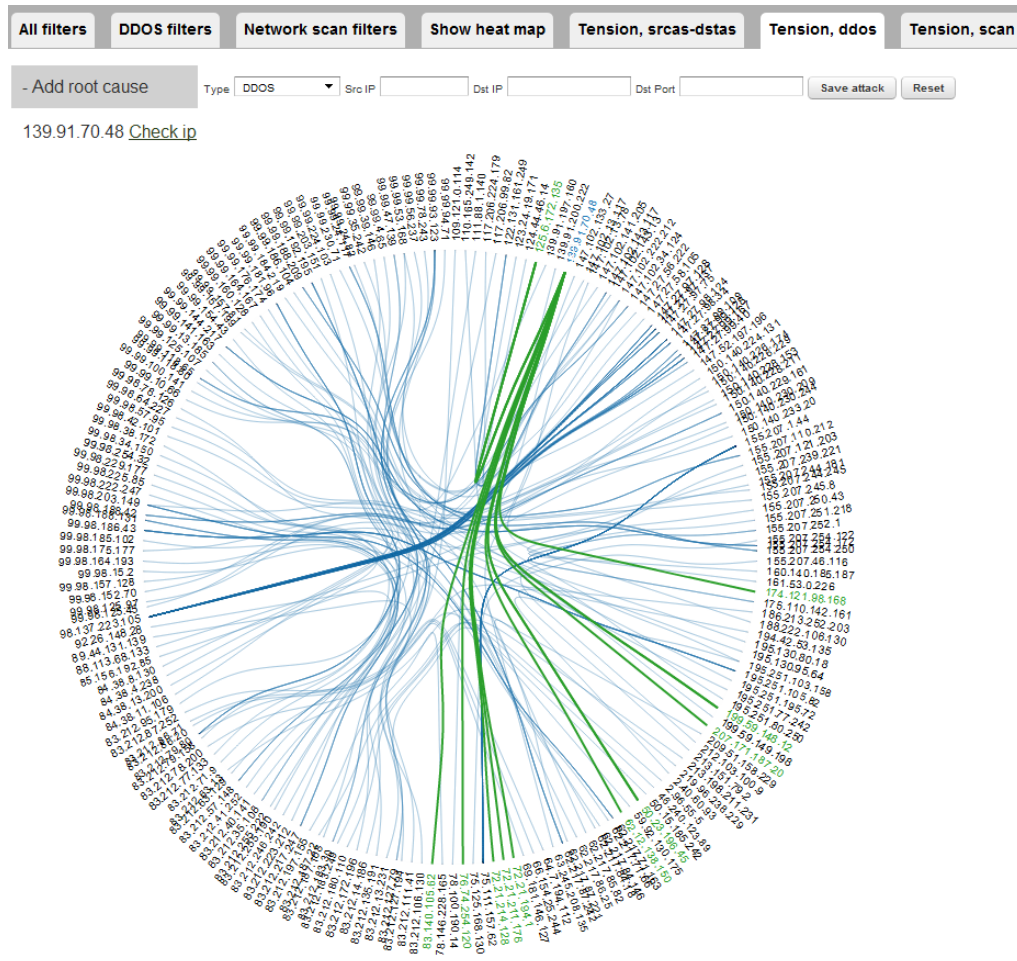
Figure 4.15: The event list.

Senatus Dashboard	
Attack Info	
Date:	2011-06-25
Time:	23:00:00
Attack type:	DDOS
Destination IP:	139.91.70.48
Flow count:	14815
Intensity:	0.91060070758254%
Originating alarm:	Alarm
Delete attack:	Delete



Network Log											
Start	End	Sif	SrcIPaddress	SrcP	Dif	DestIPaddress	DstP	F	F1	Pkts	Octets
0625.23:59:56.778	0625.23:59:56.918	185	50.16.229.9	80	162	139.91.70.48	32868	6	1	2	104
0626.00:00:00.459	0626.00:00:00.459	185	199.59.148.12	80	162	139.91.70.48	53764	6	1	1	52
0625.23:58:59.529	0625.23:58:59.529	185	208.75.122.131	80	162	139.91.70.48	54256	6	0	1	1500
0625.23:58:55.700	0625.23:58:55.700	185	208.75.122.131	80	162	139.91.70.48	53186	6	0	1	1500
0625.23:58:59.990	0625.23:59:00.450	185	208.53.48.134	80	162	139.91.70.48	34890	6	0	2	3000
0625.23:58:51.421	0625.23:58:51.421	185	208.69.40.107	80	162	139.91.70.48	45999	6	0	1	52
0625.23:58:50.511	0625.23:58:50.511	185	173.236.15.152	80	162	139.91.70.48	36549	6	0	1	1500
0625.23:58:53.951	0625.23:58:53.951	185	208.75.122.131	80	162	139.91.70.48	52783	6	0	1	1283
0625.23:58:54.781	0625.23:58:54.781	185	208.75.122.131	80	162	139.91.70.48	52938	6	0	1	1500
0625.23:58:59.612	0625.23:58:59.612	185	184.73.255.7	80	162	139.91.70.48	56416	6	0	1	52

Figure 4.16: Details of an attack.



(a) A tension graph showing a DDoS to 139.91.70.48

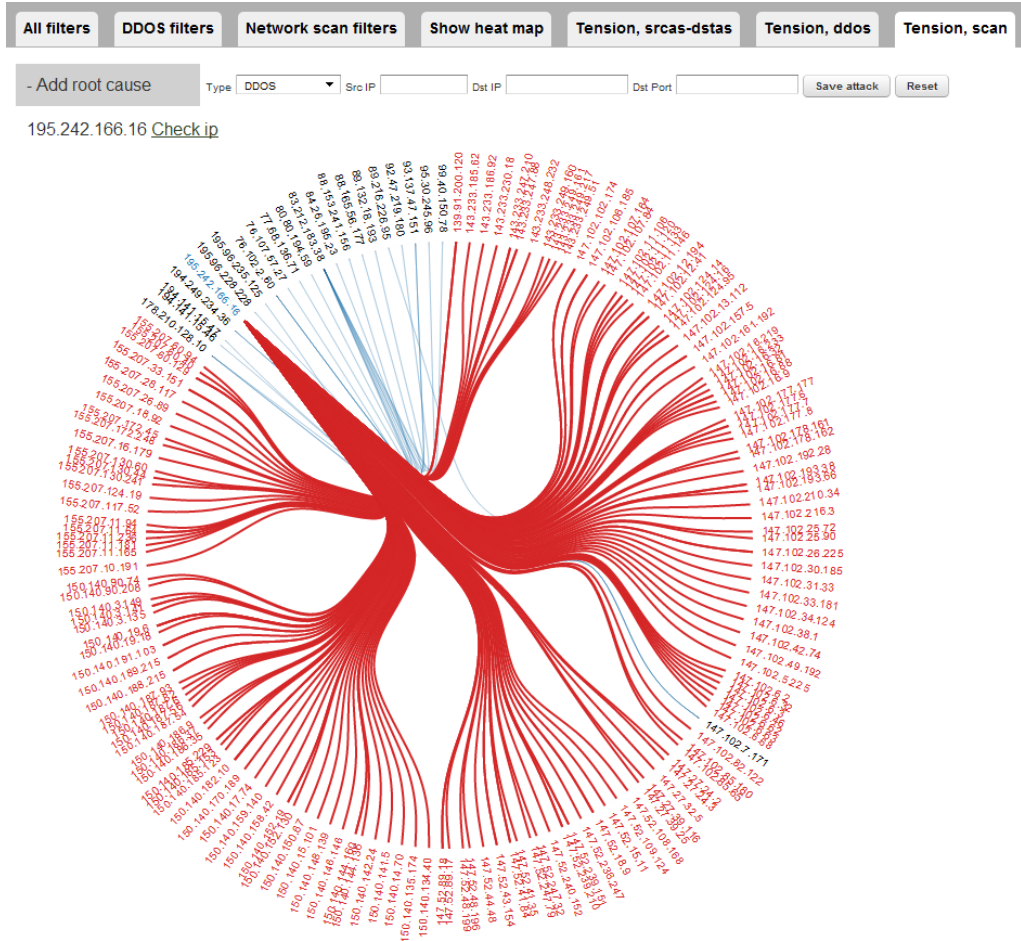
Senatus Dashboard Network Overview Events Configuration

Filtering on IP: 139.91.70.48
Lines: 14815

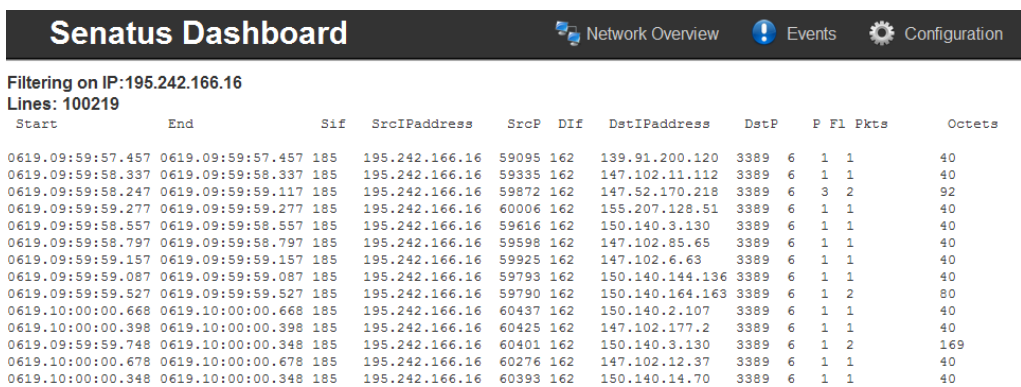
Start	End	Sif	SrcIpAddress	SrcP	Dif	DstIpAddress	DstP	P Fl	Pkts	Octets
0625.23:59:56.778	0625.23:59:56.918	185	50.16.229.9	80	162	139.91.70.48	32868	6	1	2
0626.00:00:00.459	0626.00:00:00.459	185	199.59.148.12	80	162	139.91.70.48	53764	6	1	1
0625.23:58:59.529	0625.23:58:59.529	185	208.75.122.131	80	162	139.91.70.48	54256	6	0	1
0625.23:58:55.700	0625.23:58:55.700	185	208.75.122.131	80	162	139.91.70.48	53186	6	0	1
0625.23:58:59.990	0625.23:59:00.430	185	208.53.48.134	80	162	139.91.70.48	34890	6	0	2
0625.23:58:51.421	0625.23:58:51.421	185	208.69.40.107	80	162	139.91.70.48	45999	6	0	1
0625.23:58:50.511	0625.23:58:50.511	185	173.236.15.152	80	162	139.91.70.48	36549	6	0	1
0625.23:58:53.951	0625.23:58:53.951	185	208.75.122.131	80	162	139.91.70.48	52783	6	0	1
0625.23:58:54.781	0625.23:58:54.781	185	208.75.122.131	80	162	139.91.70.48	52938	6	0	1
0625.23:58:59.612	0625.23:58:59.612	185	184.73.255.7	80	162	139.91.70.48	56416	6	0	1
0625.23:59:00.583	0625.23:59:00.583	185	174.120.243.92	80	162	139.91.70.48	54300	6	0	1
0625.23:58:54.353	0625.23:58:54.353	185	199.59.148.12	80	162	139.91.70.48	38603	6	0	1
0625.23:58:51.533	0625.23:58:52.183	185	112.78.219.185	80	162	139.91.70.48	57047	6	0	3
0625.23:58:58.589	0625.23:58:58.589	185	87.240.188.250	80	162	139.91.70.48	55918	6	0	1
0625.23:59:02.520	0625.23:59:02.520	185	209.190.61.22	80	162	139.91.70.48	38737	6	0	1
0625.23:58:56.500	0625.23:58:56.500	185	174.121.33.251	80	162	139.91.70.48	48380	6	0	1

(b) Filtering on the IP from the DDoS in the tension graph, showing 14815 incoming flows.

Figure 4.17: Alarm info tension graph for DDoS



(a) A tension graph showing a Network Scan from 195.242.166.16



(b) Filtering on the IP from the Network scan in the tension graph, showing 100219 scans.

Figure 4.18: Alarm info tension graph for network scan

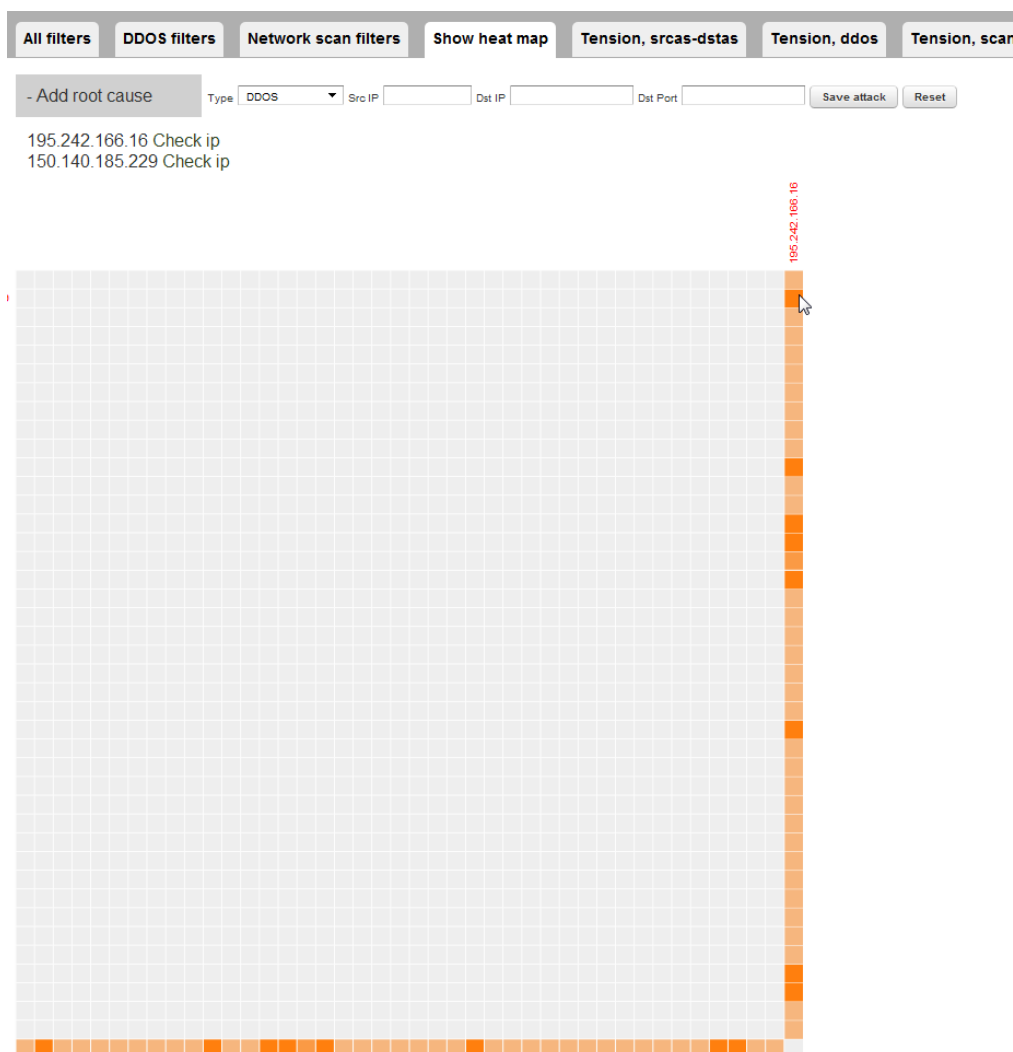


Figure 4.19: Heat graph showing a Network Scan from 195.242.166.16.

Edit vie.at

- Big changes

Watchfolder	<input type="text" value="/home/andersem/netflow/vie.at"/>
Timebin type	<input type="text" value="flow-tools"/>
Timebin size	<input type="text" value="15"/>
Default profile	<input checked="" type="checkbox"/>
Senatus path	<input type="text" value="/home/andersem/workspace/Senatus-dev/Debug/Senatus-dev"/>

Filename format	<input type="text" value="ft-v05.%Y-%m-%d.%H%M"/>
Path format	<input type="text" value="%Y/%Y-%m/%Y-%m-%d"/>
Lambda	<input type="text" value="2"/>
K-senators	<input type="text" value="20"/>
Timespan	<input type="text" value="24"/>
Mode	<input type="text" value="H1"/>
Auto	<input type="text" value="DDOS"/>
Rule	<input type="text" value="3"/>
Comment	<input type="text"/>

- Thresholds

DDOS Threshold	<input type="text" value="7400"/>
DOS Threshold	<input type="text" value="1400"/>
Port sweep Threshold	<input type="text" value="200"/>

Figure 4.20: The configuration options for the Dashboard.

4 feature values from an alarm (src AS, dst AS, src port, dst port) as input, analyzes the netflow file from a time bin with flow-tools and try to determine the root-cause of the attack.

The script is run with the following command

```
./auto_classification.py < pathnetflowfile >< src_AS >< dst_AS >< src_port >< dst_port >
```

Python is slower than compiled languages like C and Java, but the time used to interpret the code is neglectable compared to the time used by flow-tools which is used for the filtering of the network traffic files. Therefore we have preferred the advantage of Python code being much easier to write and change fast. This has been a great advantage for testing and tweaking the algorithm.

The final design of the algorithm is also implemented in C++ as part of version 2 of the Senatus back-end.

Chapter 5

Evaluation

In this chapter we are presenting an evaluation of the performance and tuning of parameters.

The first sections presents the data sets, ground truth and the tuning parameters used for the performance analysis of anomalies detected by Senatus version 0.

Section 5.5 focuses mostly on the execution time tuning and a limited detection rate analysis of version 2.

5.1 Data set

The data used in the evaluation is sampled NetFlow data collected from four different routers in the GÉANT network [91]. The GÉANT network is a backbone Internet provider with a pan-European communications infrastructure connecting Europe's research and education community. The network is co-founded by European National Research & Education Networks (NRENs) which in turn is responsible for providing Internet access to research and education communities within each country.

The data was collected on traffic links between GÉANT routers and neighbouring ASes, from June 18th to July 5th 2011, excluding June 22nd 2011. The measuring links were located in Amsterdam, Copenhagen, Frankfurt and Vienna. Table 5.1 shows an overview of the links, and it is worth noting that Vienna and Frankfurt are a lot bigger than Copenhagen and Amsterdam in terms of the average number of flows captured per time bin. Figure 5.1 gives an overview of the geographical location of the links, which are marked with green. The traffic was captured in 15 minute time bins and sampled at a rate of 1:1000.

Trace	Location	Avg. # of flows
A	Vienna	1,699,687
B	Frankfurt	1,450,190
C	Amsterdam	293,580
D	Copenhagen	299,347

Table 5.1: Overview of the measurement links

5.2 Ground-truth construction

Our way to overcome the lack of the ground-truth in our data set is based on a manual inspection of the set of flagged alarms of Senatus and the histogram-based detection (HBD) method described in Section 2.2.2.5. We ran both SENATUS and HBD on the collected traces and manually inspected the flagged alarms. If the root-cause of the alarm is successfully identified in an anomalous bin, the anomaly is added to the ground-truth.

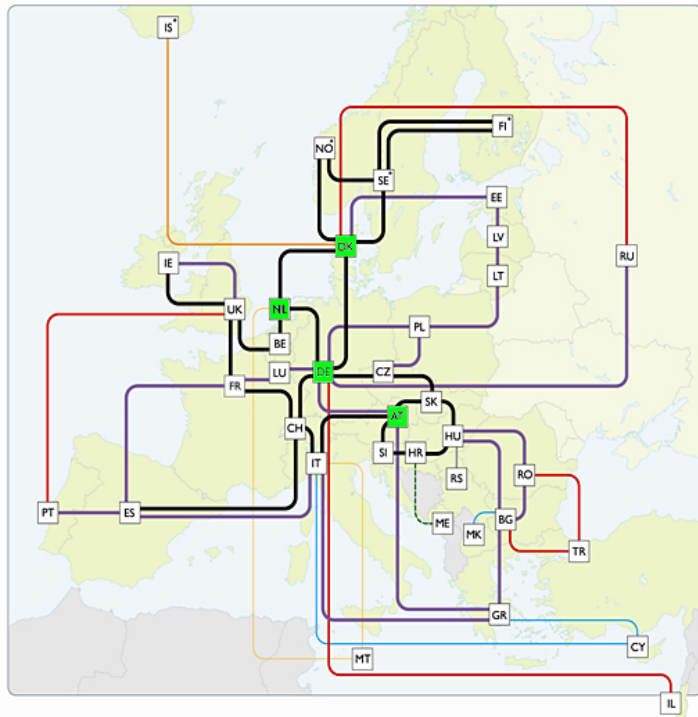


Figure 5.1: The GÉANT2 network [1]

While constructing the ground-truth, we further noticed that SENATUS can give rise to different set of network anomalies as the heuristics preceding

senators election, mentioned in Table 3.2, varies. For example, we noticed that if H1 was deployed for senators election, Senatus tended to detect more DDoS attacks, whereas if H2 was deployed Senatus detected more scan activities (see results in Section 5.4.5). Therefore we have chosen our ground-truth to be the union of Senatus H1, Senatus H2 and HBD, as shown in equation 5.1.

$$GroundTruth = SenatusH1 \cup SenatusH2 \cup HBD \quad (5.1)$$

5.3 Tuning Parameters

Chapter 3 describes the framework of Senatus and introduces some parameters that will affect the anomaly detection performance. To find the best performance we have used an experimental approach where we have tuned the parameters and compared the results from Senatus with the ground truth described in the previous section. In Chapter 3 each parameter is thoroughly described and some initial values for the parameters are proposed. In this section we analyze them and verify how well they perform. Table 5.2 lists the parameters that can be tuned, and their constraints.

Parameter	Description	Constraints
α	flow size in packets	small
β	flow size in bytes	small
K	number of senator	small
λ	PCP weighting parameter	≥ 2
j	flow aggregation level	$[1, 4]$
$\theta_i, i = 1..3$	root-cause decision threshold	large

Table 5.2: Tuning parameters

5.3.1 Traffic Filtering Heuristics

The parameters α and β are the heuristics from Table 3.2 used in the construction of the population before the election part of Senatus. From the discussion in Section 3.2.1, we saw that a previous study [29] had discovered that most of the encountered anomalies in their data set are carried by flows having a number of packets $\in [1, 3]$, and that another study [92] claim that most of the detected scans are carried by flows having a number of packets ≤ 2 . Our results support these studies. Figure 5.2 illustrates the

distribution of the average flow size in term of packets and bytes for the detected anomalies in our traces. We see that while most of the anomalous flows have an average size of one packet, anomalous flows involving less than 3 packets are in the order of 85% of all attacks in our data set. In Senatus H1 we therefore choose a value α of 3.

The discussion about the value of β in Section 3.2.1 suggested a value between 40 and 144 bytes, and our inspection of the detected attacks in our data set shows similar results. Despite a long tail due to variable size of the different anomalous flows, most frequent DoS/DDoS and scan attacks in our data set are of a small size (≤ 64 bytes). For example 52% of the detected DoS attacks and 99% of the detected scans carry flows of size less than 60 bytes. We choose our threshold β to be 64.

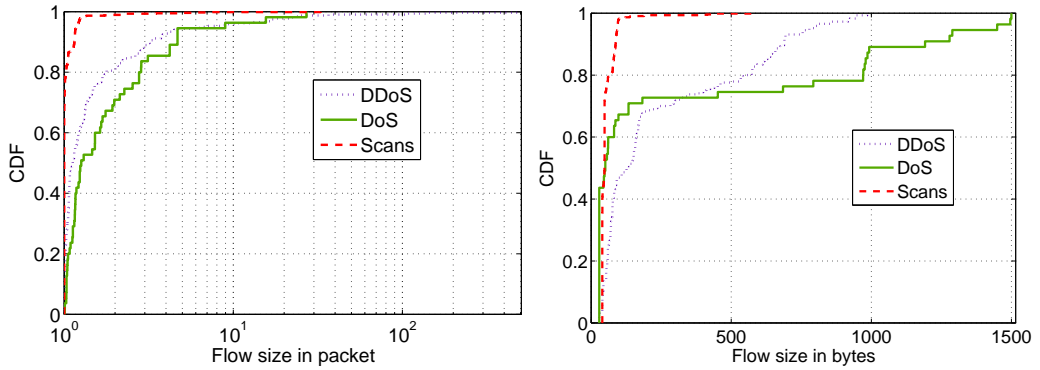


Figure 5.2: Distribution of the average flow size

5.3.2 The PCP Tuning Parameter λ

The parameter λ is discussed in Section 3.3.2. As PCP aims to minimize the weighted combination of the nuclear norm, and of the ℓ_1 -norm, one has to identify the appropriate value of the weighting parameter λ such that the matrix A is sparse while capturing the maximum number of anomalies with the least false-positive rate. The parameter λ is in the form:

$$\lambda = \frac{C}{\sqrt{\max N, K}}, C \in \mathbb{R} \quad (5.2)$$

We base our analysis on the previous observations [44] which propose a parameter value of $C = 2$, and we tune the parameter C to find an optimal detection/false-positive trade-off. The detection- and false-positive rates as a function of C are illustrated in Figure 5.3. The figure shows that both

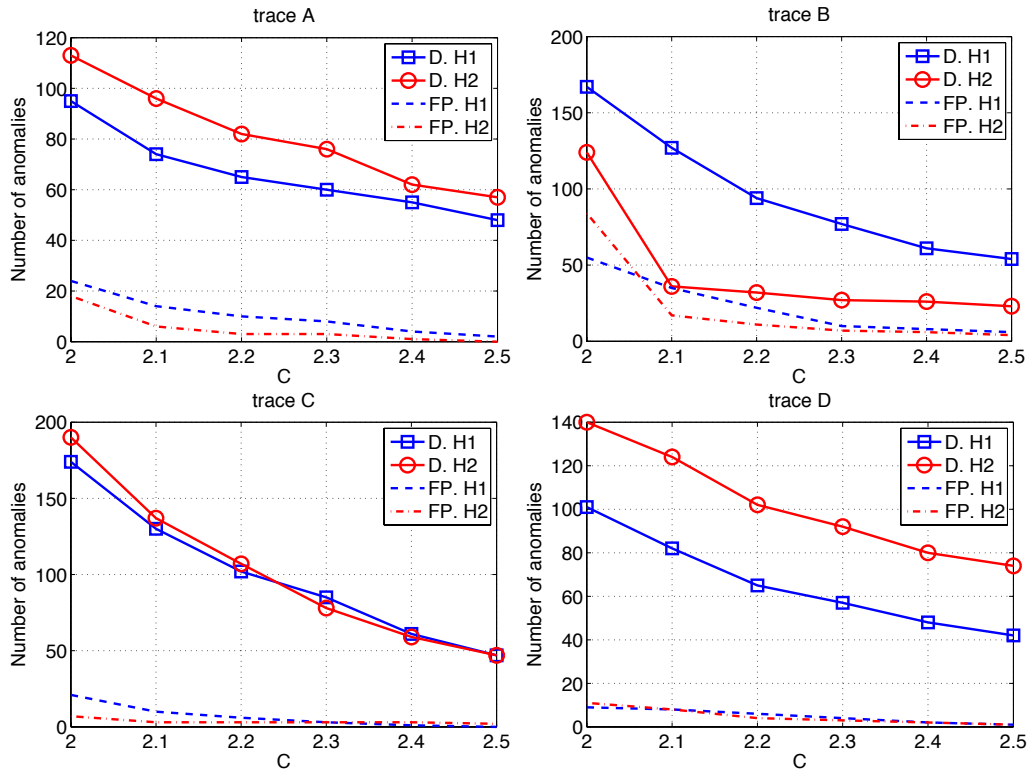


Figure 5.3: Detection and false-positive rates as a function of C

the detection and false-positive rates decrease as the value of C increases. For example 190 anomalies are detected with 7 false positives for the value of $C = 2$, while 47 anomalies are detected with only 2 false positives for $C = 2.5$ in trace C , Amsterdam, when the heuristic H2 is chosen. The figure additionally shows that while the number of false positives when H1 is chosen is higher than those when H2 is set up, it remains relatively low for all values of C .

5.3.3 Root-Cause Thresholds

In Section 3.5 we describe an algorithm that automatically finds the root-cause of an attack. Algorithm 2 shows the first version of the design. The algorithm needs a set of threshold values, θ_1 , θ_2 and θ_3 , and in this section we are discussing how to set these values. During the process of tuning the threshold values we also propose several hypothetically improved versions of the algorithm. Each version is evaluated by using an experimental approach where we measure the classification performance by inspecting the

suspicious flows flagged by Senatus, and compare the results to our ground truth.

For each version of the algorithm we have presented performance graphs. The graphs are generated by varying the threshold values in the range [200, 10.000], which covers most of the attack intensities, and calculate the corresponding classification rate based on the constructed ground truth for each trace. Our goal is to combine all possible threshold values to find the combination leading to the highest anomalies classification rate. We vary all the thresholds values, but for the versions of algorithm using three thresholds $(\theta_1, \theta_2, \theta_3)$, we have chosen to present the graphs in three dimensions $(\theta_1, \theta_2, ClassificationRate)$ instead of four $(\theta_1, \theta_2, \theta_3, ClassificationRate)$ for better readability. In all cases this is performed by setting the last dimension to the discovered optimal value, $\theta_3 = 200$. To further improve readability, we have marked the combination of thresholds that results in the best classification performance with dots.

5.3.3.1 Dynamic Threshold

The value of θ might be set up dynamically as the expected number of flows implicating the flow tuple x during the current anomalous time bin, $\theta = M_{fx}(t)$. The expected number of flows might be decided using a forecast model such as the EWMA algorithm given the apriori knowledge of the previous number of anomalous flows for the flow tuple x , at time interval $t - 1$, $M_x(t - 1)$:

$$M(t)_{fx} = \begin{cases} \alpha M_x(t - 1) + (1 - \alpha)M_{fx}(t - 1) & \text{if } t > 2 \\ M_x(1) & \text{if } t = 2 \end{cases} \quad (5.3)$$

Equation 5.3 illustrates that the prediction procedure requires some knowledge about the behavior of the number of flows for the tuple x in the past, which adds a non negligible complexity and processing overhead for the root-cause analysis algorithm. We propose a fixed threshold to bypass this issue.

5.3.3.2 Version 1.1 (One Threshold)

The first and most intuitive way to configure the algorithm thresholds is to have the same value for all the three thresholds ($\theta_i = \theta, i = 1..3$). The choice of the threshold values is summarized as finding the "optimal" minimum support that achieves the highest classification accuracy. Figure 5.4 illustrates the classification accuracy as a function of the minimum support

Algorithm 2 Root-cause($\mathcal{F}, \theta_1, \theta_2, \theta_3$)

Input: Suspicious flows $\mathcal{F} = \{f_1, \dots, f_i, \dots, f_n\}$

 threshold $\theta_1, \theta_2, \theta_3$
 \mathcal{A}_x set of flows of feature value x
Output: The root cause

```

1: for  $f_i \in \mathcal{F}$  do
2:    $x \leftarrow dstIP$ 
3:   if ( $max(\mathcal{A}_x) | x \in f_i(dstAs) > \theta_1$ ) then
4:     return DDOS,  $dstIP_{max}$ 
5:   else
6:      $x \leftarrow \{srcIP, dstIP\}$ 
7:     if ( $max(\mathcal{A}_x) | x \in f_i(\{srcAs, dstAs\}) > \theta_2$ ) then
8:       //  $x$  is the pair of  $srcIP$  and  $dstIP$  within the  $srcAs$  and  $dstAs$ 
9:       // flagged in the anomalous flow  $f_i$ 
10:      return DOS,  $\{srcIP, dstIP\}_{max}$ 
11:    end if
12:   else
13:      $x \leftarrow \{srcIP, dstPort\}$ 
14:     if ( $max(\mathcal{A}_x) | x \in f_i(\{srcAs, dstPort\}) > \theta_3$ ) then
15:       return Network Scan,  $\{srcIP, dstPort\}_{max}$ 
16:     end if
17:   else
18:     False Positive
19:   end if
20: end for

```

θ for the anomalies manually inspected in the four collected traces. The figure shows that while the classification performance varies between traces, it remains generally low, i.e. does not exceed 75% for the four collected traces. For example the best classification accuracy is about 50% in trace C, Amsterdam, for the value of threshold $\theta = 1200$ while it is 74.75% in trace A, Vienna, for the value of threshold $\theta = 1800$.

5.3.3.3 Version 1.2 (Three Thresholds)

To improve the classification accuracy, we choose to vary all three threshold values. In the following graphs, Figure 5.5, we fix the threshold θ_3 to the value of 200 that achieves the best classification performance for all traces and expose the classification rate while varying the first two thresholds (θ_1 and θ_2). The figure shows that while the classification performance

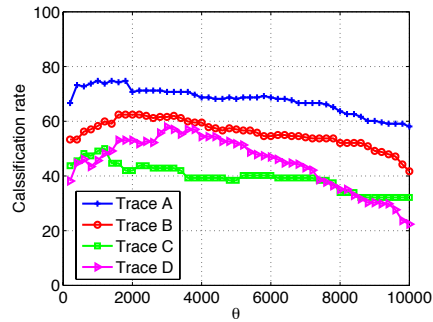


Figure 5.4: Version 1 classification rates

may vary from 38% to 88%, over 83% of the manually inspected anomalies are well classified using the "optimal" threshold values for the four collected traces. For example, 84.85% of the anomalies in trace A, Vienna, are correctly classified for the set of thresholds (5800,600,200), while 88.6% are correctly classified in trace D, Copenhagen, for the set of thresholds (6000,1400,200).

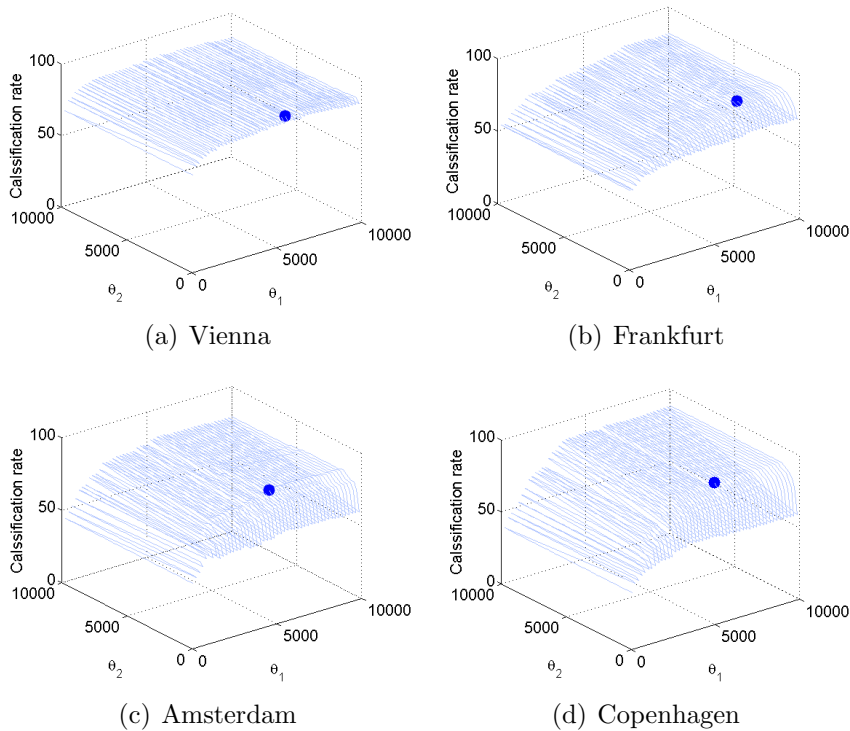


Figure 5.5: Version 1.2 classification rates

5.3.3.4 Version 2 (Two Thresholds)

The evaluation of the first version showed that we had a problem with high intensity DOS attacks. These attacks will be classified as DDoS attacks, since the first version counts the maximum destination IP without taking the number of sources into account. An example would be a DoS attack from source A to destination B with an intensity of 14000 flows. If the threshold θ_1 is below 14000, the attack will be marked as a DDoS, even if it only has one source.

Algorithm 3 Root-cause($\mathcal{F}, \theta_1, \theta_3$)

Input: Suspicious flows $\mathcal{F} = \{f_1, \dots, f_i, \dots, f_n\}$
 threshold θ_1, θ_3
 \mathcal{A}_x set of flows of feature value x

Output: The root cause

- 1: **for** $f_i \in \mathcal{F}$ **do**
- 2: $x \leftarrow dstIP$
- 3: **if** $(max(\mathcal{A}_x)|x \in f_i(dstAs) > \theta_1)$ **then**
- 4: // extended first statement
- 5: **if** $srcCount(dstIP_{max})=1$ **then**
- 6: **return** DOS, $dstIP_{max}$
- 7: **else**
- 8: **return** DDOS, $dstIP_{max}$
- 9: **end if**
- 10: // second statement removed
- 11: **else**
- 12: $x \leftarrow \{srcIP, dstPort\}$
- 13: **if** $(max(\mathcal{A}_x)|x \in f_i(\{srcAs, dstPort\})) > \theta_3$ **then**
- 14: **return** Network Scan, $\{srcIP, dstPort\}_{max}$
- 15: **end if**
- 16: **else**
- 17: False Positive
- 18: **end if**
- 19: **end for**

As a proposed approach to solve this problem we tried to add an additional statement after measuring the first threshold θ_1 . This statement checks if the maximum destination IP has one or multiple sources sending data. If there is only one source, the attack is classified as a DoS, and if there are multiple sources the attack is classified as DDOS. This change is illustrated in Algorithm 3. To prevent adding more complexity to the algorithm, we

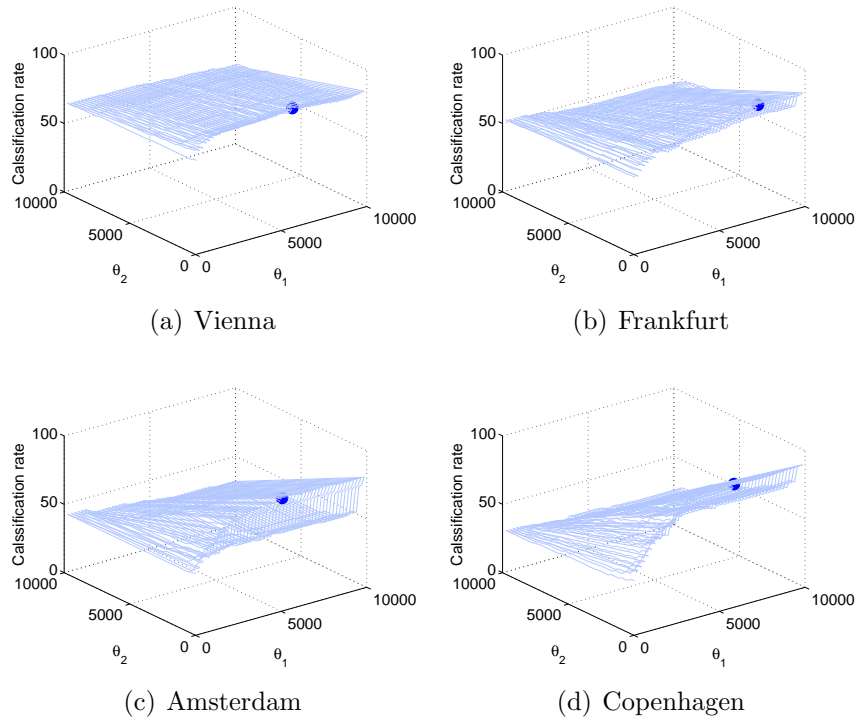


Figure 5.6: Version 2 classification rates

also removed the second statement since the extended first statement would now be able to classify both DDoS and DoS attacks. Another advantage of this is that we now only need two thresholds, θ_1 and θ_3 .

The results from version 2 of the algorithm is shown in Figure 5.6. The correct classification rates spans from 74.12% in trace D, Copenhagen, with the thresholds $\theta_1 = 3800$ and $\theta_3 = 800$, to 85.86% in trace A, Vienna, with the threshold values $\theta_1 = 5800$ and $\theta_3 = 200$. This is slightly more misclassified attacks than version 1. The reason for this seemed to be that a lot of DoS attacks wrongly got classified as DDoS attacks. This was mostly due to benign flows being sent at the same time as the attack. E.g. if a source A starts a DoS attack against destination B and at the same time source C wants to send B a file, our extended check will detect more than one source IP, hence wrongly mark the attack as a DDoS. This version of the algorithm worked well for time bins with no benign traffic being sent to the destination IP during the attack, thus some DoS attacks that was wrongly marked as DDoS in version 1 was corrected to DOS attacks in version 2. This increased the performance, but since we operated with only

one threshold for both DDoS and DoS we also "lost" a lot of DOS attacks with lower intensity than θ_1 , and this decreased the total performance.

5.3.3.5 Version 3 (Three Thresholds)

To also be able to classify DOS attacks with intensity below the threshold θ_1 , we reintroduce the second statement in our algorithm, and at the same time we keep the extended first statement (from version 2). The pseudo code of this version is shown in Algorithm 4. Figure 5.7 shows the performance graphs. The correct classification rate spans from 83.93% in trace C, Amsterdam, with the thresholds $\theta_1 = 5800$, $\theta_2 = 1400$ and $\theta_3 = 200$, to 88.6% in trace D, Copenhagen, with the threshold values $\theta_1 = 6000$, $\theta_2 = 1400$ and $\theta_3 = 200$. These are almost the same results as in version 1.2 of the algorithm. The detection rates are the same, but the thresholds are a little different.

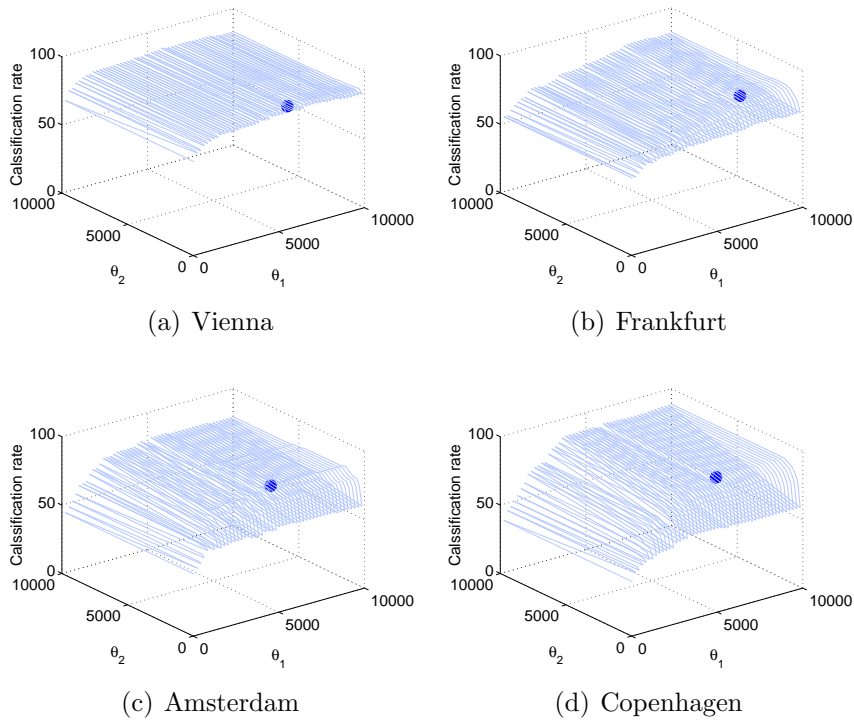


Figure 5.7: Version 3 classification rates

Algorithm 4 Root-cause($\mathcal{F}, \theta_1, \theta_2, \theta_3$)

Input: Suspicious flows $\mathcal{F} = \{f_1, \dots, f_i, \dots, f_n\}$

 threshold $\theta_1, \theta_2, \theta_3$
 \mathcal{A}_x set of flows of feature value x
Output: The root cause

```

1: for  $f_i \in \mathcal{F}$  do
2:    $x \leftarrow dstIP$ 
3:   if  $(max(\mathcal{A}_x)|x \in f_i(dstAs)) > \theta_1$  then
4:     // extended first statement
5:     if  $srcCount(dstIP_{max})=1$  then
6:       return DOS,  $dstIP_{max}$ 
7:     else
8:       return DDOS,  $dstIP_{max}$ 
9:     end if
10:  else
11:     $x \leftarrow \{srcIP, dstIP\}$ 
12:    if  $(max(\mathcal{A}_x)|x \in f_i(\{srcAs, dstAs\})) > \theta_2$  then
13:      //  $x$  is the pair of  $srcIP$  and  $dstIP$  within the  $srcAs$  and  $dstAs$ 
      flagged in the anomalous flow  $f_i$ 
14:      return DOS,  $\{srcIP, dstIP\}_{max}$ 
15:    end if
16:  else
17:     $x \leftarrow \{srcIP, dstPort\}$ 
18:    if  $(max(\mathcal{A}_x)|x \in f_i(\{srcAs, dstPort\})) > \theta_3$  then
19:      return Network Scan,  $\{srcIP, dstPort\}_{max}$ 
20:    end if
21:  else
22:    False Positive
23:  end if
24: end for

```

5.3.3.6 Summary

Table 5.3 gives a summary of the classification rates for each version of the algorithm. The table also includes the rates when the algorithm is applied to the whole data set at the same time.

Type	θ_1	θ_1	θ_1	Correct
Version 1.1 - all	1600	1600	1600	60.13%
Version 1.1 - ams	1200	1200	1200	50.00%
Version 1.1 - cop	3000	3000	3000	67.90%
Version 1.1 - fra	1600	1600	1600	62.40%
Version 1.1 - vie	1000	1000	1000	74.75%
Version 1.2 - all	7400	1400	200	85.64%
Version 1.2 - ams	5600	1400	200	83.93%
Version 1.2 - cop	6000	1400	200	88.60%
Version 1.2 - fra	7800	1400	200	84.30%
Version 1.2 - vie	5800	600	200	87.37%
Version 2 - all	7400	-	200	84.48%
Version 2 - ams	5200	-	200	83.93%
Version 2 - cop	3800	-	800	88.60%
Version 2 - fra	7400	-	200	84.30%
Version 2 - vie	5800	-	200	87.37%
Version 3 - all	7400	1400	200	85.64%
Version 3 - ams	5800	1400	200	83.93%
Version 3 - cop	6000	1400	200	88.60%
Version 3 - fra	7400	1400	200	84.30%
Version 3 - vie	6800	600	200	87.37%

Table 5.3: Automatic Root-cause Analysis Performance

5.4 Performance Analysis

5.4.1 Methodology

Previous work on performance analysis of anomaly detectors have mainly used two different approaches:

- Manual analysis by inspecting the root-cause of each alarm [2, 23, 28].
- Injecting synthetic anomalies into normal network traffic [4, 28, 93]

Manual root-cause analysis is an error-prone and time-consuming task, since each time bin can consist of up to millions of flows. However, a previous performance analysis of Senatus failed to obtain sufficient results when trying to inject anomalies in normal network traffic [94]. We therefore chose to manually inspect the root-cause of each alarm.

An alarm raised by Senatus can be classified in 4 different ways, illustrated in Figure 5.8.

For anomalous traffic:

- **True Positive (TP):** An alarm will be a TP if the manual root-cause analysis of this alarm showed that it was raised due to an anomaly.
- **False Negative (FN):** An alarm will be a FN if the manual root-cause analysis showed that there was an anomaly in that time bin, but no alarm was raised.

For benign traffic:

- **False Positive (FP):** An alarm will be a FP if the manual root-cause analysis of this alarm did not find any anomaly in the time bin.
- **True Negative (TN):** A TN means that no alarm was raised for a time bin, and the time bin did not contain any anomalies.

In this section we discuss the performance of Senatus based on the results we got from a manual analysis of over 9000 alarms from 1885 time bins, detected by Senatus and HBD. The reason for the difference between the number of time bins and number of alarms is that the detectors can raise several alarms in each time bin. If Senatus has flagged multiple alarms for

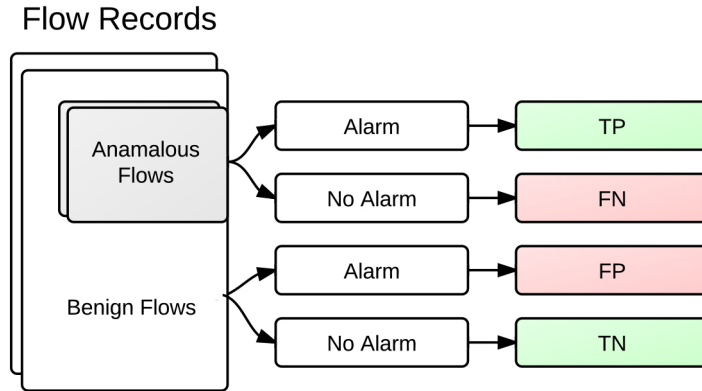


Figure 5.8: Illustration of the detection metrics.

a time bin, we analyze all of them, but we only label the time bin with one attack. If multiple types of attacks are found, we chose to prioritize them in the following order: DDoS >DoS >network scan >port scan. That means if we find both a network scan and a DDoS attack in the same time bin, we label the time bin with a DDoS attack.

5.4.2 Detected Anomalous Time Bins

Table 5.4 shows all the detected anomalous time bins. The union of Senatus H1, Senatus H2 and HBD represents the unique number of anomalous time bins found be all the approaches together. This means that our ground-truth discussed in section 5.2 consist of 1014 unique anomalous time bins.

The intersection of H1, H2 and HBD is very low, only 72 time bins, which is only 0.071% of the total amount of anomalous time bins. We also see that the intersection between H1 and H2 is significantly larger than the intersection between H1 and HBD, and H2 and HBD.

- H1 detects 53% of H2's detected anomalies and 28% of HBD's detected anomalies.
- H2 detects 56% of H1's detected anomalies and 33% of HBD's detected anomalies.
- HBD detects 16% of H1's detected anomalies and 18% of H2's detected anomalies.

	Amsterdam	Copenhagen	Frankfurt	Vienna	Total
Senatus H1	174	101	167	95	537
Senatus H2	190	140	124	113	567
HBD	72	73	66	102	313
$H1 \cup H2 \cup HBD$	321	221	260	212	1014
$H1 \cup H2$	275	165	220	142	802
$H1 \cup HBD$	226	168	211	174	779
$H2 \cup HBD$	241	199	169	186	795
$H1 \cap H2 \cap HBD$	15	20	17	20	72
$H1 \cap H2$	89	76	71	66	302
$H1 \cap HBD$	20	23	22	23	88
$H2 \cap HBD$	21	31	21	29	102

Table 5.4: Detected Anomalous Time Bins

5.4.3 Detection rates

One of the most important measurement tools for an anomaly detection method is the detection rate. The detection rate tells us how well the anomaly detection methods perform against the ground truth. The detection rate is defined as follows:

$$DetectionRate = \frac{DetectionMethod}{GroundTruth}, DetectionMethod \in \{H1, H2, HDB\} \quad (5.4)$$

Table 5.5 gives an overview of the detection rates. We see that the detection rate of Senatus H1 is 71.56% higher than HDB ($H1 = 537, HDB = 313$) and Senatus H2 has a 81.15% higher detection rate than HDB ($H2 = 567, HDB = 313$).

Detection method	Detection rate
Senatus H1	52.96%
Senatus H2	55.92%
HBD	30.87%

Table 5.5: Overall Detection Rates

5.4.4 False Positive Rates

The false positive rate is defined in Equation 5.5 for each link.

$$FalsePositiveRate = \frac{NumberOfFalsePositives}{NumberOfFlaggedTimeBins} \quad (5.5)$$

Figure 5.9 gives an overview of the false positives rates on the all the links for Senatus H1, Senatus H2 and HBD. Senatus H1 and H2 have the same false positives rates and that HBD's false-positive rates are much higher. Senatus H1 has a FP rate of 10.77% for Amsterdam and 8.18% for Copenhagen, and Senatus H2 is even better with a FP rate of 3.55% for Amsterdam and 7.28% for Copenhagen. The FP rate for HBD on these links is significantly higher with a rate of 32.71% for Amsterdam and 37.06% for Copenhagen. Frankfurt has the highest FP rate for all the detection methods with 24.77% for Senatus H1, 40.38% for Senatus H2 and 52.86% for HBD. The average FP rate was 15.97% for Senatus H1, 16.24% for Senatus H2 and 39.92% for HBD.

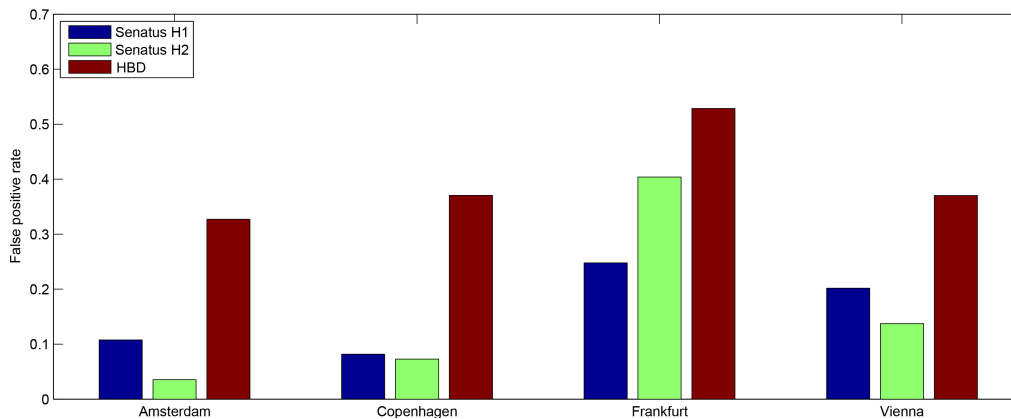


Figure 5.9: Overview of False positive rates for the different links

5.4.5 Detected Attacks

This section gives an overview over the type of attacks detected in the anomalous time bins. Figure 5.10 shows a summary of the attacks on all links combined, and Figure 5.11 shows the attacks detected on each link. Senatus H1 and HBD detects almost the same amount of DDoS/DoS attacks ($H1 = 125$, $HBD = 130$), whereas Senatus H2 detects less ($H2 = 75$). Senatus H2 detects 50% more scans than Senatus H2 and 80% more than HBD. HBD only detected 5 occurrences of network experiments (described in Section 2.1.3). Almost all of the about 175 network experiments detected by Sentus H1 and 150 detected by Senatus H2 were detected on the Amsterdam link.

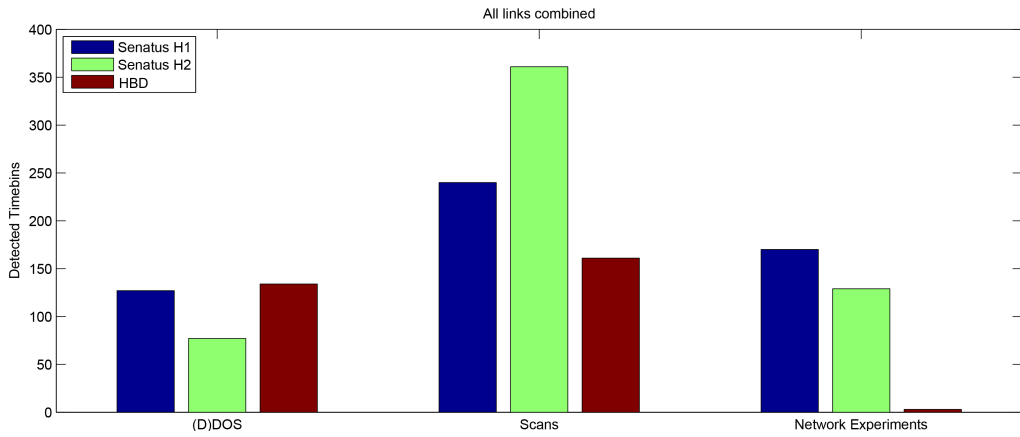


Figure 5.10: Detected attacks on all links combined

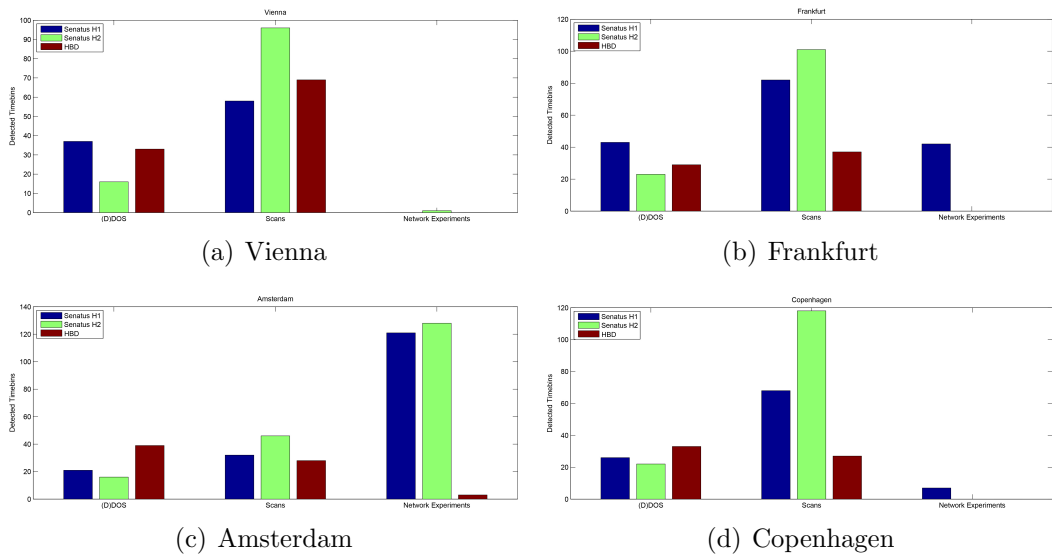


Figure 5.11: Detected attacks on each link

5.5 Implementation Performance Evaluation

This section mostly focuses on measurements of execution time of version 2 of the implementation of Senatus. However, the execution time can not be the prerogative criteria, we need to take the detection rate into the consideration as well.

These benchmarks were first possible to execute after finishing version 2,

thus leaving limited time for benchmarking. Therefore, the data set in this performance analysis is limited to 24 hours from one link, Vienna. Vienna was chosen because it had the highest amount of network traffic.

Because of the limited data set, the detection graphs are only indicative, and might show false patterns. Chapter 5 offers a more thorough and accurate detection rate analysis, thus wherever the two differs the results from the evaluation chapter should be prioritized.

5.5.1 Online versus Offline Version

Figure 5.12 illustrates the gain of using a database for storing senator values. The initial execution of the online version demands the same amount of time since the database is empty.

The data shown here is limited to 00:00 to 17:45 because of the high execution time of the offline version.

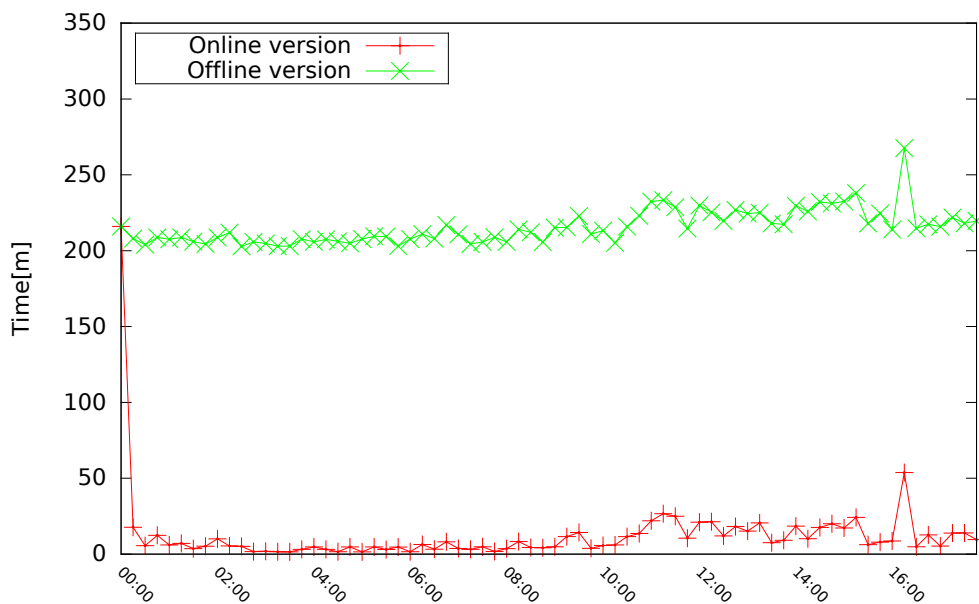


Figure 5.12: Total execution time, offline vs. online version.

5.5.2 Different Links

The default link for these measurements is Vienna, as it has largest average number of flows per time bin. Figure 5.13 shows how the execution time

varies when Senatus is run on a link with a much lower average flow count per time bin.

Table 5.1 shows the average number of flows for the links.

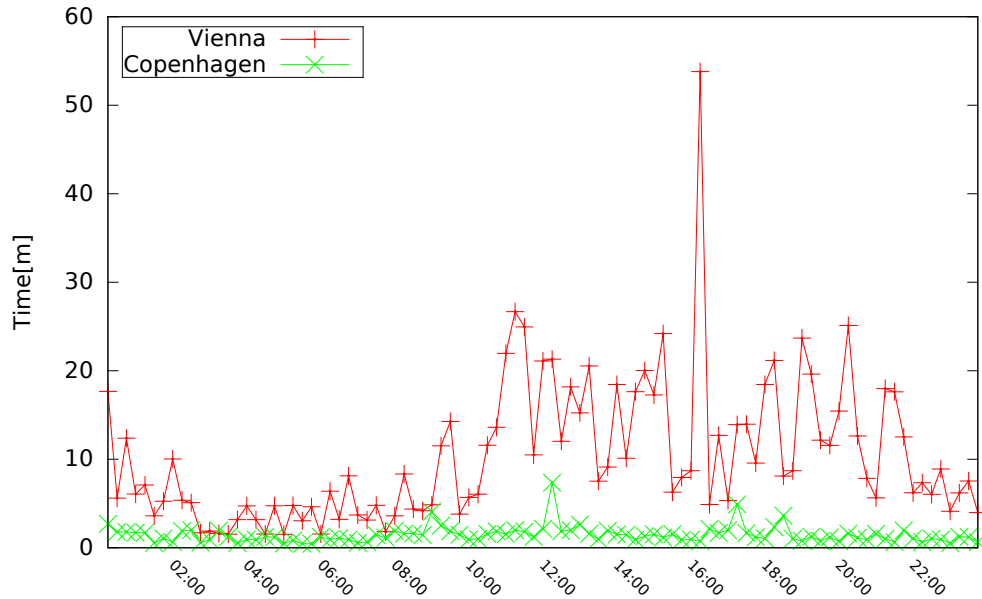


Figure 5.13: Execution time for Vienna and Copenhagen.

5.5.3 Measurement Parameters

The parameters are as shown in 4.1.5.2:

- λ - PCP tuning parameter
- K - the K number of senators
- t - timespan
- Tuning automatic root-cause analysis and decision rules together
- Heuristics, H1 and H2 with α and β

5.5.3.1 Reference Parameter Values

The reference parameter values are shown in Table 5.6. These values are the same as the default values used in the evaluation chapter, Chapter 5.

Parameter	Value
λ	2
K	20
t	24 hours
Heuristics	H1
Decision rule	R2

Table 5.6: Parameter values used for reference.

5.5.3.2 Evaluation Procedure

These benchmarks were undertaken on the same computer, with a relatively constant load.

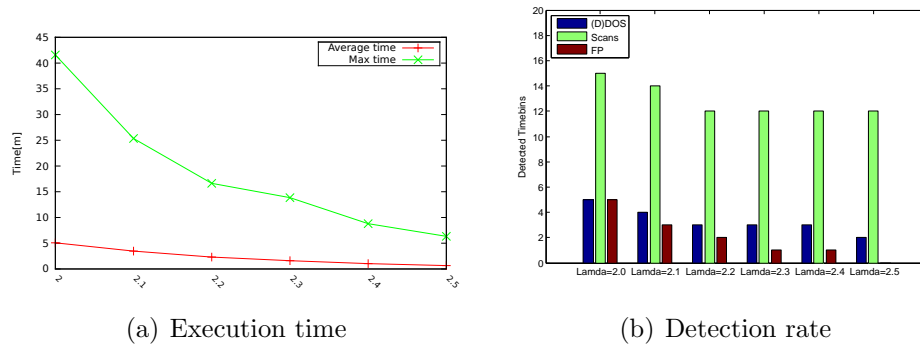
The execution time reported is excluding the selection part of *Senatus*. This is because of two reasons:

1. For properly measuring the selection time, the database would have to be cleared between each execution to ensure equality between the different benchmark conditions. This would make the total execution time too long to have time to do all the benchmark in the limited time that was available for testing.
2. The number of missing time bins stays constant, except for when changing t and K. If tuning t and K showed indications that any parameter value other than the default was superior, we would do further benchmarks of these with measurements of selection time included.

Thus the time from the start to the end of the decision part of *Senatus* is the part that is benchmarked. This will reflect the number of suspicious flows coming from voting, and in the case of the auto-script, how fast an attack is identified as an anomaly.

5.5.4 Tuning λ

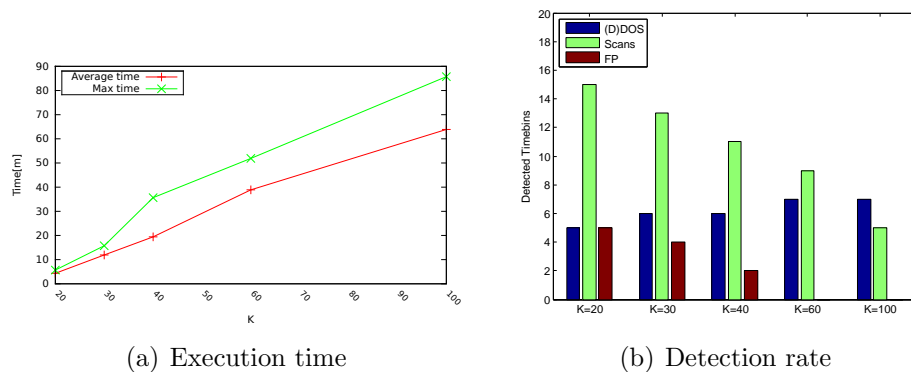
Figure 5.14 shows that increasing λ decreases both the execution time and detection rate. This point is also shown in Section 5.3.2.

Figure 5.14: Performance as a function of λ

5.5.5 Tuning K

Figure 5.15 shows how when K increases, the execution time also increases. The number of (D)DoS detected increases, while the number of scans detected decreases.

The decrease in number of scans can be explained easily; with a small K , there will be a lot of traffic that is barely flagged as anomalous by PCP. When K increases, a lot of this barely flagged traffic will not be flagged anymore, so now the non-flagged senator will not be among the candidates for the decision-part.

Figure 5.15: Performance as a function of K

5.5.6 Tuning t

With a decreased t the execution time is greatly decreased, and the detection rate also decreases.

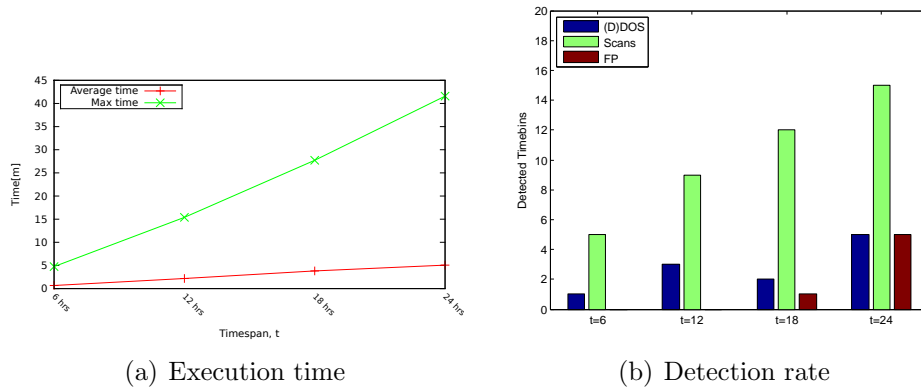


Figure 5.16: Performance as a function of t

5.5.7 Tuning Auto and Decision Rules

In Figure 5.17 we see three different combinations of parameters. The first, labelled R2, uses the default values from Table 5.6, the second adds the automatic root-cause analysis, while the third changes R2 for R1.

We see that for R2 adding the automatic root-cause analysis slightly decreases the execution time, and gives the same detected anomalies. When using R1, the execution time is increased, while the detection rate is improved.

5.5.8 Tuning Heuristics (H1,H2 and α,β)

Figure 5.18 shows execution time and performance for H1 and H2, with H1 using $\alpha \leq 1packet$ and $\alpha \leq 3packets$, and H2 using $\beta \leq 64bpp$ and $\beta \leq 200bpp$. $\alpha \leq 3packets$ and $\beta \leq 64bpp$ are previously the default values as described in Section 5.3.1, but different values were tried in this benchmark.

The figure shows that the default values, 3 packets and 64bpp gives the best execution times. For the default values we see that H2 detects less (D)DoS, more sweeps and have a much lower FP-rate than H1.

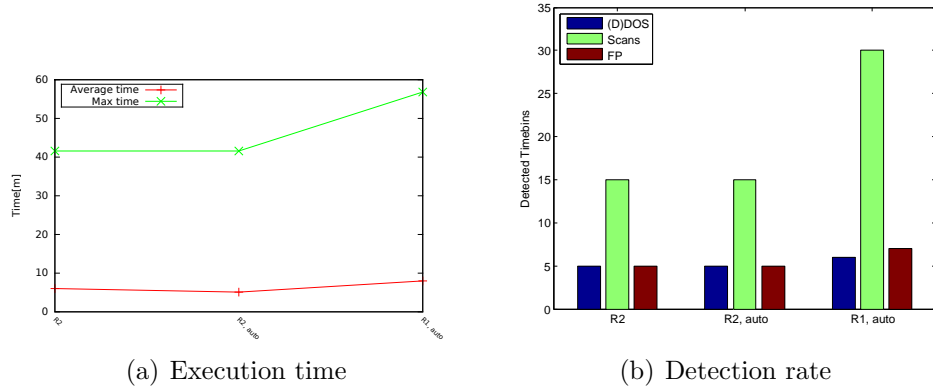


Figure 5.17: Performance as a function of auto and decision rules

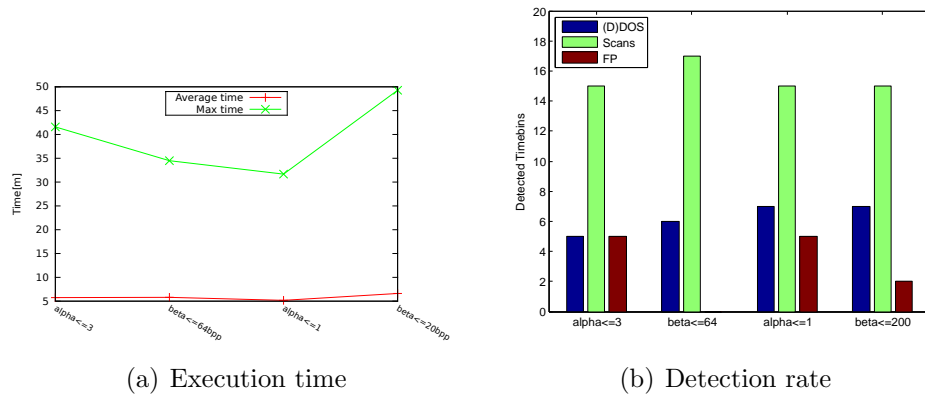


Figure 5.18: Performance as a function of α and of β

Chapter 6

Discussion

This chapter discusses the results from Chapter 5, where we compared Senatus to a similar anomaly detection technique. We also introduce a few commercial anomaly detectors on the market, and compare them to our implementation of Senatus. In addition, we address some of Senatus' limitations along with proposed solutions.

6.1 Evaluation

6.1.1 Detection Methods Comparison

To compare the detection methods we summarize the evaluation results from the previous chapter:

Detection Rate Senatus H1 and Senatus H2 detected 52.96% and 55.92% of the attacks. Even if this is only a little more than half of the attacks in our ground truth, it is still 70-80% more than the detection rate of HBD (30.87%).

False Positive Rate The average FP rate of Senatus H1 and Senatus H2 is the same (16%), which is much lower than HBD that had an average FP rate of 40%.

Type of Attacks Senatus detected significantly more network scans than HBD, especially Senatus with heuristics H2 which detected over 100% more network scans than HBD. On the other hand, HBD tends to detect slightly more DoS/DDoS attacks than Senatus. Senatus with H1 detects almost twice as much DoS/DDoS attacks as Senatus with H2.

The detection rate is probably the most important metric for an anomaly detection method, but it is important to evaluate it combined with the FP rate. If we take this into consideration we see that Senatus perform better than HBD. Which heuristics to use in Senatus depends on the kind of attacks that are most prevalent in a network. In a network where most attacks are network scans, Senatus H2 should be preferred. If detecting Denial of Service attacks is more important, Senatus H1 has the best performance.

6.1.2 Automatic Root-Cause Analysis

From the evaluation of the algorithm we see that finding the right design and threshold values is a very challenging task.

The results show that there is a difference in the intensity of DDoS, DoS and network scan attacks, hence the threshold should be set individually for each attack type. While the optimal values for the thresholds θ_2 and θ_3 were very consistent, respectively around 1400 and 200, the value for θ_1 tended to vary more. The varying values makes it harder for a network administrator to set the right threshold, but from our results a θ_1 value that is between 3-4 times the value of θ_2 seems like a good rule.

Each version of the algorithm has its strength and weaknesses. As mentioned above the thresholds should be set individually and version 1 of the algorithm is using the same value for all thresholds which turned out to be too generic and simplified, and it had a very bad performance compared to the other version. Version 2 of the algorithm only need configuration of two threshold values, but the classification rate was worse than version 1 and 3. Version 1 and 3 had the same classification rates, but version 1 is less complex since it doesn't have the extended first statement. Version 1 of the algorithm seemed to be the best choice in our data set.

An improvement to version 3 to make it better in differentiating between DDoS and DoS, could be to count the number of flows from each source that sent data to the flagged destination IP. Instead of just counting number of source IPs to determine if the attack is a DoS, we can now see how many percent of the traffic that comes from one source. If for example 95% of the traffic comes from the same source, we can classify the attack as a DoS even if there is some benign traffic being sent at the same time. This way we create a buffer that handles "noise".

Even if a classification rate of around 85% is more than acceptable, our algorithm has known weaknesses. The algorithm is not able to detect port scans, and the reason for this is that a port scan has almost the exact same

characteristics as a DoS. Port scans will most likely be marked as DoS attacks by our algorithm.

6.2 Implementation Performance

This section discusses three points from Section 5.5:

- Execution time as a function of the number of average flows.
- Improvement in execution time when introducing databases.
- Choice of default set of parameters for the implementation for the best trade-off between execution time and detection rate.

6.2.1 Execution Time as Function of Number of Average Flows

Figure 5.13 implementation performance evaluation shows that for links with an extra high number of flows, the worst-case execution times may get worse than expected. This point is further discussed in Section 6.5.2.

6.2.2 Improvement in Execution Time from Database Storage

Figure 5.12 visualizes clearly that there is a significant improvement in execution time when senator values are stored in a database.

There are two occasions when there will be a minimal or no improvement. These are when there are few or no values in the database, for example when running Senatus for the first time, or when there is a gap in data to analyze. This can be seen in the first time bin in Figure 5.12.

6.2.3 Choice of Default Parameters

This section discusses each of the parameters tuned in Section 5.5, and explains the choice of default parameter for the implementation.

6.2.3.1 PCP Tuning Parameter λ

Figure 5.14 shows how the execution time drastically decreases as λ is increased. The figure also shows how detection rate decreases, if only slightly.

Section 5.3.2 shows that the decrease in detection rate is much more evident with a larger data set. $\lambda = 2$ remains the default value.

6.2.3.2 K

Figure 5.15 shows that as K increases, both execution time and detection rate increases. The gain in detection rate is not sufficient to compensate for the drastic increase in execution time. $K = 20$ remains the default value.

6.2.3.3 t

Figure 5.16 shows that as t decreases, both execution time and detection rate decreases. The gain in execution time is not sufficient to compensate for the decrease in detection rate. $t = 24$ remains the default value.

6.2.3.4 Auto and Decision Rules

Figure 5.17 shows how execution time varies only slightly between the three cases, whereas both the detection rate and false positive rate increases for R1. The gain in detection rate for automatic root-cause analysis and R1 is sufficient to compensate for the worse execution time, so the default option for the implementation is changed to use these.

6.2.3.5 Heuristics

Figure 5.18 shows the detection rate and execution time of H1 and H2 with different values for α and β .

Section 5.3.1 also evaluates H1 with $\alpha \leq 3\text{packets}$ and H2 with $\beta \leq 64\text{bpp}$, and shows different results than in this section. Because of the limited data set in this section, the detection rate from Section 5.3.1 is used as reference instead.

The results from H1 with $\alpha \leq 1\text{packet}$ are interesting, and show an overall good performance, with a low execution time, and high detection rate.

The results from H2 with $\beta \leq 200\text{bpp}$ are also interesting with a good detection rate, but with the overall worst execution time.

When we see the difference between Section 5.3.1 and this section, it is clear that more evaluation of H1 with $\alpha \leq 1\text{packets}$ and H2 with $\beta \leq 200\text{bpp}$ is needed for validation. H1 with $\alpha \leq 3\text{packets}$ is therefore kept as the default heuristic.

6.3 Geographic Distribution of Anomalies

Figure 6.1 shows the geographical distribution of sources of network scans found in our dataset. China is by far the largest source of the network scans, with 35% of the attacks, with United States as a clear number two with 16%. Network scans are often caused by worm activity, spreading to systems with poorly patched operative systems or lack of updated anti-virus software.

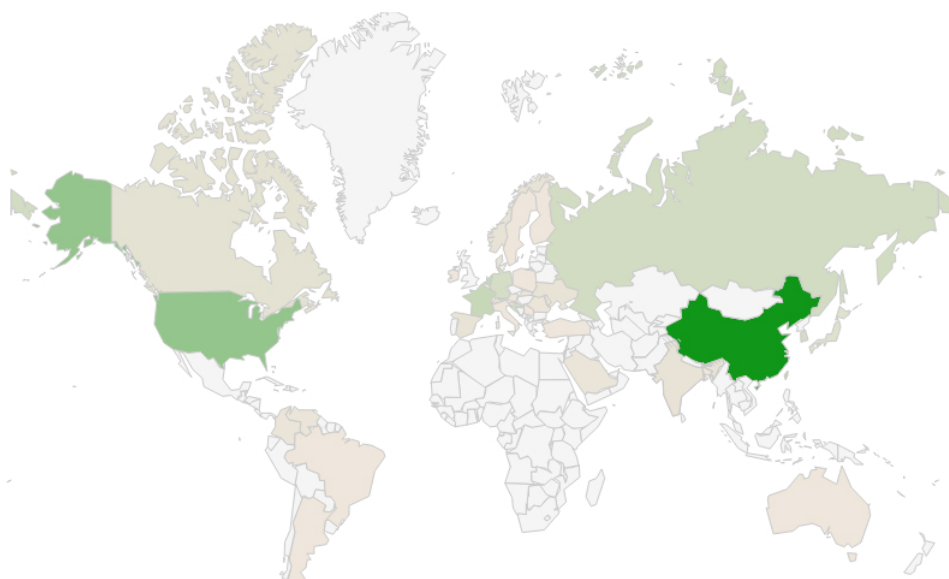


Figure 6.1: Geographic distribution of the sources of network attacks found in our dataset.

Figure 6.2 shows the geographical distribution of the targets of DDoS attacks in our dataset. The distribution of DDoS attacks is more evenly spread, with United States as the most targeted country with 18% of the attacks, and China as a number two with 12%. It is difficult to say something about the motives behind each DDoS attack, but popular DDoS targets like social networks and financial institutions are often headquartered in USA. A few examples of targets of DDoS attacks in our data set are Google, Facebook and Bank of America.

Future work

An interesting study would be to look at the geographic distribution of the sources of the DDoS attacks found in our dataset, since this could possibly reveal the location of large botnets. We also believe there could be a cor-

relation between worm activity and botnets, since worm spreading and the infection phase of a zombie both exploits software vulnerabilities. However, to perform this kind of study it is needed to extract all the participating IP addresses in each attack from the NetFlow data. In our dataset, the largest DDoS attacks had several hundred thousand participants. For each of the participants, we need to translate the IP to a set of coordinates, and the provider we use in the Dashboard only accepts a limited number of requests per hour. Because of time constraints we were not able to perform this study, and we leave this task as possible future work.

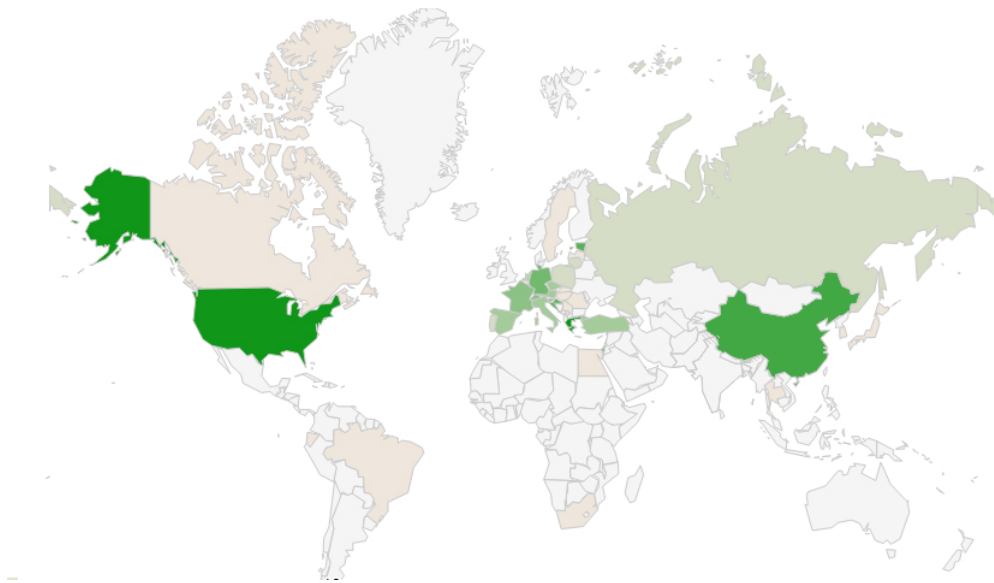


Figure 6.2: Geographic distribution of the targets of DDoS attacks found in our dataset.

6.4 Existing Commercial Products

Although the goal of this work has not been to commercialize Senatus, we believe that the provided functionality, combined with individual customization possibilities, would make this implementation of Senatus a highly useful tool for a network operator.

Anomaly detection is an important task for any network operator, but studies have revealed that many of the current commercial products fail to achieve satisfying performance in real networks [95, 41]. This is not

necessarily due to low detection rates, but mostly because of high false-positive rates, which result in wasting time for a network administrator.

In this section we will introduce a few of the commercial anomaly detection products on the market. Evaluating such tools is a time-consuming task and out of the scope for this thesis, we therefore base this section on the work done by Molina et al. in a recent study on operational experiences with anomaly detectors in backbone networks [41]. This study is particularly interesting because the main requirements Molina et al. set when selecting products in their evaluation is also met by Senatus:

- *Sampled NetFlow support*: Some commercial products require access to packet payloads to detect anomalies. For large network operators with enormous amounts of traffic this would be highly impractical because of the great volumes needed to be stored. As described in Chapter 4, Senatus supports the use of sampled NetFlow data.
- *Non-intrusive collection of data*: Some products require deployment of additional infrastructure in the network to obtain information needed in the anomaly detection process. Senatus do not require any additional installations in the network.
- *Accurate detection and classification*: The product should have a high detection rate combined with a low false-positive rate. Also, the time between the event and the detection should not exceed 30 minutes. Based on the evaluation of Senatus in Chapter 5, we are confident that Senatus can meet both these requirements, although the time between event and detection also depends on the time bin period, which is chosen by the network operator and not Senatus.
- *Collection of evidence related to anomalies*. This includes IP addresses and ports, related to the anomaly. In Senatus, this information is available in the Dashboard.
- *Scalability*: The product need to be scalable, and support a variety of both research traffic and "normal" traffic. In our evaluation of Senatus, we used data from the GEANT2 network which includes these types of traffic, and contains millions of flows per 15 minute time bin. We therefore claim that Senatus has proved that it is highly scalable, and can operate in an environment similar to what Molina et al. requires.

Based on the aforementioned requirements, Molina et.al. selected the three applications *NetReflex*, *StealthWatch* and *Peakflow SP*.

6.4.1 NetReflex

NetReflex from Guavus [96] collects traffic and routing information, and uses this information to perform the following three tasks [41]:

- Automatic topology discovery
- Real-time traffic analysis
- Anomaly detection and classification

Compared to Senatus

Both Senatus and NetReflex give the system administrator access to raw NetFlow-data, if further investigation of an alarm is required. Furthermore, NetReflex is able to display real-time network analysis. Senatus relies on the NetFlow-data to display this information, implying that the delay will depend on the time bin period. Some main differences between NetReflex and Senatus are that NetReflex:

- applies entropy and volume metrics along with PCA (described in section 2.2.1).
- detects anomalies on a fusion of BGP, NetFlow and IS-IS data.
- offer topology analysis

6.4.2 StealthWatch

StealthWatch from Lancope [97] consists of the six parts Management Console, FlowCollector, FlowSensor, FlowSensor VE, IDentity and FlowReplicator. It supports flow data both in NetFlow, IPFIX and sFlow format.

Compared to Senatus

Both StealthWatch and Senatus allow manual tuning of threshold values to control the number of false-positives. Another similarity is that both Senatus and StealthWatch detects changes based on the behavior of addresses and applications (see section 2.2.2), although StealthWatch has more a fine-grained approach by monitoring IP addresses instead of ASes. This can result in a more accurate analysis, but it might also cause scalability

issues [41].

As opposed to Senatus, StealthWatch:

- Requires a learning phase where IP addresses are divided into categories based on type, e.g. SQL server or end-host
- Requires SNMP and BGP data in addition to flow data

6.4.3 Peakflow SP

Peakflow SP from Arbor Networks [98] is the only of the three tools that, in addition to detection, can perform protection by applying countermeasures to block traffic. This is done by internal communication in the network equipment to block the addresses responsible for the malicious activity.

Compared to Senatus

The main difference between Senatus and Peakflow is that Peakflow detects anomalies based on variations in volume metrics. Other differences are that Peakflow:

- Requires SNMP and routing information in addition to flow data
- Gathers information about anomalies from customers to build new signatures

6.5 Limitations

6.5.1 Limitations in the Senatus Framework

As discussed in Section 3.2.2, Senatus may potentially miss certain types of anomalies, e.g. large-flow DDoS attacks, because of the choice of filtering rules. Our performance evaluation indicates this might be the case, where HBD shows slightly better performance in detecting DDoS attacks.

Furthermore, setting the right threshold values in the automatic root-cause analysis algorithm requires a ground truth with labeled attacks. The threshold value is also affected by the sampling rate used when capturing the NetFlow files, since a lower sampling rate will result in lower threshold values; Given a DDoS attack consisting of 1000000 flows and a flow sampling rate of 1/1000, only ~ 1000 of the flows will be captured.

6.5.2 Limitations of the Implementation

Some limitations remain in our implementation of Senatus. This section presents these and suggests solutions.

6.5.2.1 High Maximum Execution Time

The average execution time is good, but if Senatus is given a time bin with a higher than normal amount of flows, the execution time might get higher than expected.

Two possible solutions are threading and distributed computing:

Threading Threading, parallel executions in a program flow, has the potential of significantly improving the run-time performance. The primary focus on where to apply threading would be for the many system calls that are done. If these system calls could be run in parallel and exploit a multiple CPU-core system, this would greatly shorten the execution time.

With boost[68], which is already included as a library, there are capabilities for threading.

Distributed Computing Distributed computing would take the parallelism from threading to a higher level. Several computers would run different parts of the analysis, and the execution time could be even further reduced.

Preparing Senatus for use in a distributed computing environment would mean massive amounts of research and changes to the code.

6.5.2.2 Parameter Checking

Few incoming parameters are checked for validity. This poses security risks such as the possibility of segmentation faults¹.

Segmentation faults leave a vulnerability open for buffer overflow attacks, where an attacker exploits buffer overflows to inject references to the buffer for getting access to a certain property of the system [100].

Introducing parameter checking to Senatus is an easy, but timely task. A list of allowed inputs would have to be created for each parameter. Every input parameter would then have to be checked against this list, and abort

¹When a program attempts to access memory that is not accessible[100]

the execution if the usage of non-allowed parameters is discovered. This would prevent the known segmentation faults.

6.5.2.3 Directory Monitoring Tool

An issue with the directory monitoring tool is that it occasionally will report changes when no changes have been made. This might lead to Senatus being run for time bins that are already analyzed, thus using unnecessary system resources.

We have not been able to verify the reason for this issue to happen, but most likely it is caused by underlying changes in the file system that triggers the directory monitoring tool.

Another solution would be to further develop the daemon mentioned in 4.2.2.1 to handle this task.

Chapter 7

Conclusion

In this thesis we have completed a number of tasks related to the Senatus framework. First of all, we have enhanced the framework by including a root-cause analysis algorithm. Automated root-cause analysis significantly increases the usability of such systems, since it eliminates the time-consuming task of manually inspecting network logs to identify the root-cause.

One of the shortcomings of current anomaly detection techniques is the high ratio of false-positives they produce. Our evaluation showed that Senatus performs significantly better than the similar approach we compared it to in this regard. The evaluation also showed that Senatus has the overall best detection rate. Senatus has a superior detection rate for network scans, but performs slightly worse for detecting Denial of Service attacks.

In addition to the enhancements in the framework and the performance evaluation, we have implemented a high-performance version of Senatus in C++, which includes the automatic root-cause analysis algorithm to provide a complete solution for anomaly detection and root-cause analysis. Our solution is able to analyze a 15-minute time bin and identify the root-cause in just a few minutes. Our implementation also includes a Web Dashboard for Senatus and a corresponding directory monitoring tool, which leverages all of Senatus' features to deliver an online solution for automatic network anomaly detection. The Web Dashboard shows and visualizes information of a detected attack, in addition to providing a number of tools for manual root-cause analysis in the case of a lacking automatic analysis. One of the features of these tools is the possibility to visualize network traffic, which easily can reveal anomalous patterns.

We have also addressed a few limitations of our work, and identified possible

solutions open for future work.

Bibliography

- [1] Anomaly detection in backbone networks: building a security service upon an innovative tool. <http://tnc2010.terena.org/files/TNC%20-%20Anomaly%20Detection%20in%20Backbone%20Networks.Final.ppt>. [PowerPoint Presentation; last visited 06-06-2012].
- [2] Anukool Lakhina, Mark Crovella, and Christophe Diot. Mining anomalies using traffic feature distributions. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '05, pages 217–228, New York, NY, USA, 2005. ACM.
- [3] Guilherme Fernandes and Philippe Owezarski. Automated classification of network traffic anomalies. In Yan Chen, Tassos D. Dimitriou, Jianying Zhou, Ozgur Akan, Paolo Bellavista, Jiannong Cao, Falko Dressler, Domenico Ferrari, Mario Gerla, Hisashi Kobayashi, Sergio Palazzo, Sartaj Sahni, Xuemin (Sherman) Shen, Mircea Stan, Jia Xiaohua, Albert Zomaya, and Geoffrey Coulson, editors, *Security and Privacy in Communication Networks*, volume 19 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 91–100. Springer Berlin Heidelberg, 2009. 10.1007/978-3-642-05284-2_6.
- [4] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing network-wide traffic anomalies. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '04, pages 219–230, New York, NY, USA, 2004. ACM.
- [5] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2):39–53, April 2004.

-
- [6] F. Lau, S.H. Rubin, M.H. Smith, and L. Trajkovic. Distributed denial of service attacks. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 3, pages 2275–2280 vol.3, 2000.
- [7] Jeremy Kirk. European parliament says its website victim of ddos attack. http://www.computerworld.com/s/article/9223740/European_Parliament_says_its_website_victim_of_DDOS_attack, Jan 2012.
- [8] IDG News Robert McMillan. 'anonymous' takes down visa.com in wikileaks protest. http://www.pcworld.com/businesscenter/article/213024/anonymous_takes_down_visacom_in_wikileaks_protest.html, Dec 2010.
- [9] Steve Ragan. Nasdaq and bats web sites fall victim to ddos attacks. <http://www.securityweek.com/nasdaq-and-bats-web-sites-fall-victim-ddos-attacks>, Feb 2012.
- [10] S.Baranowski. Global information assurance certification - security essentials practical - how secure are the root dns servers? Technical report, SANS Institute, 2003.
- [11] J. Lemon. Resisting syn flood dos attacks with a syn cache. *BSDCon 2002 Paper*, 2002.
- [12] S. Kumar. Smurf-based distributed denial of service (ddos) attack amplification in internet. In *Internet Monitoring and Protection, 2007. ICIMP 2007. Second International Conference on*, page 25, july 2007.
- [13] CERT Advicory. Ca-1996-01 udp port denial-of-service attack. Technical report, CERT, 1997.
- [14] C. Leckie and R. Kotagiri. A probabilistic approach to detecting network scans. In *Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP*, pages 359–372, 2002.
- [15] E. Silenok CB Lee, C. Roedel. Detection and characterization of port scan attacks. Technical report, University of California, San Diego. Department of Computer Science, 2003.
- [16] V. Yegneswaran, P. Barford, and J. Ullrich. Internet intrusions: Global characteristics and prevalence. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 138–147. ACM, 2003.

-
- [17] Planet-Lab. "<http://www.planet-lab.org/>".
- [18] Princeton University. Codeen - a content distribution network for planetlab. <http://codeen.cs.princeton.edu>.
- [19] P. Casas, L. Fillatre, S. Vaton, and I. Nikiforov. Volume anomaly detection in data networks: An optimal detection algorithm vs. the pca approach. *Traffic Management and Traffic Engineering for the Future Internet*, pages 96–113, 2009.
- [20] I.T. Jolliffe and MyiLibrary. *Principal component analysis*, volume 2. Wiley Online Library, 2002.
- [21] A. Lakhina, K. Papagiannaki, M. Crovella, C. Diot, E.D. Kolaczyk, and N. Taft. *Structural analysis of network traffic flows*, volume 32. ACM, 2004.
- [22] Anukool Lakhina, Mark Crovella, and Christophe Diot. Characterization of network-wide anomalies in traffic flows. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement, IMC '04*, pages 201–206, New York, NY, USA, 2004. ACM.
- [23] Paul Barford, Jeffery Kline, David Plonka, and Amos Ron. A signal analysis of network traffic anomalies. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement, IMW '02*, pages 71–82, New York, NY, USA, 2002. ACM.
- [24] A. Kind, M.P. Stoecklin, and X. Dimitropoulos. Histogram-based traffic anomaly detection. *Network and Service Management, IEEE Transactions on*, 6(2):110–121, june 2009.
- [25] A. Abdelkefi and Y. Jiang. Compressible traffic features.
- [26] S. Uhlig, B. Quoitin, J. Lepropre, and S. Balon. Providing public intradomain traffic matrices to the research community. *ACM SIGCOMM Computer Communication Review*, 36(1):83–86, 2006.
- [27] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May, and A. Lakhina. Impact of packet sampling on anomaly detection metrics. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 159–164. ACM, 2006.
- [28] George Nychis, Vyas Sekar, David G. Andersen, Hyong Kim, and Hui Zhang. An empirical evaluation of entropy-based traffic anomaly detection. In *Proceedings of the 8th ACM SIGCOMM conference on*

- Internet measurement*, IMC '08, pages 151–156, New York, NY, USA, 2008. ACM.
- [29] K. Xu, Z.L. Zhang, and S. Bhattacharyya. Internet traffic behavior profiling for network security monitoring. *Networking, IEEE/ACM Transactions on*, 16(6):1241–1252, 2008.
- [30] Abdelkefi A., Jiang Y., and Dimitropoulos X. K sparse approximation for traffic histograms dimensionality reduction. Technical report, NTNU, ETH, 2012.
- [31] William Stallings. *Cryptography and network security - principles and practice (3. ed.)*. Prentice Hall, 2003.
- [32] Arno Wagner Daniela Brauckhoff, Xenofontas Dimitropoulos and Kave Salamatian. Anomaly extraction in backbone networks using association rules. Technical report, IEEE/ACM Transactions on Networking, 2009.
- [33] S. Kullback. The kullback-leibler distance. *The American Statistician*, 41(4):340–341, 1987.
- [34] Jochen Hipp, Ulrich Güntzer, and Gholamreza Nakhaeizadeh. Algorithms for association rule mining — a general survey and comparison. *SIGKDD Explor. Newsl.*, 2(1):58–64, June 2000.
- [35] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of 20th International Conference on Very Large Data Bases*, page 487–499, September 1994.
- [36] Klaus Julisch. Clustering intrusion detection alarms to support root cause analysis. *ACM Trans. Inf. Syst. Secur.*, 6(4):443–471, November 2003.
- [37] P. Casas, J. Mazel, and P. Owezarski. Knowledge-independent traffic monitoring: Unsupervised detection of network attacks. *Network, IEEE*, 26(1):13–21, january-february 2012.
- [38] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, September 1999.
- [39] F. Silveira and C. Diot. Urca: Pulling out anomalies by their root causes. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, march 2010.

-
- [40] Atef Abdelkefi. Senatus: Anomaly detection needs democracy? Technical report, In progress.
- [41] Maurizio Molina, Ignasi Paredes-Oliva, Wayne Routly, and Pere Barlet-Ros. Operational experiences with anomaly detection in backbone networks. *Computers & Security*, 31(3):273 – 285, 2012.
- [42] Haakon Ringberg, Augustin Soule, Jennifer Rexford, and Christophe Diot. Sensitivity of pca for traffic anomaly detection. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '07, pages 109–120, New York, NY, USA, 2007. ACM.
- [43] D. Brauckhoff, K. Salamatian, and M. May. Applying pca for traffic anomaly detection: Problems and solutions. In *INFOCOM 2009, IEEE*, pages 2866 –2870, april 2009.
- [44] Abdelkefi A., Jiang Y., Oslebo A., and Kvittem O. Robust traffic anomaly detection with principal component pursuit. In *ACM CONEXT, Student Workshop*, 2010.
- [45] Emmanuel J. Candès, Xiaodong Li, Yi Ma, and John Wright. Robust principal component analysis? *J. ACM*, 58(3):11:1–11:37, June 2011.
- [46] Donoho D.L. Compressed sensing. *IEEE Transactions on Information Theory*, 52:1289–1306, 2006.
- [47] C. Chen, B. He, and X. Yuan. Matrix completion via an alternating direction method. *IMA Journal of Numerical Analysis*, 32(1):227–245, 2012.
- [48] Zuowei Shenx Jian-Feng Cai, Emmanuel J. Candes. A singular value thresholding algorithm for matrix completion. *SIAM Journal of Optimization*, 2010.
- [49] Yi Ma Zhouchen Lin, Minming Chen. The augmented lagrange multiplier method for exact recovery of corrupted low-rank matrices. Technical report, arXiv:1009.5055v2, 2011.
- [50] nghiaho12. Opencv vs. armadillo vs. eigen on linux. <http://nghiaho.com/?p=936>. [Website; last visited 06-05-2012].
- [51] jBlas. jBlas. <http://www.jblas.org>. [Website; last visited 05-10-2012].

-
- [52] Jama. JAMA : A Java Matrix Package. <http://math.nist.gov/javanumerics/jama/>. [Website; last visited 05-10-2012].
- [53] The Colt Project. Colt. <http://acs.lbl.gov/software/colt/>. [Website; last visited 05-10-2012].
- [54] Lapack++. <http://math.nist.gov/lapack++/>. [Website; last visited 05-14-2012].
- [55] ATLAS. Automatically Tuned Linear Algebra Software . <http://math-atlas.sourceforge.net>. [Website; last visited 05-10-2012].
- [56] Blas. Basic Linear Algebra Subprograms. <http://www.netlib.org/blas>. [Website; last visited 05-10-2012].
- [57] C. Sanderson. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, Technical report, NICTA, 2010.
- [58] Armadillo. Armadillo web page. <http://arma.sourceforge.net/>. [Website; last visited 05-14-2012].
- [59] Opencv. <http://opencv.willowgarage.com/wiki/>. [Website; last visited 06-05-2012].
- [60] Eigen. http://eigen.tuxfamily.org/index.php?title=Main_Page. [Website; last visited 06-05-2012].
- [61] R. Hundt. Loop recognition in c++/java/go/scala. *Proceedings of Scala Days*, 2011.
- [62] Sun Microsystems. Memory management in the java hotspot virtual machine. Technical report, 2006.
- [63] nfdump. <http://nfdump.sourceforge.net/>. [Website; last visited 06-05-2012].
- [64] Mark Fullmer. Flow-tools website. <http://www.splintered.net/sw/flow-tools>. [Website; last visited 05-21-2012].
- [65] Cisco. Netflow website. http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html. [Website; last visited 05-21-2012].
- [66] Awk. <http://tldp.org/LDP/abs/html/awk.html>. [Website; last visited 06-05-2012].

- [67] head. <http://www.computerhope.com/unix/uhead.htm>. [Website; last visited 06-05-2012].
- [68] Boost. Boost C++ Libraries. <http://www.boost.org>. [Website; last visited 05-14-2012].
- [69] Inc. Free Software Foundation. The GNU Lesser General Public License, version 3.0. <http://www.opensource.org/licenses/lgpl-3.0.html>. [Website; last visited 05-14-2012].
- [70] SQLite 3. Appropriate Uses For SQLite. <http://www.sqlite.org/whentouse.html>. [Website; last visited 05-21-2012].
- [71] Canonical. Binary package “libsqlite3-dev” in ubuntu oneiric. <https://launchpad.net/ubuntu/oneiric/+package/libsqlite3-dev>. [Website; last visited 05-14-2012].
- [72] Oracle. Connector C++. http://forge.mysql.com/wiki/Connector_C%2B%2B. [Website; last visited 05-14-2012].
- [73] CDT Project. Eclipse CDT. <http://www.eclipse.org/cdt/>. [Website; last visited 05-14-2012].
- [74] The Eclipse Foundation. About the eclipse foundation. <http://www.eclipse.org/org/>. [Website; last visited 06-05-2012].
- [75] Mercurial. Mercurial web page. <http://mercurial.selenic.com/>. [Website; last visited 05-14-2012].
- [76] Bitbucket. Bitbucket web page. <http://bitbucket.org/>. [Website; last visited 05-14-2012].
- [77] Circular dependencies. http://www.schmid.dk/wiki/index.php/C%2B%2B_Circular_Dependency. [Website; last visited 06-05-2012].
- [78] Django Project. Django. <https://www.djangoproject.com/>. [Website; last visited 05-29-2012].
- [79] Play Framework. The Play Framework. <http://www.playframework.org/>. [Website; last visited 05-29-2012].
- [80] The PHP Group. PHP. <http://www.php.net/>. [Website; last visited 05-29-2012].

-
- [81] The Eclipse Foundation. Eclipse PHP Development Tools. <http://www.eclipse.org/projects/project.php?id=tools.pdt>. [Website; last visited 05-29-2012].
- [82] E. Ecma. 262: EcmaScript language specification. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr., 1999.*
- [83] The jQuery Foundation. jQuery. <http://jquery.com/>. [Website; last visited 05-31-2012].
- [84] Mozilla Developer Network. Document Object Model (DOM). <https://developer.mozilla.org/en/DOM>. [Website; last visited 05-31-2012].
- [85] Google. Google Maps API. <https://developers.google.com/maps/>. [Website; last visited 05-31-2012].
- [86] Google. Google Chart Tools. <https://developers.google.com/chart/>. [Website; last visited 05-31-2012].
- [87] Michael Bostock. D3 JS. <http://d3js.org/>. [Website; last visited 05-31-2012].
- [88] ipinfodb.com. IPInfoDB. <http://ipinfodb.com/>. [Website; last visited 05-31-2012].
- [89] R. Love. Kernel korner: Intro to inotify. *Linux Journal*, 2005(139):8, 2005.
- [90] What is a cronjob and how do i use it? <https://service.futurerequest.net/index.php?/Knowledgebase/Article/View/23>.
- [91] The GÉANT project. GÉANT2. <http://www.geant2.net/>. [Website; last visited 05-29-2012].
- [92] Carrie Gates, Carrie Gates, Josh Mcnutt, Joseph B. Kadane, and Marc Kellner. Detecting scans at the isp level. Technical report, 2006.
- [93] Augustin Soule, Kavé Salamatian, and Nina Taft. Combining filtering and statistical methods for anomaly detection. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement, IMC '05*, pages 31–31, Berkeley, CA, USA, 2005. USENIX Association.

-
- [94] Helge Skrautvol Magnus Ask. Anomaly detection and identification in feature based systems: An empirical evaluation. Master's thesis, Norwegian University of Science and Technology, Department of Telematics, 2011.
- [95] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 305–316, may 2010.
- [96] Guavus. NetReflex. <http://www.guavus.com>. [Website; last visited 06-03-2012].
- [97] Lancope. StealthWatch System. <http://www.lancope.com/products/stealthwatch-system>. [Website; last visited 06-03-2012].
- [98] Arbor Networks. Peakflow SP: Traffic Anomaly Detection. <http://arbornetworks.com/peakflow-sp-traffic-anomaly-detection.html>. [Website; last visited 06-03-2012].
- [99] restricting multiple instance of a program. <http://www.linuxquestions.org/questions/programming-9/restricting-multiple-instance-of-a-program-242069/>. [Website; last visited 06-06-2012].
- [100] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

Appendix A

Detailed Description of the Senatus Flow

The following sequence diagrams should give a thorough description of Senatus version 2: Online. The sequence diagrams are split, for readability purposes.

1. Figure A.1(a) describes how time bin data is read from the database. If not in the database, data is fetched with system calls to `flow-tools` or `nfdump`.
2. Figure A.1(b) shows how the senators are pulled from disk.
3. Figure A.2 shows the how the senatorvalues are either fetched from database, or processed in system calls.
4. Figure A.3(a) shows how the voting uses `filterflows` and `inexact_alm_rpca` to catch the abnormal values, for then to combine into suspicious flow-objects.
5. Figure A.3(b) shows the final filtering process to determine whether or not a suspicious flow is indeed an anomalous flow, and in that case they are saved to the database.

It also shows how the automatic root-cause algorithm can be used by choice. If the automatic root-cause algorithm is used, it is applied to each detected anomalous flow. If the root-cause is discovered, the Senatus execution is terminated.

APPENDIX A. DETAILED DESCRIPTION OF THE SENATUS FLOW

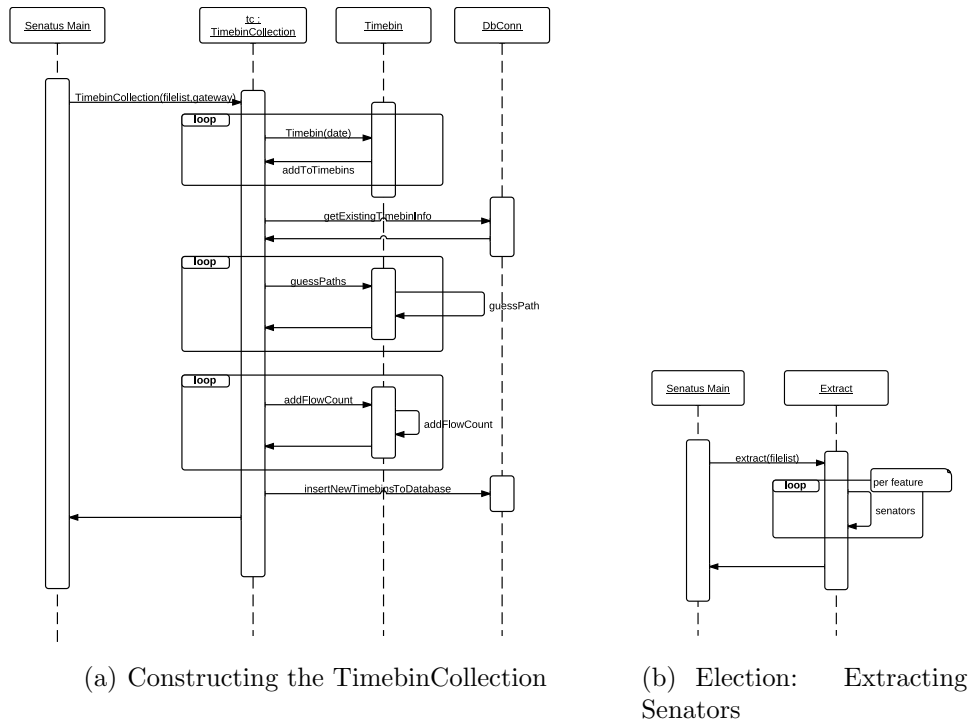


Figure A.1: TimebinCollection and extracting senators

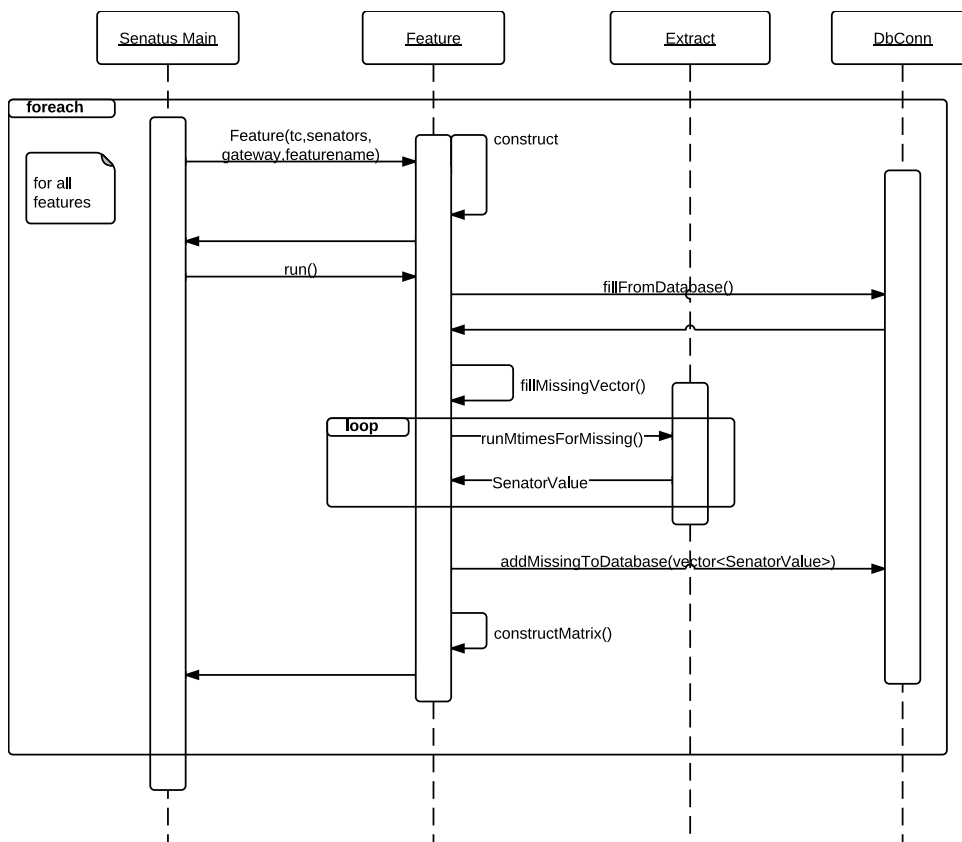
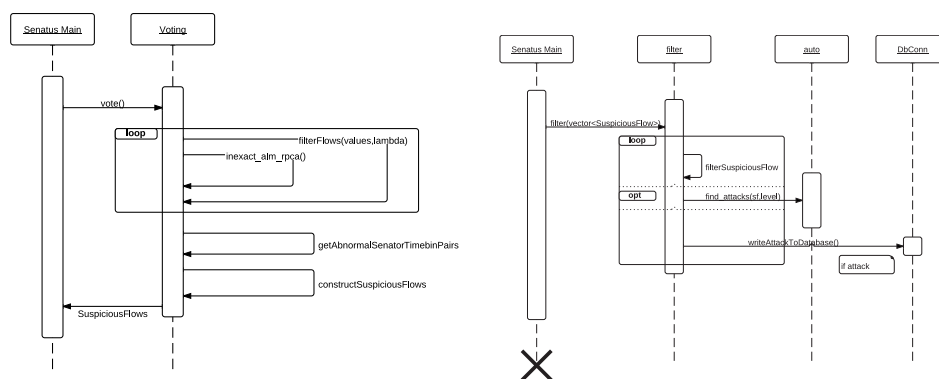


Figure A.2: Election: Getting senator values from database



(a) The voting filters further to get a list of suspicious flows

(b) Decides if attack or not

Figure A.3: Voting and Decision