Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

René Gallati
<ddevel (at) gallati.net>

Near Real-Time Detection of Traffic Usage Rhythm Anomalies in the Backbone

Diploma Thesis DA-2005-03
October 2004 to February 2005

Tutor: Thomas Dübendorfer
Co-Tutor: Arno Wagner
Supervisor: Prof. Bernhard Plattner

Abstract

The usage patterns of certain network protocols (such as the amount of e-mail traffic in bytes per hour) show a clearly visible daily rhythm. In case of an attack, this rhythm is disturbed. We use backbone border router flow-level data in this thesis and apply existing and new algorithms that detect anomaly patterns.

Four algorithms using diverse methods such as peak detection, past data comparison and Holt-Winters forecasting have been implemented in a new anomaly detection system. Using UPFrame [17], an application framework for the processing of UDP data, Internet flow traffic can be monitored and alerts raised, when an anomaly has been detected.

Concentrating on mail traffic with a distinct daily usage pattern, it has been found that there is no single algorithm capable of a good rhythm anomaly detection with a very low false positive rate. Running several algorithms in parallel improves the false positive detection rate at the cost of sensitivity, though the struggle for a low false positive rate remains.

Contents

1 Introduction **4**

2 The Problem **5**
2.1 Our data . . . 5
2.2 The goal . . . 7

3 Survey **9**
3.1 Types . . . 9
3.2 Signature based systems . . . 9
3.3 Statistical analysis . . . 10
3.4 Signal analysis . . . 10
3.5 Combined systems . . . 11

4 Algorithms **12**
4.1 Holt-Winters . . . 12
4.1.1 Description . . . 13
4.1.2 Parameters . . . 14
4.1.3 Limitations . . . 14
4.2 Fuzzy-Algorithm . . . 14
4.2.1 Parameters . . . 15
4.2.2 Scoring . . . 15
4.2.3 Limitations . . . 17
4.2.4 Pseudocode . . . 17
4.3 Lookback-Algorithm . . . 18
4.3.1 Parameters . . . 18
4.3.2 Limitations . . . 18
4.3.3 Pseudocode . . . 19
4.4 MinMax-Algorithm . . . 20
4.4.1 Parameters . . . 20
4.4.2 Limitations . . . 20
4.4.3 Pseudocode . . . 20

5 Implementation **21**
5.1 About 'Horus' . . . 21
5.2 Design . . . 21
5.2.1 Filter and IO . . . 22
5.2.2 Data storage . . . 24
5.2.3 Modules and algorithms . . . 24
5.2.4 Control . . . 25
5.3 Implementation details . . . 26
5.3.1 Modes of operation . . . 26
5.3.2 Data structures . . . 27
5.3.3 Handling of long flows . . . 28
5.3.4 Configuration . . . 29
5.3.5 Extending Horus . . . 29
5.4 Resource usage . . . 30

6 Evaluation **31**

7 Summary and Outlook **35**
7.1 Outlook . . . 35
7.2 Thanks . . . 35

A Parameters **36**

B Configuration **38**

C Using UPFrame **40**

D API **41**
D.1 Structures . . . 41
D.1.1 Config . . . 41
D.1.2 Module . . . 42
D.2 Functions . . . 42
D.2.1 Blocklist . . . 42
D.2.2 Config . . . 43
D.2.3 Elog . . . 43
D.2.4 Memory . . . 43
D.2.5 RRDlib . . . 43

E File formats **45**
E.1 NetFlow File format . . . 45
E.2 Intermediate File format . . . 45

F Original problem description **46**
F.1 Introduction . . . 46
F.2 The Task . . . 46
F.3 Deliverables . . . 47

List of Figures

1 Detailed traffic flow data . . . 6
2 12 hours of SMTP traffic data, raw . . . 7
3 12 hours of SMTP traffic data, averaged . . . 7
4 SMTP traffic: precise and averaged during mydoom outbreak . . . 12
5 Symbol of Horus . . . 21
6 Overview of the Horus program . . . 22
7 Flow of data through the Horus program . . . 23
8 Structure of the bucket-list . . . 27
9 Plot of the Sobig.F worm outbreak . . . 31
10 Output from Holt-Winters of Sobig.F outbreak . . . 31
11 Output from Fuzzy algorithm during Sobig.F outbreak . . . 31
12 Output from Lookback algorithm during Sobig.F outbreak . . . 32
13 Plot of one week normal traffic . . . 32
14 Output from Holt-Winters of one week normal traffic . . . 32
15 Output from Fuzzy of one week normal traffic . . . 32
16 Output from Lookback of one week normal traffic . . . 33
17 Plot of mydoom outbreak with incomplete data from only one router . . . 33
18 Plot of Holt-Winters morning alerts . . . 34

List of Tables

1 Linear scoring scale in Fuzzy algorithm . . . 16
2 Exponential scoring scale in Fuzzy algorithm . . . 17
3 Defined data sources in the round robin database files . . . 27
4 Distribution of long flows according to size . . . 29
5 All available command line arguments in Horus . . . 37
6 Configuration directives . . . 39
7 Configuration structure . . . 41
8 Module structure . . . 42

1 Introduction

The importance of the Internet has increased tremendously over the last few years. Businesses whose market and customers are entirely Internet based, information and distribution channels that cannot work without the Internet and a multitude of services and technologies that require the universal communication between systems over the Internet are widespread today. This has effectively made an Internet connection as important as electricity, water and the telephone system in the industrial countries. It is expected that the telephone system and the Internet will converge in the next few years, with the telcos slowly looking into replacing their proprietary networks by IP-capable technology [1].

With the advent of new services like VoIP (Voice over IP, transmitting voice telephony services over an IP network) that finally get increasingly deployed or companies using VPN (Virtual private networks) technology to connect their business units and partners over the Internet instead of buying traditional leased lines, the dependency on a working Internet connectivity has risen dramatically. Depending on the use or business, an interruption of the Internet connectivity can have severe consequences and cause many problems.

Because of the increasing dependence on the Internet, it has become important to ensure flawless operation and protect the network from attacks. Unfortunately, the growth and importance of the Internet also draws malicious users to the network. Furthermore bugs and errors in software used in the systems connected to the Internet now have far more consequences, since the systems are no longer isolated and thus code written to exploit such bugs can spread very fast throughout the network, not limited anymore by company, department or country borders. A prime example is the SQL.Slammer worm, which managed to infect an estimated 90% of all vulnerable systems in merely 10 minutes, worldwide [2]. Since the worm with its brute-force flood property saturated many links, the effects on the global Internet was very visible. Entire backbone networks connecting many thousand customers became hardly reachable or disappeared entirely, causing further problems.

DDoS flooding attacks are capable of knocking out entire sites by over-saturating the network link with millions of requests or by exhausting all server resources with too many fake requests. While DDoS attacks are known for a long time, effective countermeasures are scarce. With the number of bot-nets [3] increasing steadily, the potential damage and number of people capable of mounting a successful DDoS attack is further rising. ”Owners” of bot-nets also started to sell off access to their army of compromised systems for spamming purposes or blackmailing [4] businesses by threatening to launch a DDoS attack and thereby preventing them from conducting their business on the Internet.

Due to the general increase in bandwidth available to end-users through CATV and DSL broadband technologies, not only the amount of often badly maintained systems that are easily subverted increases, but also the available upstream bandwidth required for DDoS attacks to be effective. The current situation is such that even big sites with a large amount of resources can be knocked out by a large DDoS attack with hundreds to thousands of participating systems with each only having a fraction of the resources of the victim.

In the light of these developments and new threats, it becomes clear that effective tools for reliable detection need to be readily available in order to allow network operators to detect and react to such events. The goal of this thesis is to provide one such tool that will enable network operators to be notified of unusual events in the network and hence decrease the time to deploy countermeasures.

2 The Problem

The first-most problem in anomaly detection lies in the definition of anomaly itself. Anomalous behaviour can often be easily spotted by a human observer, given the right visual representation of the underlying data. Defining the anomaly in a formal way, however, is not easy. Consider for example the sudden change of traffic on a network link by 20 megabits/sec. While such an event certainly stands out on any graph displaying the link load if the links capacity is of similar magnitude (like an E3 link with 34 mbps capacity), it will probably be easily overlooked on a graph displaying the load of a high-traffic link such as an STM-4 (622 mbps) where the change is simply dwarfed by the normal variance of the link traffic.

Thus the "I know one when I see one" human approach to anomaly detection can not be ported to an automatic system. The next best step from observing absolute changes is to put them into relation to the magnitude of the value. This will subsequently fail to detect continuous degradation if the change is not too abrupt. A cascading effect such as a spreading worm that shuts down the infected machines after a short phase of reproduction might slowly kill all normally traffic producing machines, resulting in a continuous decrease of traffic on the link(s).

What we therefore desire is a system or algorithm that continuously monitors the input data and over time learns to distinguish the normal rhythms from disruptions and anomalies. It should further realize that there is less variation during the night hours than during rush-time and consequently be more sensitive to sudden changes in the night than during the day.

Another problem in anomaly detection is the fact that while anomalies may be detected, the systems that do so usually cannot really tell why exactly they sent an alarm. This is not a big problem in itself, since the system's purpose is to notify a human operator to take a closer look at things. But if such relative vague alerts turn out to be false or unfounded too often or if the system just produces too many alarms, it often is quickly considered useless and turned off or ignored.

Furthermore, DDoS attacks and widespread worm infections don't come and go unnoticed. Outages caused by an attack get reported to the ISP at some time, after the cause has been detected. Having a system in place that is capable to detect such attacks early can be a huge competitive advantage. By enabling the operator to detect and mitigate the problem early allows him to prevent the damage or outage that otherwise may occur, both to his equipment and assets as well as those of his customers. In an environment where critical services are provided, it is important to detect and react to attacks as early as possible.

Finally, the system should be easy to set up and maintain, meaning the cost both in terms of money and time should be non prohibitive.

2.1 Our data

Network operators are generally concerned with parameters such as link load and capacity in order to be able to schedule maintenance and upgrades. This data is readily available on all routers as simple counters. It can be easily gathered and stored for long periods. The downside is that this data is already heavily summarized. In order to be able to better detect certain anomalies we gather data from border routers as NetFlow data records [15]. That means we collect information about the individual flows which enables us to see attacks that may be suppressed in the overall data flow of big links, but show distinctively when one considers only certain classes of traffic. We are especially interested in mail traffic (comprising of traffic from or to port 25/TCP) in order to catch mail worm activity.

Having such detailed data comes at a price, however. The amount of data to process and store can be very huge, depending on the capacity and traffic of the links that are under observation. As a point of reference, one hour of traffic at one border router carrying 300 mbit/sec of traffic on average generates between 200 and 550 MBytes of compressed (using bzip2) Cisco NetFlow v5 data. The latter amounts to 1.6 GBytes in uncompressed flow records (per hour). These values are not set in stone of course, and will vary widely depending on the network topology,

link capacities and usage levels. But they can be taken as a first estimation of the amount of data to process. In order to collect, process and store this data from several routers, UPFrame [17] is used. UPFrame has previously been developed for the DDoSVax team for this purpose. Storing this data longterm needs an infrastructure capable to deal with this amount of data. For detection of anomalies only in near-real time, one or two computers with current standard hardware are sufficient.

The NetFlow data is being generated on the routers and sent to our collector in the form of UDP packets. A router sends one packet whenever enough flow records have been generated to fill a packet or after a configurable timeout. For TCP connections which comprise the majority of data traffic, the record can only be exported after the connection has been shut down. The router will export a flow record after it has seen the FIN-ACK sequence, signalling the termination of the connection. Unfortunately, this may never happen if one end of the connection is disconnected from the network or crashes. Furthermore, there can be long flows in which the TCP connections are established for hours or even days. In order to not consume too much memory on the routers and to report such long established flows in a timely manner, they are reported after a configurable amount of time and immediately registered as a new flow again. This essentially treats a long flow as several short ones.

While this ensures that data about flows arrives in a timely, albeit somewhat incomplete manner, full real time operation is not possible when information about long flows is held back several minutes in this manner. As we show in Section 5.3.3, only a minor percentage of the flows in the data we received were long flows and of those very few had a significant size.

Since we only create NetFlow records at the border, data about internal traffic not leaving the network is unavailable to us. This reduces the amount of data, but not the usefulness. Large DDoS attacks usually involve hundreds if not thousands of machines distributed around the world and worm outbreaks occur uncoordinated in a pseudo or completely random fashion. This automatically leads to traffic coming in or leaving our borders and thus getting detected. Another side effect is that the necessity of maintaining a graph of the topology of our network falls away. We are mainly concerned with the interaction of our network with the Internet, assuming that internal problems are dealt with in a different way.

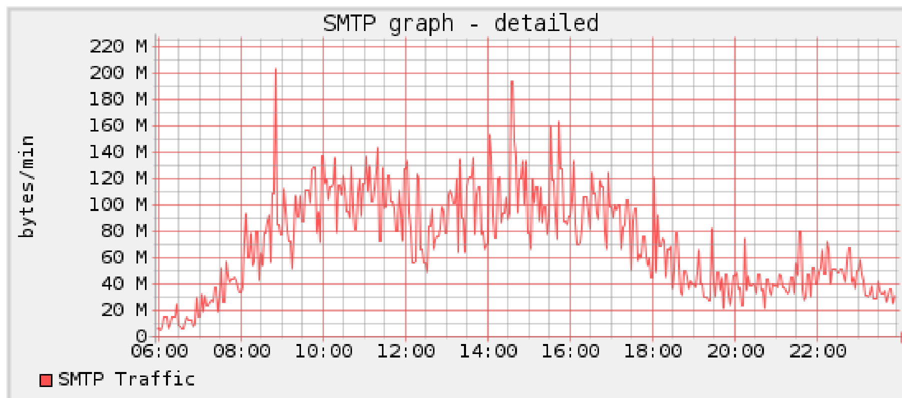

Figure 1: Detailed traffic flow data

The collected aggregated data from our four border-routers display the bursty behaviour common in network traffic flows, as can be seen in Fig 1. Note that the interval between points is one minute. Averaging over the last minutes removes the somewhat erratic spikes and smoothens the graph at the cost of some information loss. Figures 2 and 3 show 12 hours of mail traffic data in unprocessed one minute intervals and as averaged over the last 3 minutes respectively.

As can be seen in Fig 2, the spikes can easily deviate from the previous data point by 50% or more. This fact necessitates the use of algorithms that process such data to employ some sort of smoothing or weighting as not to overvalue these bursts. On the other hand, a large spike could be

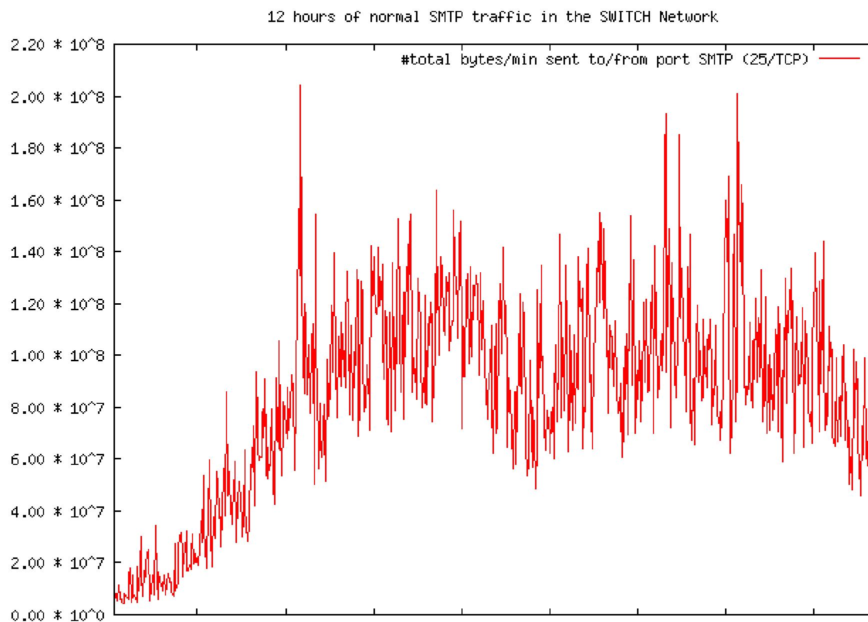

Figure 2: 12 hours of SMTP traffic data, raw

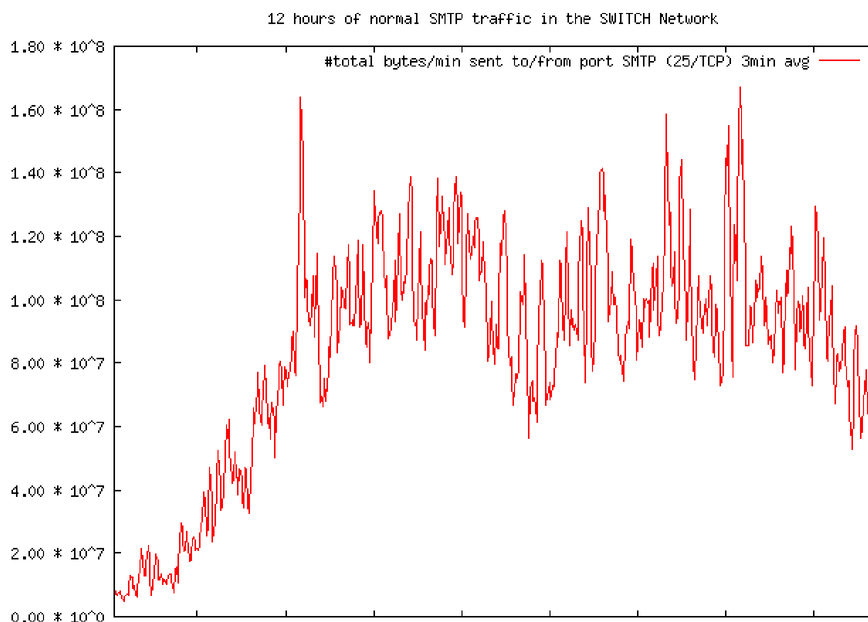

Figure 3: 12 hours of SMTP traffic data, averaged

a hint for a single machine that was used to send a lot of mail messages to many recipients within a short time frame, such as a mailing-list server or a spammer. Smoothing out such bursts in order to obtain a more usable graph can hide such information, thereby decreasing the efficiency of the system as a whole to detect anomalies.

2.2 The goal

Based upon the flow level data as described in 2.1, the goal of this thesis is the design and development of a Traffic Anomaly Detector. This system should be able to detect rhythm anomalies and raise an alert so that countermeasures can quickly be employed. In order to be useful, its false alarm rate must be as low as possible and deployment should be easy. Specifically, it should be possible to use it without complex configuration or the need to make the network

topology known to the system at any one time.

We are concentrating our efforts on mail traffic, because the time and date of email worm outbreaks is well known and archived NetFlow data of these events are readily available. Furthermore, mail traffic shows a very distinct daily pattern toward which we will try to train our algorithms to. It should also be mentioned that mail and web traffic have somewhat similar usage patterns which should allow the algorithms trained and validated against mail traffic anomalies to be used against web traffic anomalies also without the need of painstakingly detailed analysis of web traffic.
In Section 3 we present a survey of the field of Anomaly Detection. Later in Section 4 we describe the algorithms used in the implementation in some detail. Section 5 takes a closer look at the implementation of our Anomaly Detection System and in Section 6 we evaluate the results achieved.

3 Survey

A survey of existing systems trying to achieve the same goal as this thesis was conducted. About every ISP (Internet Service Provider) worth its name is monitoring his equipment for failures and rate/throughput as well as data traffic in their network. Yet there are very few documents and papers about deployment or operation of tools that monitor flow data and report anomalies not using hard coded thresholds like activity on well-known worm-ports.

As mentioned in Barford et al. [7], most ISPs seem to simply have deployed their own monitoring system, based on router throughput and server usage statistics and are raising alarms when certain thresholds are reached or merely visualize data and identify anomalies by recognizing patterns from past experience. Learning systems that adapt to a perceived level of "normality" and report abnormal behaviour seem to be either not widely in operation or not documented.

Nevertheless one can find a small number of interesting research activities in this field. A helpful guide covering some projects in Anomaly Detection and measurement is Neil Patwari's Anomaly Detection Resource Page [8].

3.1 Types

We classify anomaly detection into the following types:

- Signature based systems: A system that has a list of well defined signatures and raises an alarm if the signature pattern occurs. This can range from an exceeded threshold of an arbitrarily monitored device variable to an infected host contacting some well known master server on a specific port. Signature based systems are often static, that is to say, can neither detect new anomalies even if they show very clearly in the data nor can they detect unexpected variations of known anomalies (worm opens backdoor on another port) until they are updated with new signatures.
- Statistical analysis: A system that performs statistical analysis over the dataset to calculate predictions and expected behaviour and then reports deviations from the estimated result. Such systems are generally able to adapt to slowly changing environments.
- Signal analysis: A relatively new method that decomposes the data into frequency components by using signal analysis techniques and subsequently applies a statistic analysis over these components. Strictly speaking, signal analysis is a specialized case of statistical analysis.
- Combined systems: A system that combines some or all of the above types in order to reduce false positives of one method or enhance detection rate due to deficiencies or slow detection in one method.

3.2 Signature based systems

Signature based systems are the oldest (see [18]) and most widely deployed systems for monitoring networks. Those that only read counter values out of network equipment and alert on threshold violations, are simple to setup, operate and maintain. However, the drawback of such a monitoring system is its limited ability to detect DDoS or worm attacks unless those are of such a magnitude that the pre-set thresholds are exceeded. Attack traffic that is too low to dominate for example a monitored backbone link can however be more than enough to completely knock out a small customer site with limited bandwidth capacity and going undetected until the network operator gets notified by the victim.

While signature based systems are well suited for observing global values such as link saturation, they are generally useless to detect anomalies such as traffic levels falling down to night-levels during peak time due to the failure or attack since that is neither exceeding a maximum threshold, nor is the traffic completely down or sinks below another warning threshold. One could amend such a rule with expected traffic low and high-values during day and a different set during nighttime, but that would lead only to a huge set of static rules to maintain for every

change of the network, which quickly becomes cumbersome and unmanageable.

Examples of signature based systems are traditional Intrusion Detection Systems (IDS for short) like SNORT [9] or BRO [10], but also PHAD [11] as used in the combined system described in Mahoney [12].

Of course, reactive inspection from stored NetFlow [15] data after an incident as a SecurityFocus[20] article describes is also a possibility.

3.3 Statistical analysis

An example of statistical anomaly detection is NETAD [13], which observes events and assigns probabilities when the next event of the same type is to be expected. This produces a ”score” and the system raises an alarm as soon as this score exceeds a threshold. The NETAD system needs to be trained in an attack-free environment, which makes it somewhat less useful than other approaches, since the simulation of a backbone network is not trivial. Alternatively, it can be left running in training mode, which does however decrease the amount of attacks detected.

Another approach by Brutlag [6] uses a variant of the Holt-Winters [19] algorithm that decomposes the data into a baseline, trend and seasonal variance. This approach is used successfully at Microsoft’s WebTV [14] to monitor thousands of devices for aberrant behaviour. As this method is also employed in this thesis, a more detailed insight into the algorithm is provided in Section 4.1.

In Lakhina [21] principal component analysis on origin-destination (OD) flows is being used to analyze the structure of network traffic flows. The result of this work leads to a system that is able to decompose traffic levels gained from the links in a network into OD-flows and detect volume anomalies (a sudden positive or negative change in an OD-flow’s traffic) with a high detection and a small false alarm rate. Principal Component Analysis is used to separate the space of link level traffic measurements into subspaces which are then distinguished into normal and anomalous traffic behaviour using a separation step. Finally, anomalies are detected by projecting traffic of a given link onto these subspaces, thereby revealing quite clearly distinguishable anomalies which are then reported (after an additional quantification step the system performs). While not specifically mentioned in the paper, we assume that the method outlined requires a huge amount of processing power since the calculations take place over large nxn matrices. It is unclear whether their method can be directly applied to our ”border router only” backbone traffic without having traffic data of any intermediate router nodes.

Statistical analysis allows a system to react to unknown and not prespecified events by predicting the immediate expected future compared against the real measured data. This is a feature that signature based systems cannot perform. The major drawback is the loss of concise reports to the user. The system may detect an anomaly but is not able to tell what exactly the anomaly is or why the alarm has been raised. Nevertheless, statistical anomaly detection has been widely used and has proven quite effective in the test scenarios.

3.4 Signal analysis

Signal analysis is strictly speaking also a statistical method using special wavelets. The approach is relatively new and is looking to become a promising tool for anomaly detection. A special class of wavelets, called graph-wavelets are applied over the spatial domain of the measurement in the paper of Crovella and Kolaczyk [23] in contrast to the time domain which is used in almost every other system. The wavelet transformation can decompose the measurement into high, mid, and low frequency components, which represent the bursty, trend and seasonal components of the original signal. Anomalies can then easily be detected since the seasonal, normal traffic does not show the anomaly while it clearly shows in the high and mid frequency ”bands”. The prototype system as described by Crovella and Kolaczyk [23] is built to detect traffic shifts due to link failures and anomalies in a network by collecting router statistics from every link. It furthermore needs complete knowledge of the network graph it works on, which makes it not ready for everyday usage. Another implementation of the signal analysis approach is the IMAPIT system developed and described in Barford [7]. While the IMAPIT system is not real-time capable due to the

use of a three hour sliding-window for a weighted calculation in the employed deviation score mechanism, it does not need to know the network graph but works with traffic data from a single router and shows a high detection rate as well as a very low false-positive rate. Unfortunately, while both wavelet approaches certainly look promising, they need to be carefully tuned to be effective.

3.5 Combined systems

To decrease the amount of false alarms or to achieve higher confidence a combination of the methods outlined above or consideration of different input sensor is suggested. Roughan et al. [24] use a combination of different sensor inputs, BGP route update counts and SNMP traffic counters to increase their detection rate.

4 Algorithms

Our traffic data is very bursty with many spikes (noise) but nevertheless shows a very distinct daily rhythm pattern with a sag during the midday hour. Saturdays and Sundays can easily be seen due to reduced use of the Internet over the weekends.

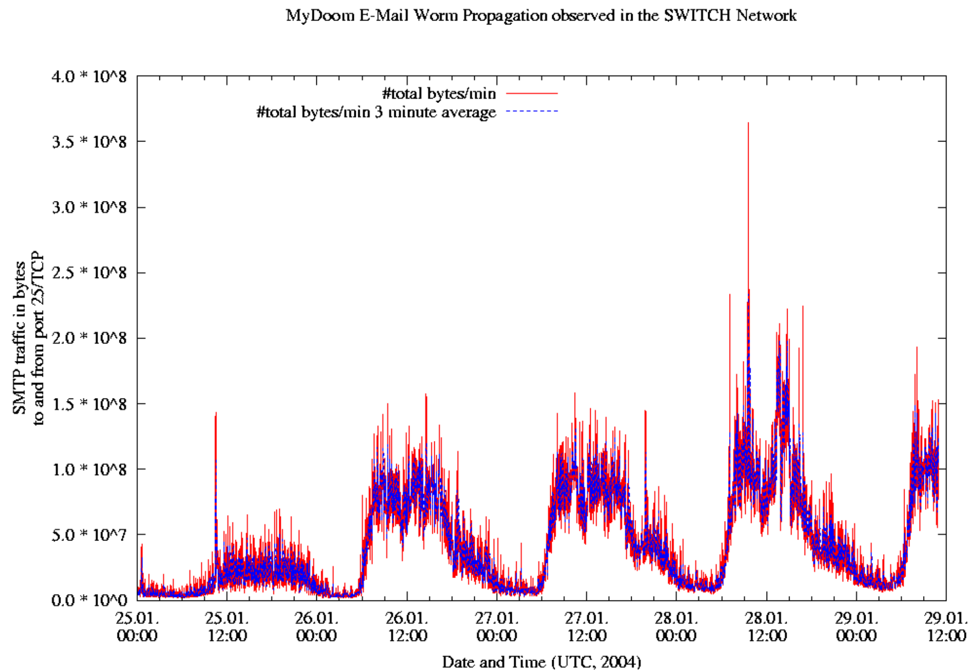

Figure 4: SMTP traffic: precise and averaged during mydoom outbreak

Network traffic not only shows daily rhythms but is generally also showing higher order seasonal effects such as different usage patterns in summer than in winter months and between normal workdays and special holidays where a large part of the population is not at work. Additionally, network systems tend to be growing with more users connecting to the network or upgraded connection speeds due to the proliferation of affordable broadband connectivity at home and increased importance of the Internet. Furthermore, network providers need to regularly change and adapt the network in order to accommodate new users, increase fault tolerance or to be able to provide new services with higher bandwidth demands. This causes the data to be non-stationary.

Because of these constant changes in the usage patterns and in the capability of the network, it is desirable that algorithms working on such data are free of fixed limits and thresholds in order that they can adapt to a live, evolving network but still be able to detect anomalies and rhythm disturbances.

4.1 Holt-Winters

In 4.1.1 we take a closer look at the Holt-Winters [19] method as implemented in the development versions of the RRDTool [16] by Jake Brutlag [6] which is used in Horus, our implementation of a network traffic anomaly detector. The implementation of Horus is described in Section 5.

Later in Section 4.2, we present one algorithm designed and implemented for this thesis, which employs a somewhat fuzzy-logic approach to score the deviation from the level of normality perceived. Following in Section 4.3 is the Loopback-Algorithm, which calculates the median over the values gathered some days before and compares them to the actual observation, taking into account the variability of the deviations. Lastly, in Section 4.4 we are explaining the MinMax-Algorithm from Thomas Dübendorfer and show its application in this thesis.

4.1.1 Description

The Holt-Winters method is an exponential smoothing forecasting algorithm that is built upon the premise that the observed time series is composed from three distinct components: A baseline, a linear trend and a seasonal effect. The seasonal effect allows to take different behaviour at regular intervals into account, such as a sag during meal times or less variation in usage during nighttime. The Holt-Winters method furthermore assumes that these components change over time and adapts by applying exponential smoothing [25] over the data. A prediction based on the three components is then being made together with a confidence band around the prediction. The prediction is derived from the seasonal variability of the input data encountered. Anomalies are then marked as any reading where the observed value exceeds the confidence interval. This makes the algorithm an ideal candidate for our cause.

The Holt-Winters algorithm as used in RRDTool predicts one single[1] value based upon the sum of the three components as follows:

$$\hat{y}_{t+1} = a_t + b_t + c_{t+1-m}$$

where m is the period of the seasonal trend, in our case a day expressed in the number of observations. The three components a, the baseline, b the linear trend and the seasonal trend c are updated with:

$$\begin{aligned} a_t &= \alpha(y_t - c_{t-m}) + (1-\alpha)(a_{t-1} + b_{t-1}) \\ b_t &= \beta(a_t - a_{t-1}) + (1-\beta)b_{t-1} \\ c_t &= \gamma(y_t - a_t) + (1-\gamma)c_{t-m} \end{aligned}$$

The new baseline is the observed value adjusted by the last seasonal coefficient and the predicted slope is added to account for the change due to the linear trend. The new trend or slope is simply the difference between the new and old baseline. The new estimate of the seasonal component finally is the difference of the observed value and the corresponding baseline while considering the value one season ago.

α, β and γ are the three adaptation parameters of the value with $0 < \alpha, \beta, \gamma < 1$. These parameters determine how much weight newer values have over older predictions. A large value lets the parameter put more weight on recent observations, in other words lets the algorithm adapt to changes more quickly while smaller values put more weight on the past history of the time series.

Comparing the prediction $\hat{y}_t$ to the observed value y_t, the confidence bands measure the deviation for each point. The Holt-Winters algorithm in RRDTool takes seasonal variability into account which is to say that the average deviation can be different during a seasonal cycle. The formula to calculate the confidence band including the expected deviation at time t is:

$$d_t = \gamma|y_t - \hat{y}_t| + (1-\gamma)d_{t-m}$$
$$(\hat{y}_t - \delta_- \cdot d_{t-m}, \hat{y}_t + \delta_+ \cdot d_{t-m}) \tag{4.1}$$

Where d_t is the predicted deviation at time t and the confidence band thus the interval from $\hat{y}_t - \delta_- * d_{t-m}$ to $\hat{y}_t + \delta_+ * d_{t-m}$ with δ_+ and δ_- as scaling factors for the width of the confidence band. Since a symmetric band is desired, we set $\delta_+ = \delta_-$ and denote it in the future as parameter δ.

If an observed value falls outside the confidence band interval from formula (4.1) an anomaly is detected. Since short bursts or values outside the prediction occur quite frequently, abnormal behaviour is only reported if several such violations occur within a specified time frame. The

[1] The Holt-Winters algorithm is able to predict more than one future value

size of this sliding window and the alert threshold can be freely chosen.

More details about the implementation of the Holt-Winters algorithm in RRDTool can be found in Brutlag [6].

4.1.2 Parameters

As seen in 4.1.1, there are several model parameters for the algorithm. Choosing the right parameters is vital in order to achieve the desired result of a low amount of false alarms yet early detection of rhythm anomalies.

The following formula can be used to calculate the parameters:

$$\alpha = 1 - \exp\left(\frac{\ln(1 - \text{total weight as }\%)}{\#\text{ of time points}}\right) \tag{4.2}$$

For example if you want the last 15 minutes with an observation every minute to account for 80% of the weight then the formula gives $\alpha = 0.1$. By simple rearrangment of the formula to

$$\text{total weight as }\% = 1 - \exp(\ln(1 - \alpha) \cdot \#\text{ of time points}) \tag{4.3}$$

it becomes easy to get the weight of a certain time span at a given value of α. For example, with $\alpha = 0.001$, 4 hours of values with a sample rate of 1 observation per minute (240 values) gives a weight of 21.3%. The same timespan using a sample rate of 1 observation per 5 minutes (48 values) gives a weight of 4.7%.

Although the formulae use α as parameter, they apply to β and γ in the same way. If the last day at 1 observation per minute (1440 values) should contribute only 25% weight you will get a parameter value of 0.00019976 by using formula (4.2).

At least one of the parameters α, β and γ should allow adaptation in a short time frame. Since β tries to capture a slowly changing trend and updates on γ only happen infrequently once per seasonal cycle for each coefficient, the recommended choice is α.
The initialization of the coefficients of the parameters is bootstrapped from the first observed values. In case of γ, the seasonal component, one complete seasonal cycle needs to be encountered first. The effect of the initialization diminishes with more observed values as newer values with their higher weight suppress older ones.
In the RRDTool implementation two more parameters become available. Per definition every single observed value that oversteps the confidence band around the predicted value would raise an alert. Our input data, however, is too bursty and short spikes do occur quite frequently. To mitigate this problem an alert is only reported when there are more than a specified amount of such violations within a time window. The two parameters set the threshold for the number of alerts required and the size of the window and default to 7 and 9.

4.1.3 Limitations

A limitation of the Holt-Winters method is the availability of only one seasonal component. Since the daily rhythm is most prominent it does not make sense to use another rhythm for γ. Higher order seasonal effects such as weekly, monthly or quarterly seasons cannot be added and the parameters must therefore be chosen in a way to allow these rhythms to be absorbed in α and β.

4.2 Fuzzy-Algorithm

We designed and implemented a new anomaly detection algorithm that we called "Fuzzy"The basic idea of the Fuzzy algorithm is that the observed value at some time t is supposed to be similar to the value observed at time $t_{-n*timeperiod}$. Thus the Fuzzy algorithm looks back a certain period (days in our case) and compares each value found with the current observed value. The difference of the values is then calculated and a score depending on the relative deviation assigned. Higher deviations result in higher score values. These score values over

the past days are summed up and compared against a threshold. If the score is exceeding the threshold, an alert status is set for the current time. This method allows for some "natural deviation" by giving low scores for low deviations and high scores for large deviations. Because of the somewhat arbitrary distinction into "score buckets", we call this algorithm Fuzzy.

Since our traffic data is bursty, this would produce an alert anytime a spike is observed. In order to work in the presence of such spikes, the Fuzzy algorithm does not work on a single value at a time t, but on the average of the last three values (i.e. $(t, t_{-1}, t_{-2})/3$). This averages out small spikes and reduces the amount of alerts tremendously that would otherwise occur often in such a noisy signal. To reduce the false positive rate further, an alert is only reported if there has been a certain number of alerts within a moving time span window.

Our SMTP data differs significantly between workdays and weekends. On weekends, the traffic can be a third or less of the volume of a normal workday. This caused the first weekend unaware Fuzzy algorithm to always raise alerts on weekends. The problem has been resolved by comparing values only against past observations of the same day-type. Days have been divided into weekdays (Monday through Friday) and weekend days (Saturday and Sunday). So if the algorithm has to process an observation on a Sunday, it compares against data from past Saturdays and Sundays but never from weekdays. This assumes that a weekday is always a day which shows an usage of the same magnitude in our data as other weekdays. Special holidays during weekdays are currently not implemented and will therefore be considered anomalous by the Fuzzy algorithm.

4.2.1 Parameters

The Fuzzy algorithm needs a few parameters which are all tunable by editing the configuration file. First and foremost is the *lookback* parameter which specifies with how many reference values the current value is compared. Setting *lookback* to 7, the default value, makes the Fuzzy algorithm compare the current value to the values at the same time of the last 7 days of the same day-type. If the current day happens to be a Saturday, this would mean that the farthest day still being looked at is 27 days ago.

The next-most important parameter is the *alertscore. Alertscore* specifies the score at which the current value is considered anomalous. If the summed up scores exceed this value, an alert status for the current time interval is set. *Alertwindow* and *alertthreshold* define the size of the window and the number of such alert states to be encountered before an alert is logged to the system. The window always looks back from the current value, thus including the current time interval and the $alertwindow - 1$ last intervals.

With the *uselinear* parameter the scoring method to use can be selected. The scoring methods are discussed in the next section. If this value is set to 1, linear scoring is used. Otherwise the exponential scoring method takes place.

Finally it has been found that with much lower traffic on weekends, the influence of bursts is higher due to the smaller traffic volume. Many false positives during weekends were the result. This led to the introduction of a new parameter, *weekendbonus*, which raises the *alertthreshold* during the weekend by the given amount. The default value of *weekendbonus* is 6. Also when working on generally low volume signals like the count of flows (in comparison against bytes transferred), the normal deviations from the mean in the signal can be too small and cause the Fuzzy algorithm to never raise an alert. To prevent this, the *flowfudge* parameter has been added which allows to lower the *alertscore* value by the given amount if the algorithm works on "low-volume" flow data.

4.2.2 Scoring

There are two different scoring methods implemented in the Fuzzy-Algorithm, a linear and an exponential scoring scale. Both scoring methods calculate a score between 0 and 5, with 0 being

the "best" score or no significant deviation and 5 being the "worst" score or massive deviation. While scores higher than 5 can be produced by the calculations, they are capped at 5 which equals to over 100% deviation from the comparing value in both the linear and exponential method. These scores are calculated seperately for each of the *lookback* days the algorithm is comparing to.

The linear score simply assigns one point for each 20% increase in deviation from the current observation. This results in a scoring scale as in Table 1.

Score	Deviation
0	below 20%
1	between 20% and 40%
2	between 40% and 60%
3	between 60% and 80%
4	between 80% and 100%
5	over 100% deviation

Table 1: Linear scoring scale in Fuzzy algorithm

When scoring exponentially, the equation applied is

$$score = trunc((e^{deviation} * scale) - shift) \tag{4.4}$$

and

$$\begin{aligned} scale &= \frac{maxScore + shift}{e} \\ deviation &= abs((\frac{oldvalue}{newvalue}) - 1) \end{aligned} \tag{4.5}$$

where *deviation* is the deviation between the old and the new observed value and *scale* a scaling factor. The parameters are chosen such that the resulting score will be 5 when the deviation is 1 (meaning 100% deviation).

Parameters for exponential scoring

When using the exponential scoring method two new parameters become available, *scale* and *shift*. These parameters can be adjusted to change the underlying exponential curve which assigns scores according to the relative deviation. *Scale* can be calculated directly from the above equation and shift can be used to select at which point (deviation) the score should switch from 0 to 1. For example, if one wants to obtain a score of 1 starting at 20% deviation ($y = 0.2$), then using the following equation (which can be developed by inserting equation 4.5 into 4.4) with $maxScore = 5$:

$$shift = \frac{maxScore * e^y - e}{e - e^y} \tag{4.6}$$

Continuing the example, using $y = 0.2$ and $maxScore = 5$ and equation 4.6, the value 2.26386 for *shift* is obtained and the *scale* value then becomes 2.67223 when entering the value into 4.5.

The default values for *scale* and *shift* are 2.57515 and 2, respectively. This results in a scoring as in Table 2.

Note that the parameter *maxScore* in above equations should be the same value as the linear scoring method uses as maximum value in order for the two methods to remain comparable. Linear scoring has a hard-coded maxScore of 5.

Score	Deviation
0	below 15.27%
1	between 15.27% and 44.03%
2	between 44.03% and 66.35%
3	between 66.35% and 84.58%
4	between 84.58% and 100%
5	over 100% deviation

Table 2: Exponential scoring scale in Fuzzy algorithm

4.2.3 Limitations

While the Fuzzy algorithm performs quite well given its simplicity, it has no real perception of circadian rhythms and especially can not distinguish special holidays during weekdays. It furthermore has some sort of asymmetry between week and weekend days because the latter ones need a lot longer to adapt to a changed baseline, because only 2 out of 7 days are weekend days and as such eligible for comparison. This becomes less problematic when there is fewer variance between weekend and weekdays in the signal.

4.2.4 Pseudocode

The Fuzzy algorithm in *pseudocode* looks as follows:

```
uint val, oldval;
int score, qscore, j, windowcount;
double q;

int avg3at (timecode t) {              // average over last 3 values to
   return (valueat(t) + valueat(t-1) + valueat(t-2))/3;
}

val = avg3at(t);
j = 0; i = 1;                          // i counts days back, j counts days taken
while j < LOOKBACK do                  // compare with last 14 days of same type
   if(daytype(t) == daytype(t-i*TAG)) {// same daytype (weekend, weekday)
      oldval = avg3at(t-i*TAG);
      j++; i++;                        // inc counters
   } else {
      i++;
      continue;                        // jump next possible day
   }
   if(val == 0) {                      // special case, val == 0, no div zero!
      if(oldval == 0) {                // if oldval also 0 ok, otherwise assign
         qscore = 0;                   // maxscore
      } else {
         qscore = 5;
      }
   } else {
      q = oldval / val;                // quotient
      q -= 1;                          // minus 1
      q = abs(q);                      // remove sign
      if(USELINEAR == 1) {
         qscore = floor(q * 5);        // scoring  qscore = | q -1 | * 5;
      } else {
         q = exp(q);
         q *= scale;
         qscore = floor (q - shift);   // qscore = exp(| q - 1 |) *scale - shift;
      }                                // exp levels see below
```

```
      if(qscore > 5) qscore = 5;
   }
   score += qscore;
end; //while

// now we have LOOKBACK days. Perfect would be score 0. We say score <= ALERTSCORE
// is ok. The rest is anomalous.
if((not weekend & score > ALERTSCORE) | // check if we need to boost ALERTSCORE
   (weekend & score > ALERTSCORE + WEEKENDBONUS)) // for weekend
   setalertat(t);

// alert window:
for i = 0 to ALERTWINDOW-1 do          // check this and 4 last alertstatus
  if(t-i hasAlert)
     windowcount++;
  end; //for
  if(windowcount >= ALERTTHRESHOLD)    // "3 alerts in window of 5"
     raiseAlarm();
} //for
END
```

4.3 Lookback-Algorithm

Lookback is a second algorithm developed for this thesis. As the name implies it performs a look back at past data. Similar to the Fuzzy algorithm, Lookback looks at observed values of the same day-type and distinguishes weekdays from weekend days. In order to avoid false positives, Lookback collects data in a window and calculates the median value so that occasional spikes cannot distort the reading too much. The median value of such a window is stored in an array and the deviation of each value in the window from this median is calculated and the largest deviation is also stored for later use. The maximum deviation from the median in the window is used as a reference point for the deviation to expect.

After the required amount of such Lookback operations and calculations were performed, the median and maximum deviation values that were stored in an array are now used to compare to the current data value. First, the median of the medians in the array is calculated and compared against the current data value (which is not a single data point but also a median of the last data values in a window). The deviation from the current value is then compared to the average of the maximum deviations in the array. If it exceeds this average of maximum deviations, an alert is logged and the current signal considered anomalous, otherwise the data is considered normal.

4.3.1 Parameters

Lookback only uses two parameters for its operation. The first one, *lookback* is shared with the Fuzzy algorithm and selects the number of days of the same day-type to compare the current value against. Second comes *medwindow* which specifies the size of the median-window that is used for looking up the actual and older values. It is set by default to 5 which means that the median value from 5 minute intervals are calculated and compared. The *medwindow* parameter may be an even integer value in which case the two median values in the window are arithmetically averaged and returned.

4.3.2 Limitations

Lookback like the Fuzzy algorithm, has no real notion of daily rhythms except the built-in mechanism to compare the current value to other days' values at the same time. Thus Loopback cannot deal gracefully with holidays during the week either. Furthermore it has been found to perform poorly on low volume signals with relative high variability such as the number of traffic flows.

4.3.3 Pseudocode

The Lookback algorithm in *pseudocode* looks as follows:

```
int getmedian(int start, int end) {  // subroutine, calcs median between
   int len = end-start;              // interval values
   array[len];                       // create array of size
   for i = 0 to len
      array[i] = valueat(start+i);   // store values
   end; // for
   sort(array);                      // sort array
   if(odd(len))                      // if odd number of values
      return(array(len/2));          // return middle one (int div -> 3/2 = 1)
   else
      return((array(len/2) + array((len/2)+1)) / 2); // otherwise alg. average
}                                                    // of middle ones

int getaverage (array a) {           // subroutine, calcs average over array
   int len = a.size;
   int val = 0;
   for i = 0 to len
      val += a[i];                   // sum of all values in array
   return (val/len);                 // return sum/number of values (= average)
}

void lookbackto(timecode t, median m, maxdev d) { // subroutine, calcs values
   int dev = 0; int meddev = 0;                   // over past data
   m = getmedian(t, t-(MEDWINDOW -1));  // select the median from t .. t-(medWindow-1)
   for i = 0 to MEDWINDOW-1 do
      meddev = abs(valueat(t-i) - m);   // distance median - value
      if(meddev > dev)                  // largest yet?
         dev = meddev;                  // save largest deviation from median
   end; //for
   maxdev = dev;                        // set largest deviation from median
}

uint64 med[LOOKBACK];                   // array of median values in the past
uint64 mdev[LOOKBACK];                  // array of max deviation from median in the past
int j=0; int i=1;                       // count sucessful days
while j < LOOKBACK                      // until we have lookback many days
   if(daytype(t) == daytype(t-i*TAG)) {   // same daytype (weekend, weekday)
      lookbackto(t-i*TAG, med[j], mdev[j]); // fill array
      j++;
   }
   i++;
end; //while

uint64 median, deviation, confidence;

confidence = getaverage(mdev);       // get the average of all max deviations seen
median = getmedian(med)              // get the median of all medians seen in interval
deviation = abs(valueat(t)-median)   // deviation between actual value from median of
                                     // medians
if(deviation > confidence)
   raiseAlarm();
END
```

4.4 MinMax-Algorithm

The MinMax algorithm has been developed by Thomas Dübendorfer for the RespiG semester thesis [26]. It is able to find local extrema in a noisy signal and had to be slightly adapted for inclusion in this thesis due to the absence of negative values in our SMTP signal.

MinMax alternatingly searches for a local minimum and a local maximum. It first looks for a data value that is a minimum. Whenever the current value is smaller than the smallest seen so far, that value becomes a temporary minimum. When an observation occurs where the distance from the last temporary value exceeds a specific threshold (in the other direction, i.e. positive when currently searching for a minimum and vice versa), then the last temporary value is converted into an extrema and the current value becomes the new temporary maximum. At this point the algorithm switches and searches for a local maximum.

4.4.1 Parameters

The MinMax algorithm uses only one parameter, *minmaxthreshold* which defines the deviation of the current value from the last found temporary value in order to turn the temporary extrema into a permanent one. Note that there is currently no adaptation of the threshold value to the magnitude of the input signal.

4.4.2 Limitations

Due to the MinMax algorithm's internal function of defining values as temporary extremas and either converting them to final extremas or discarding them later, it cannot work in a realtime manner like the other algorithms. That is to say, the MinMax algorithm is always lagging behind as it cannot decide instantly if the current value is an extrema value or not. This is a problem inherent to the definition of a local extrema. It also does not really raise any alerts but it does find spikes which generally appear like when a sudden increase of traffic flows during a DDoS attack happens. As such, the MinMax algorithm can work together with the other algorithms to decrease the false positive rate by contributing additional information.

Another difference to the other algorithms is the fact that the MinMax algorithm is not stateless. It needs to know whether it is currently looking for a maximum or a minimum and it furthermore needs to store the last temporary value (and time) encountered. All the other algorithms can fetch their data from the archive of stored past data and/or calculate it using a mixture of the current and past network data.

4.4.3 Pseudocode

The MinMax algorithm in *pseudocode* looks as follows:

```
MinMaxContext ctx;                     // context to store state
uint64_t currentValue, currentTime;    // current observed value
ctx = getContext();                    // get state

// better temporary value found ?
if(currentValue * ctx->findMax < ctx->temporaryExtremeValue * ctx->findMax) {
   ctx->temporaryExtremeValue = currentValue; // put into state
   ctx->temporaryExtremeTime = currentTime;
}
// check if threshold exceeded
if(abs(currentValue - ctx->temporaryExtremeValue) > MINMAXTHRESHOLD) {
// add extrema value to extremas, take new value as temporary
   insertExtreme(ctx->temporaryExtremeValue, ctx->temporaryExtremeTime);
   ctx->temporaryExtremeValue = currentValue;
   ctx->temporaryExtremeTime = currentTime;
   ctx->findMax *= -1;                     // and switch direction
}
END
```

5 Implementation

This section describes the design and implementation of our network traffic anomaly detection system. The system has been named "Horus".

5.1 About 'Horus'

Horus is the name of an ancient Egyptian deity. Unfortunately many deities and minor gods at one time or another are associated with that name [27]. One prevalent description of Horus is the battle against his uncle Seth, god of chaos and disorder. By defeating Seth in revenge for the murder of his father, Horus became known as god of order [28]. Horus is often portrayed as hawk-headed being or as a falcon. The symbol of Horus [29] depicted in Figure 5 is a stylistic eye which may symbolize foresight. In later dynasties every Pharaoh was said to be the earthly representation of the god Horus.

Figure 5: Symbol of Horus

Since the anomaly detection system's job is to find the order in the chaos, the name of the god Horus has been chosen for the program. The symbol of the eye is furthermore a fitting picture for a monitoring system.

5.2 Design

Horus, the program that implements our anomaly detection system based on NetFlow data, has been implemented in a way that allows it to either work as a stand-alone application processing the input data from archived NetFlow files, its own preprocessed format or as an UPFrame plugin, which can work on recorded or live data directly from the routers. These two different modes of operation, called *file mode* and *plugin mode* respectively, are explained in Section 5.3.1.

Horus has been built with extendibility in mind. New modules and algorithms (for an explanation of the differences of modules and algorithms, see Section 5.2.3) can relatively easily be added to Horus. Section 5.3.5 explains the procedure to extend the functionality of Horus and the available API functions. The design of Horus can be divided into the following distinct subsystems which are explained in further details below:

Filter and IO: The filter and IO subsystem is responsible for retrieving the network traffic data. This data is filtered so that only desired data records are being processed and stored into the internal data structures.

Data storage: The data storage subsystem provides data structures and access methods for the data. Part of the data is stored in memory only waiting for further processing, while processed data needs to get stored on disk in order to be accessible later and for longer periods of time.

Control: The control subsystem is the glue in Horus, making sure that all subsystems work together. It initializes the program, reads the configuration and controls the execution of the subsystems.

Modules and algorithms: In Horus, the processing of the data is being handled by different modules which can be dynamically turned on or off. The output of these modules then is being processed by one or more of the algorithms described in Section 4.

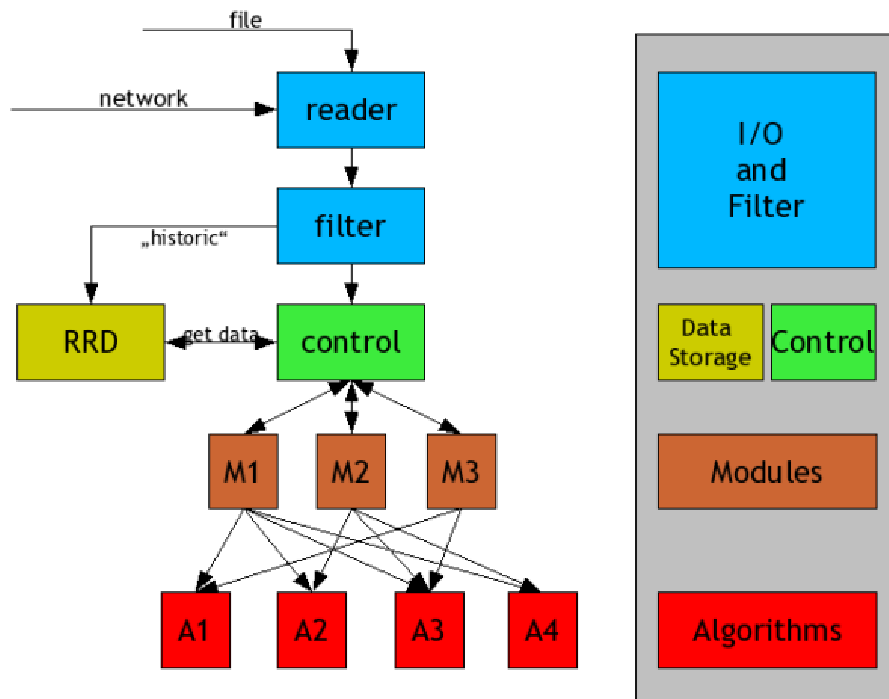

Figure 6: Overview of the Horus program

Figure 6 gives a broad overview of Horus's subsystems and their relation with each other. The flow of the data is depicted in Figure 7. As can be seen, data processing of interesting data (that is, data that is not filtered out) happens two times. The first processing is done by the modules and happens just after a data record has been accepted by the filter. The modules then in turn inspect the data record and add data to the respective *aspect* that they observe. At the time of this writing, there are modules for counting bytes, packets and flows per minute. It is also here, where the handling of long flows takes place. After passing through the modules, a data record is no longer useful and the memory is freed. The modules however store the data of their observed aspect in a special data structure. This structure, the bucket-list, is explained in Section 5.3.2. It is the data in this structure which is subsequently fed to the algorithms and stored for archiving purposes.

5.2.1 Filter and IO

The Filter and IO subsystem's responsibility is to retrieve the data from a source and filter it accordingly such that unwanted data records can be discarded at this early step not using any more system resources. The filter looks at each incoming NetFlow data record which describes one single traffic flow and either accepts the flow and passes it to the data storage subsystem or ignores it, causing the memory of this record to be reused. At the moment the filter is hard coded to only accept TCP packets. The TCP port of the packet the filter accepts can be set through a configuration directive in the config file. By default, this is port 25 (SMTP). See Appendix B for all the valid configuration settings in the config file. A traffic flow record passes the filter if the flow describes a TCP data flow and the source or the destination port matches the configured TCP port.

The subsystem accepts different types of input sources for retrieval of NetFlow data. Additionally, it also accepts a preprocessed special file format. In order to feed live data, UPFrame [17] is required. Please see Appendix C about how to setup and use UPFrame with Horus. In UPFrame mode Horus reads its data from shared memory directly from UPFrame. It then sends each valid NetFlow data record to the filter. Records which are accepted by the filter are then passed on to the different active modules for processing. This is called *plugin mode*.

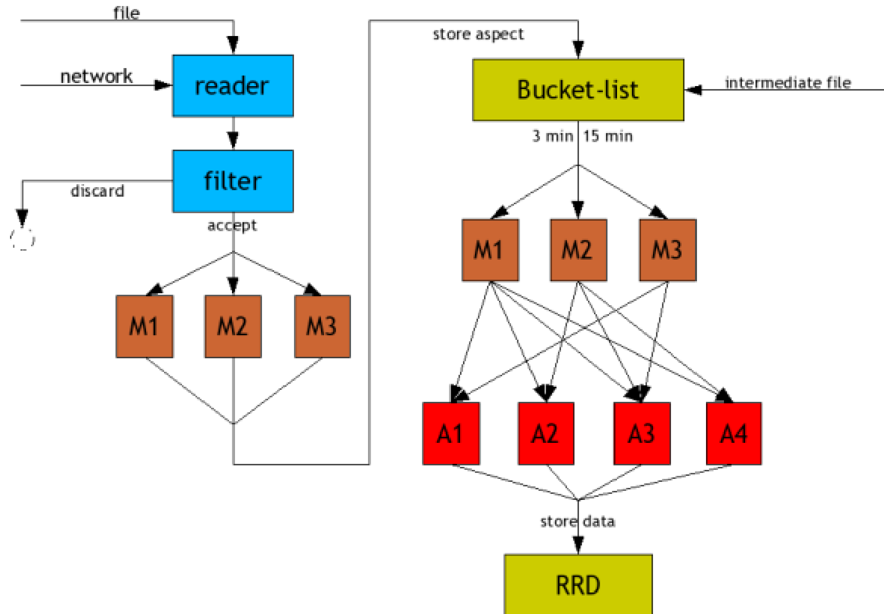

Figure 7: Flow of data through the Horus program

In order to test Horus or trying to find good parameters for the algorithms a reproducible behaviour is needed. This cannot be accomplished using live network data due to the change in input values and the time it requires to perform one single test run until enough input data has been processed. Horus therefore also allows to work on archived NetFlow data. This mode of operation is called *file mode*. Appendix E explains the different file formats that are accepted. The files are essentially NetFlow UDP packets serialized and stored in binary format. They can be bzip2 compressed or uncompressed. Uncompressed NetFlow files use more disk storage but can be processed faster and with less CPU usage while compressed NetFlow files use less disk storage but incur more CPU processing due to the compression and decompression. Both types can be mixed freely in one run because Horus can distinguish the type of the input file. The NetFlow data records stored in the NetFlow files are given over to the filter which accepts and passes them on to processing if they match the filter criterion or rejects them which frees the memory of the record.

Since processing of NetFlow files over a longer period of time such as a month can take a huge amount of time due to the size of the NetFlow files, another type of input data is accepted by Horus. By enabling the *TextOutput*-Module, Horus not only processes the data as it would normally, but it also stores the result of the internal data structure into a text file called *outfile.txt*. This file then contains all the data after filtering the input data and after processing of this input data by the different modules, but before the data is analyzed by the algorithms. Thus when such preprocessed data is again used as input file, Horus can skip the filter and processing of a huge amount of data and directly build the internal data structure which then can be analyzed. This means that a large amount of NetFlow files need to be processed only once and can be directly stored in this intermediate format, which takes a lot less storage space and subsequently less CPU time to analyze.

As an example, several months of NetFlow data totalling over 2 Terabyte of NetFlow files has been converted to an intermediate file which was only a bit over 1 megabyte large. Subsequent analysis of the data then took approximately 30 minutes, while the same analysis over the original files took 22 cluster nodes more than 2 days. It is therefore recommended to work with the intermediate file format for offline analysis whenever possible in order to save time and disk storage space.

5.2.2 Data storage

Horus has two main processing steps and consequently two different types of data to process and store. The first step of processing occurs when the modules are processing the individual flow records that passed the filter. The modules thereby collect data which they store in a special data structure, the bucket-list. This list normally only exists in the memory of Horus but can be stored to disk by enabling the *TextOutput*-Module. The data structure holding this data is described in more detail in Section 5.3.2.

The data is then fed to the algorithms which have to decide whether an alarm condition is present or not. For this decision some of them need to access past data. The output of the algorithms along with the data from the modules is collected in intervals of one minute and stored in a RRD archive. RRD stands for round-robin database and is a special database that never grows in size. Instead the amount of data to be stored must be set and RRD then populates the data in the database by filling up slots. Once all slots are full, RRD overwrites the oldest entry. There are also possibilities to consolidate data together in longer intervals such that the data is available for a limited period in full resolution (which is called a primary data point or PDP) but for a much longer time the average, maximum, minimum or last value of several PDP's.

The data in the round-robin databases can be manipulated by a suite of programs called "RRDTool"[16]. More details about the use of RRDs can be found in Section 5.3. Internally RRDs are used to store the output of the algorithms and the observed aspect values for some weeks. This allows the algorithms at any time to look back to the situation some days ago for comparison or calculation of deviations. The data of the RRDs are generally only present on disk and are not stored for longer periods in main memory. Each access to data stored in a RRD causes the database file on disk to be opened and read. The overhead and resource consumption of the repeated access on these files has not been proven to be a problem. More details about the resource usage can be found in Section 5.4.

Some of the algorithms also put internal states into separate RRDs, so that the number of missing data points or an internal variable at any given time can be read out later. Furthermore, each algorithm puts at least one variable into an RRD which defines whether the algorithm raises an alarm or not. These variables can be read out and processed independently from Horus by another program.

5.2.3 Modules and algorithms

As there are two different processing steps of the input data, there are two distinct types of program code that process the data. These two types are called Modules and Algorithms.

Modules

Modules are little pieces of code that can be switched on and off after compilation of the Horus program by setting the corresponding values in the configuration file. Modules can be thought of as plugins within Horus, though they cannot be loaded dynamically after compilation at this point. It is not difficult to insert a new Module into Horus. The process is detailed in Section 5.3.5.

The job of Modules is to process the raw NetFlow data records that passed the filter into some aggregated value, which we call *aspects*. One example of this is the bpcount-Module which counts bytes and packets of the NetFlow records and stores these values into a bucket-list in one minute intervals. Since NetFlow data records can arrive late, special processing is needed for these *long-flows*. This process is explained in more detail in Section 5.3.3.

The data values the Modules produce as output are stored in a bucket-list. Each entry in the list corresponds to one time interval with a size of 60 seconds. This means that for example the number of bytes for all traffic flows that occurred during the same minute are summed up and stored together in the same bucket-list entry. Modules are free to calculate more than one aspect at the same time. For example, the Module *bpcount* counts the number of bytes and packets in one pass, while the Module *flowcount* only counts flows. In order to not mix different aspects, a Module

has to allocate or reserve space in the bucket-list. This process is explained in more detail in 5.3.2.

A Module does not necessarily need to process NetFlow data records. Modules get called by Horus and passed each NetFlow record that passed the filter but this data can be ignored. Later, Modules are called again to process the bucket-list entry for a specific minute. The *TextOutput*-Module is a special Module that does ignore all NetFlow data records, but stores the entire contents of the bucket-list entry in to a text file. *MemoryLogger* is another special Module which does nothing else than checking the amount of memory currently in use by Horus and storing that value into a round-robin database.

Modules need to conform to some basic interface by providing a small set of function calls, which are recorded in a special structure by Horus. Each Module gets called once upon initialization and once upon finalization, before Horus terminates. The remaining functions are called by Horus whenever a new NetFlow record that passed the filter is encountered (there are actually two different function calls, one for short and one for long flows) or when a bucket-list entry is to be processed which are again two different calls, once for an entry that is at most 3 minutes older than the newest entry and once for each entry that is older than 15 minutes. Details about this behaviour is found in Section 5.3.3. A Module can perform any arbitrary action during its runtime. It can furthermore allocate and use a new RRD, register a new aspect in the bucket-list or disable itself (or another Module). The possible actions are outlined in Section 5.3.5 and the API is detailed in Appendix D.

Algorithms

The algorithms were described previously in Section 4 and are executed by the Modules on their respective aspect. An algorithm takes the current time code as input and retrieves the calculated value for that current time. It then proceeds to perform its calculations and stores the resulting output in a round-robin database. It is possible to select exactly which of the four implemented algorithms work on what aspects. This can be configured by setting the correct values in the configuration file. See Appendix B for details.

The Holt-Winters algorithm is an exception to this rule since it has been built directly into RRDTool[2] and inserting the processed value from a Module is sufficient to initiate the Holt-Winters forecasting algorithm for the value at the insertion time.

5.2.4 Control

The rest of the functions not already described above can be summed up under the control subsystem. This part of Horus performs the setup of all necessary data structures, parses the command line arguments given and reads in a configuration file with additional configuration directives. Arguments and parameters are then sanitized where necessary, followed by the preparation of the subsystems and activation of the necessary Modules conforming to the actual configuration. Finally processing starts in the manner as given by the command line arguments.

Other duties of the control subsystem are the interface to the round-robin database system through the use of an API library which provides functions to create, read and update data in the RRDs, as well as necessary management functionality so as not to lose track of any used resource. Going hand in hand is the memory management using function hooks to track and prevent excessive memory use. Horus can be configured to never exceed a specific amount of memory and it will not allow any subsystem to allocate more than this limit. When memory consumption reaches a level 1 megabyte short of the hard limit, warnings are issued. If a memory allocation fails due to overdrawing the limit, a special function call is performed which may be used to free less important memory. This function is currently not used but is available.

The memory management also provides functions to retrieve the amount of currently used memory, which can conveniently be used to check if there is a resource leak anywhere in the code of a Module or Algorithm. Additionally, functions are provided to access the configuration

[2] starting with the development version 1.1.x

directives of the configuration file and for logging fatal and non-fatal errors into the error-logfile maintained by Horus. Please see Appendix D for a detailed list of available API functions and their explanations.

5.3 Implementation details

In order to manage and store the incoming data efficiently, Horus aggregates the aspect output of the Modules into entries of its internal block-list structure, like described in Section 5.3.2 below. Such an entry represents an interval of one minute. This means that Horus works with a resolution of one minute. This is also the interval that data gets stored in the RRDs. The number describing the entry is the time point at which this one minute interval starts in the standard Unix epoch format. We call these values timecode within Horus. Since the epoch format counts seconds, two consecutive timecodes differ exactly by a value of 60 (seconds) from each other.

The use of RRD as a storage medium for data has some consequences. For one, RRD does aggregate and consolidate data values, if they are not arriving in the same interval as was foreseen at creation time of the RRD archive. This is why Horus always inserts data into RRDs at exact timecode intervals. Furthermore, RRD overwrites old data that exceed the preset amount of slots. This has the effect that if more than a certain amount of values have been inserted, old data is overwritten and lost. At the time of this writing, the RRD archives store 90 days of values (129600 data points in minute resolution).

Another peculiarity of RRD is that it can optionally consolidate several data points into one single data point of smaller resolution. While this feature is not used in Horus, this has the consequence that RRD never allows to insert data with a timecode value of x, if data has previously been inserted at timecode y and $x \leq y$. Consequently, data must always be inserted in strict ascending order. If data is missing at any one point, RRD returns *NaN*, not 0 or an arbitrary value.

We recommend to familiarize yourself with RRD first before trying to extend Horus or reading data out of the RRD files Horus' algorithms create and use. More information about the function of RRD can be found at the official homepage [16].

Horus writes its RRD files into the configured work directory (or the current directory, if none is given). The RRD file names should be self describing. A capital "H" in the name signifies an RRD file of *high precision*, ie one that contains the data after the 15 minute period in which long flows are included and a capital "L" hence signifies the *lower precision*. Low precision data gets processed and written when a new element with a timecode that lies 4 minutes ahead is added to the bucket list. This allows a short reaction time at the cost of the contributing data of long flows that span more than 3 elements and could not yet be recorded.

The RRD files can be queried independently by other programs. This can be used to get the actual status of Horus, since an alarming system has not been included (yet). In order to find the alert conditions, the *data sources* defined in Table 3 are available.

5.3.1 Modes of operation

There are two basic modes in which Horus can be running, *file mode* and *plugin mode*.

In *file mode*, Horus is pulling the flow data from files by either reading NetFlow files or the already preprocessed data from its own intermediate format. Whenever Horus reaches the end of the file(s), it finishes processing by flushing the remaining data from the internal data structure to the Modules and exits afterwards.

When in *plugin mode*, Horus is running as an UPFrame plugin, retrieving NetFlow data over the framework out of shared memory. Since there is not really an "end" of data, Horus does never stop until it is forcefully terminated. While in *plugin mode*, the parameter *-n* | *–no-flush* to

Algorithm type	DS-name	Explanation
Holt-Winters	FAILURES (CF-Type)	Set to 1 if too many observations exceeded the confidence intervall in a window
Fuzzy	alert	Set to 1 if an alert (score exceeded alert score) has been reported
Fuzzy	walert	Set to 1 if too many alerts within the alert window have been encountered
Fuzzy	missing	Set to 1 if there was data missing, so the result is expected to be skewed
Lookback	alert	Set to 1 if an alert has been reported by the Loopback algorithm
Lookback	missing	Set to 1 if there was data missing, so the result is expected to be skewed
MinMax	extrema	Set to 1 if an exremum was detected

Table 3: Defined data sources in the round robin database files

prevent data from being handed to the Modules until all files has been processed is consequently meaningless.

5.3.2 Data structures

There are two important data structures in Horus, the global configuration structure *_HORUS_CONFIG* which is available[3] under the global symbol name "config" and a special linked-list which is called bucket-list.

The configuration data accessible under "config" contains all the flags and parameters that can be manipulated by the command line arguments such as the debuglevel and the list of input files to process.

At the core of data processing in Horus lies a special linked-list called bucket-list. This list, which is accessible through the global configuration structure stores the aspect data of the Modules. The bucket-list stores list entries which are sorted according to their *time value*. This means that there exists exactly one list entry per minute of flow data encountered. At the front of the list are the newest entries (the entries with the highest *time value*) while the oldest entries are located at the rear of the list. See Figure 8 for an overview of the lists structure.

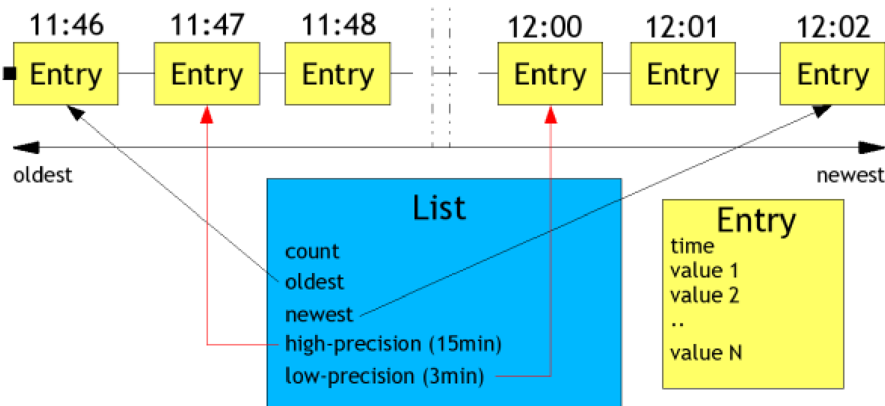

Figure 8: Structure of the bucket-list

The list maintains pointers to the oldest and newest entries as well as two special pointers

[3]when including global.h

to the entries at the position which are (according to the *time value* in the entries) 3 and 15 minutes past the newest entries. These pointers are used to decide when data is to be sent to the Modules for the second round of processing where the algorithms are used to detect anomalies. The 3 minute pointer is internally called the "low-resolution" pointer since it is likely that the entry at the 3 minute position does not yet contain information about newer long-flows. Such flows while being active at this time, have not yet terminated and are therefore not reported yet. Conversely, the pointer at the 15 minute position is called the "high-resolution" pointer as at this time all flow data has been received. Note that the exact maximum time a router holds back to report long flows depends on the router's configuration and can most likely be changed. In order to work correctly with Horus, the router(s) active timeout value at which long, still active flows are reported must not exceed 15 minutes.

When Horus is actively processing network flow data, the list is growing with every new minute interval encountered and the list entries are removed automatically again, when they have been processed by a call to the Modules' high-precision "store" function. This means that unless the *-n | –no-flush* parameter has been given, the list will grow to 15 elements at most.

A list entry contains fields in an array where Modules store the data of their observed aspects in the minute interval equal to the entries' *time value*. To allow several aspects to be handled without unwanted mixing or static allocation, Modules are required to request and reserve their storage space. Appendix D explains how this can be accomplished.

5.3.3 Handling of long flows

We call network traffic flows that span more than one entry in the bucket-list "long flows". Long flows require special processing, depending on the aspect that is being worked on. In some cases this is trivially easy, in others more effort is needed. For example when counting flows, we only need to increment the counter for each entry in the bucket-list that is covering the time interval, during which the network flow was active. However, if we want to count the bytes of the flow, we need to distribute the overall amount from the NetFlow record among all the bucket-list entries during the flow's duration. Since we only have the information about the total amount of bytes (and packets), we assume that the flow sent data with a fixed average speed. Thus, the amount of bytes transmitted per minute interval is calculated and summed up. There is a special treatment for the first and last entry of the bucket-list to add only a fraction of the full minute interval, if the flow did not start exactly at the minute boundary.

The *bpcount*-Module found in *src/bpcount.c* performs this distribution of a network flow's bytes and packets over several bucket-list entries, while the solution in the *flowcount*-Module in *src/flowcount.c* simply adds the counter of the proper entries.

In order to retrieve the exact start time of a flow, the router's uptime from the NetFlow header files must be used and special treatment taken, since that value wraps around approximately every 40 days. Luckily, this speciality has already been taken care of by the NetFlow-tools package from the DDoSVax team.

This is the reason why every Module needs to implement two functions, one for normal, short flows and one for long flows. As a test run with special code has shown, there are relatively few long flows with significant size. Table 4 shows the distribution of long flows compared by their size. The data used to create this statistic is a random selection of 4 hours of NetFlow data and a full run over archived data during the Sobig.F outbreak consisting of 270 GBytes of compressed data in 1130 files each spanning an hour.

Since processing of long flows can be far more complex than that of normal flows, Horus by default only calls the long flow handling code of Modules if the flow is at least 100 KBytes large. Thus using less processing resources at the cost of a slight error by not distributing the data of smaller flows over all appropriate bucket-list entries. This default value can be overridden by the *-t or –threshold* command line argument. Setting it to 0 causes Horus to perform the long flow handling on all flows that span more than one bucket-list entry, any other value will set the limit under which a long flow will be accounted for by the single flow handling code which

stores the aspects of the flow in one single bucket-list entry.

Size	Longflows (4 random)	Percentage	Longflows (Sobig.F)	Percentage
0 .. 1 KB	431557	90.57%	10494834	63.63%
1 KB .. 10 KB	34817	7.31%	4416527	26.78%
10 KB .. 100 KB	5972	1.25%	602427	3.65%
100 KB .. 1 MB	2427	0.51%	893114	5.41%
1 MB ..	1740	0.37%	86655	0.53%
total	476513	100%	16493557	100%

Table 4: Distribution of long flows according to size

5.3.4 Configuration

The configuration and parameters of Horus can be influenced by the command line arguments when executing the Horus binary and configuration directives read from a configuration file. First the arguments given on the command line are evaluated, afterwards the configuration file is parsed. A different location and name of the configuration file can be specified with the `-c <config file>` argument. If an illegal value has been given as an command line argument, execution stops immediately. When giving no argument at all, a short usage is printed giving all available command line arguments. Giving `–help` or `-?` as only argument will cause Horus to print an explanation of its parameters. Appendix A lists and explains all available command line arguments.

Once the command line arguments have been processed, a configuration file is read. If it has not been explicitly set by the `-c` command line argument, the file *horus.conf* in the current directory is used. Note that directives in the configuration file can overwrite command line arguments. Currently this is the case with the `-w <directory>` argument and the `config.workdir` directive which both set the working directory where Horus will store and use its output data files. The full list and explanation of available configuration directives for the configuration file can be found in Appendix B.

5.3.5 Extending Horus

Horus can be extended by adding new Modules and Algorithms. In order to facilitate work, an easy to use API is being provided by Horus. The details of which can be found in Appendix D.

A new algorithm can be added simply into *src/algo.c*. In order to be used by the Modules however, each Module must be changed to actually take advantage of any new algorithm. This is preferably being done optionally, controlled by a configuration directive in the config file. It is important to remember that algorithms basically work stateless. They are called by Modules and can receive data such as the actual bucket-list entry, but otherwise don't have any state at all. If a new algorithm needs to preserve state, that must be stored either in its own RRD (probably distinct for each Module that uses the algorithm) or using some other, own mechanism.

Integrating a new Module requires a few steps more. Modules should be implemented in their own file and therefore need to be added to the Makefile in order to be compiled automatically. Modules need to conform to some basic interface, which means that they have to implement 6 different functions. The function prototypes are defined in *global.h* and are also listed and explained in Appendix D. Since Horus does not support dynamic loading of Modules, they need to be known to Horus at compile time. This means that a special registration function in the new Module needs to be called by *horus.c* during runtime. This function then fills in a *_MODULE* structure where the 6 necessary function calls are registered. By adding this structure to the global list of Modules, the new Module then becomes known to the system and can be used. The file *src/foo.c* contains a dummy Module that does not perform anything but can be used as an example on how to populate the module structure. The registration function needs to be known to Horus, so a forward declaration should be added to *horus.h*, where all

current modules' registrations are placed. The call to this registration function then takes place in *src/horus.c* in the *registerModules* function. Due to the way the Modules add themselves to the list, the first called Module will become the last one in the chain of execution. The exact chain is being printed to the console when running Horus with a debuglevel of 6 or higher.

The aforementioned Module *src/foo.c* is a good reference on the basic usage of the interface for adding a Module to the system. *src/textoutput.c* furthermore performs some simple processing of data by writing the contents of a bucket-list entry to a text file and *src/memlog.c* uses an RRD to store current memory usage. To understand the processing of raw network flow data into aspects and perform distribution of data over several entries for long flows, a close look at *src/bpcount.c* and *flowcount.c* is recommended, both of which perform long flow handling in a different way. For starters, it is recommended to only process flows which do not span multiple time intervals, which is easier to understand and implement.

5.4 Resource usage

One of the requirements for the implementation was the efficient resource usage for operation under tight resource constraints. Horus has been implemented carefully such that memory usage stays low (generally near 1 MB, unless the *-n* | *–no-flush* command line argument has been given). The memory usage of Horus can be monitored by turning on the *MemoryLogger*-Module, which stores the current memory usage of Horus once per processing of a bucket-list entry (which happens once per minute normally) into an RRD. Memory usage can furthermore be limited in Horus directly which will prevent it to use more than the given amount of RAM.

CPU usage varies depending on the mode of operation. Since the *plugin mode* for UPFrame has been implemented late in the thesis, no comparison values are available. However when looking at the resource usage when Horus runs in *file mode* where it processes data as fast as it can (under full CPU utilization) the measurements show that Horus is mainly bound by I/O limits. For example, processing 24 hours of NetFlow files comprising of 3.3 GByte bzip2 compressed data took 35 minutes and 57 seconds. This means that Horus is faster by a factor of 40 compared to real-time. The factor can be improved by processing uncompressed NetFlow files, then equaling to 10 GBytes which Horus processes in 5 minutes and 12 seconds (factor 277 compared to real-time). The system which ran Horus was a AMD Athlon XP 2800+ with 1 GByte of RAM.

By letting Horus process data from its *intermediate fileformat*, we can bypass the NetFlow I/O, the filter and the processing of the raw data into aspects by the Modules. What remains is a performance measure of the algorithms and the cost of storing and accessing that data into the RRDs. Measuring the time of processing the data read from the intermediate file took only 11 seconds (under full CPU utilization). This can be used as a measure of comparison between CPU usage for processing and I/O. As can be seen, data processing is taking less than 5 % of the time compared to a full run on NetFlow files read from disk (uncompressed). This should provide ample reserves for processing in near real-time. Once near real-time processing is no longer possible, the step to convert the raw network flow data into aspects could be moved to a dedicated system.

6 Evaluation

For evaluation, Horus was fed archived data from known worm events and also from weeks with no known anomalies. It was revealed quickly, that each single algorithm used found the signal anomalous at times where no alert should have been reported. These false positives could be suppressed by combining the output of several algorithms and only consider an anomalous situation when at least 3 raise an alert condition.

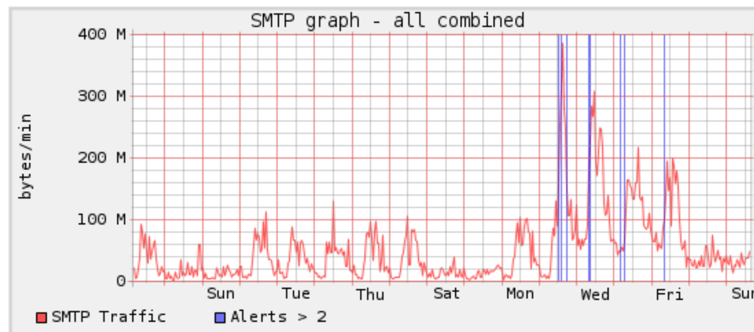

Figure 9: Plot of Sobig.F outbreak. Combined output of Holt-Winters, Fuzzy and Loopback

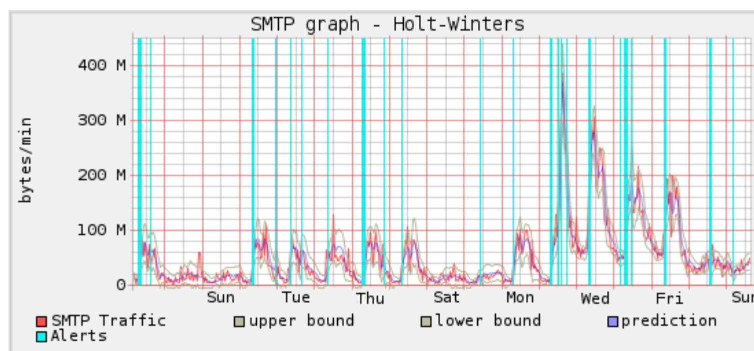

Figure 10: Output from Holt-Winters of Sobig.F outbreak

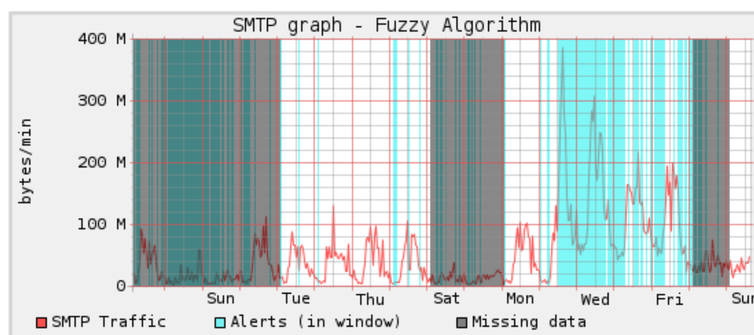

Figure 11: Output from Fuzzy of Sobig.F outbreak. There is some missing data prior to te outbreak.

Figures 9 - 12 show a plot of the combined output of the Holt-Winters, Fuzzy and Lookback algorithms and each algorithm separately on data during the Sobig.F outbreak using the bytes per minute aspect. It can be clearly seen that each individual algorithm produces a lot more alerts than all of them combined.

Performing the same analysis over a week of normal traffic as can be seen in Figures 13 - 16, again the combined algorithms produce an acceptable result but the individual algorithms by

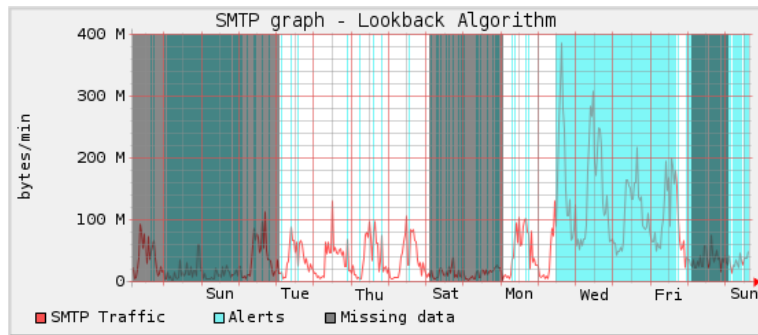

Figure 12: Output from Lookback of Sobig.F outbreak. There is some missing data prior to the outbreak.

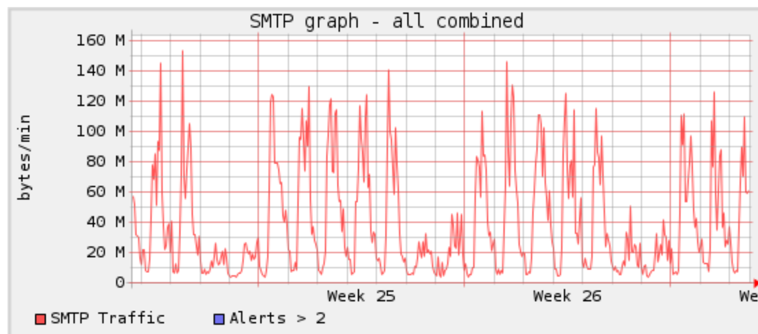

Figure 13: Plot of one week normal traffic. Combined output of Holt-Winters, Fuzzy and Loopback

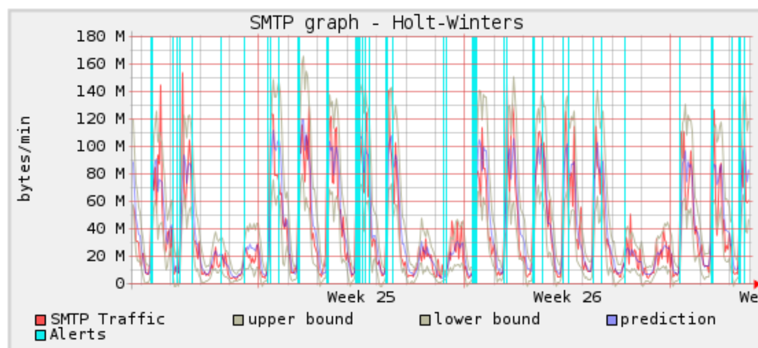

Figure 14: Output from Holt-Winters of one week normal traffic

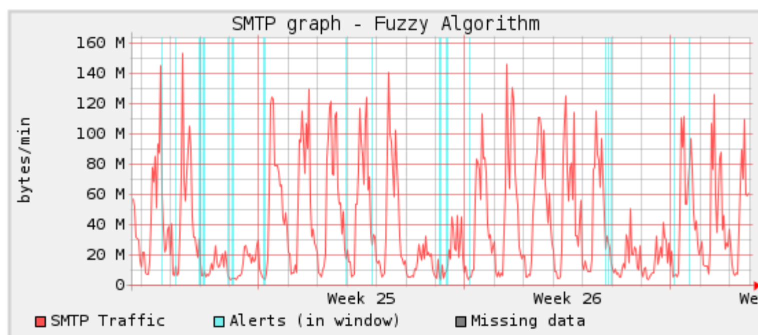

Figure 15: Output from Fuzzy of one week normal traffic

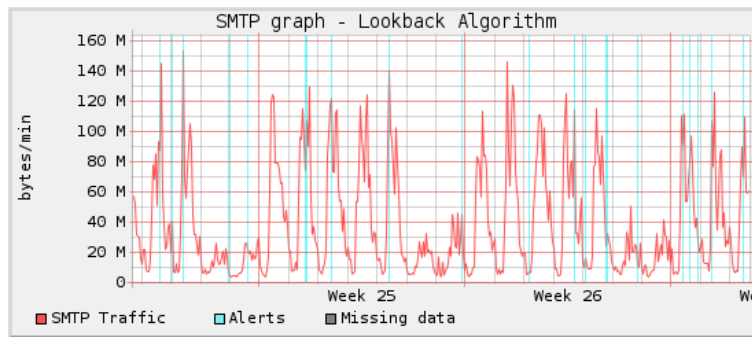

Figure 16: Output from Lookback of one week normal traffic

themselves are classifying too much traffic as anomalous.

When feeding incomplete data to Horus, the detection rate deteriorates fast. Figure 17 shows the combined output of the Holt-Winters, Fuzzy and Lookback algorithms on data from only one router (of four total). The algorithms are not able to detect the outbreak on Tuesday. Inspecting the signal it is however not really possible for a human either to find the time of the outbreak without prior knowledge of the exact date and time.

Further results revealed that the packet aspect showed much similarity to the byte aspect, albeit at a much lower magnitude. It showed almost the same large variability though which made it a poor choice to use. Using the flow aspect, we observed that the Fuzzy algorithm often stayed very close to the alert threshold only seldom exceeding it for short amount of times during anomalous conditions. We then added another parameter to the Fuzzy algorithm (*flowfudge*) which lowers the alertscore a bit when working on the flow aspect. With this small change, detection rate could be improved.

We also found that the Lookback algorithm performed very poorly on the flow aspect. It produced a huge amount of false positives, declaring detection of an anomalous signal almost all the time. On the other hand, the flow aspect proved to be a good input signal for the MinMax algorithm to detect sudden spikes. We then decided to turn Lookback off and MinMax on for the flow aspect by default.

A peculiarity that has been found is that the Holt-Winters algorithm, while generally be well able to adapt to the usage pattern, seems to have trouble with a steep raise we observed in the mornings of work days. As can be seen in Figure 18, Holt-Winters is reporting anomalous behaviour on every morning. We attribute this form of the signal to the fact that the people using this network being almost uniformly punctual which reflects in an almost simultaneous start of usage.

Unfortunately we had to find out towards the end of the thesis, that RRDTool which has been

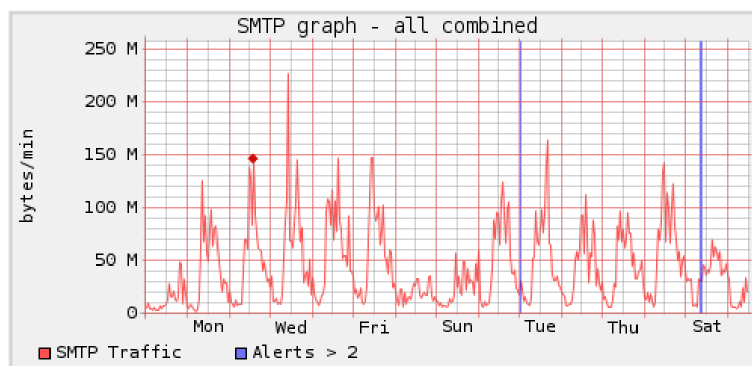

Figure 17: Plot of mydoom outbreak with incomplete data from only one router

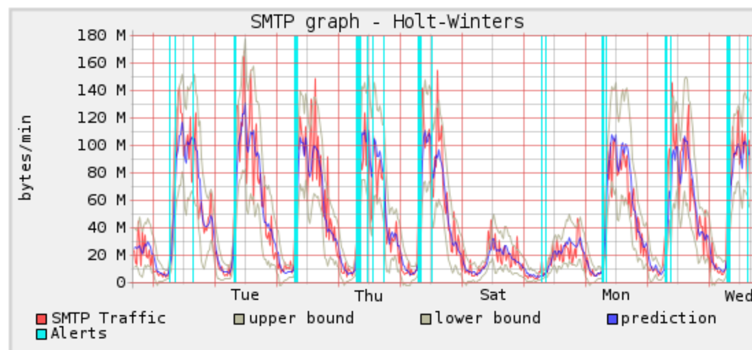

Figure 18: Plot of Holt-Winter unable to cope with a steep usage increase during morning hours

used to plot the graphics, tends to suppress information. With our resolution of one data point per minute and a resulting amount of 1440 points per day, RRDTool is prone to average or suppress data when plotting larger periods of time (such as a week). The main problem is the fact that the graphical representation of an alert, which is a vertical line in the plot, can also be suppressed. The result is that there are more alerts during the period of a plot than actually painted. So we have currently more false positives than expected. The problem can be mitigated somewhat by creating a moving window for the alerts and selecting good parameters for the length of the window and the amount of alerts encountered before raising a "real alert" that notifies a human operator. Still, the false positive rate is too high.

7 Summary and Outlook

During this Diploma thesis, an UPFrame plugin named Horus has been developed. It is capable to read in network flow data delivered via UPFrame in near real-time from live routers or to process archived data read from disk. The constraints of careful resource usage in terms of required processing power and memory utilization have been met. Through close observation of resource usage during the implementation phase, a memory leak in the Holt-Winters algorithm in the used development version of RRDTool has been found and fixed[4].

Horus implements two data processing Modules which observe three different aspects of the network flow data and four Algorithms to analyze the resulting aspect data. This results in 12 different combinations that can be used in order to detect anomalies. Not all of them have been proven useful.

Problems encountered during this thesis range from badly documented APIs to the very noisy signal to start with. The input data makes it hard to find good parameters and the sheer amount of data caused long delays until the result of one run was available. This led to the development of the *intermediate format* which sped up cycle times tremendously.

While the system basically works, there are still too many false positives in order to provide a useful tool for backbone operators. More work is certainly needed to reach the initially set high goals.

7.1 Outlook

Some ideas on how to improve the detection rate of Horus or Horus itself that surfaced during this thesis are listed below:

- Finding and implementing better aspects which more clearly expose anomalies.
- Finding and implementing better algorithms which adapt to the traffic rhythm and have a low false positive rate.
- When an alert is raised, start a program that tries to verify the claim independently and using more resource demanding analysis techniques. For example if a sudden increase in SMTP traffic flows is detected, an external program could look at all the NetFlow traffic during the event interval and try to find out what the exact source of the spike was.
- Trying to reduce the amount of parameters needed by changing the algorithms to find the best values themselves.
- Adding caching mechanism to reduce disk I/O for lookups in the RRDs.
- Adding the ability to change configuration during runtime.
- Adding a configurable alarming system.

7.2 Thanks

I would like to thank my tutors, Thomas Dübendorfer and Arno Wagner for their great support during the work on this thesis. Thanks also go out to Tobi Oetiker, creator of RRDTool which the author has been using for a long time outside of this thesis' scope and to Jake Brutlag who implemented the Holt-Winters forecasting algorithm in the development branch of RRDTool. A great help were the templates from Lukas Ruf [5] which were used to create this documentation. I wish to further thank Urs Steiner for proof-reading and helpful hints as well as all the people that were working at G.69 for the fun discussions and gossiping at the dining table.

[4]see `http://www.ee.ethz.ch/~slist/rrd-developers/msg01380.html`

A Parameters

There are several command line arguments that can be passed to Horus. A short list of all available arguments will be printed when Horus is executed with no arguments given:

```
pc:~$ ./horus
Usage: horus [-d|--debuglevel <num>] [-l|--reclimit <num>]
        [-o|--output outputfile.txt] [-e|--every <N>] [-p|--p-offset <N>]
        [-m|--memlimit <num>MBytes] [-n|--no-flush] [-t|--threshold <num>Bytes]
        [-x|--textfile] [-q|--quiet] [-c|--config-file other.conf]
        [-w|--workdir <dir>] [-?|--help] [--usage]
        [OPTIONS]* -f <files to process>
```

Using the –help or -? argument causes Horus to display and explain each argument in a more verbose manner.

The following list details all available parameters and their effect:

Short	Long	Argument-Type	Explanation
-d	–debuglevel	Integer 0..30	Sets the debuglevel in Horus. The higher the value, the more verbose the messages printed to the console are. Defaults to 5 which prints only few messages. Set to 2 or 1 to suppress all but errors. To furthermore suppress errors being printed to the console, also set -q.
-l	–reclimit	Integer 1..x	When in *file mode*, aborts processing of the current file after having processed that many records. This can be used to quickly check some functions.
-o	–output	String filename	This is an old method of producing some data in a special text format which was used during development. Should not be used anymore.
-e	–every	Integer 2..x	When in *file mode*, this causes Horus to not process every single file, but only every Nth file. This mechanism can be used to process a large amount of data on several machines in parallel. Use together with the -p | –p-offset argument.
-p	–p-offset	Integer 0..[every-1]	With this argument, the offset of the Horus program can be set. This is used in conjunction with the -e | –every argument. For example, when -p is set to 0 and -e to 2, this program instance will process all the odd files in given as input, starting with the first then the third, etc. If -p is set to 1, it will move its offset by this amount and thus process all even files starting with the second, then the fourth and so on.
-m	–memlimit	Integer 1..x	Sets the amount of memory in Megabytes that Horus may use up maximally. If this value is set too low, Horus will most likely abort in the middle of processing. Default value is 2 (Megabytes). Under normal conditions, Horus will not need much more than 1 MB. When using argument -n | –no-flush the usage can increase much.

-n	–no-flush	None	Flag to switch a special mode in Horus. When *no-flush* is set, the internal bucket-list is not processed by the modules and algorithms until the last input file has been fully read and all records have been stored in the bucket-list. This overrides the normal behaviour where bucket-list entries are given to the Modules after at most 16 minutes. Can be used whenever there are several distinct files with data during the same period that should be aggregated (when having separate files for each router, for example).
-t	–threshold	Integer 0..x	Sets the threshold for long flows. A flow with fewer than that many bytes will be handled as if it were a short flow. Defaults to 100 KBytes (102400). Set to 0 if all flows that span more than one minute interval are to be handled by the long flow code.
-x	–textfile	None	Flag to indicate that the input file is not a NetFlow data file but a text file containing data in the *intermediate format* and needs to be processed differently. Note that when using *-x* only one single file is processed, no matter how many are given on the command line.
-q	–quiet	None	Flag to indicate that non-critical errors are not logged to stderr. This behaviour is overridden when Horus cannot write to its error logfile. Does *not* lower verbosity of Horus' messages. Use -d | –debuglevel to suppress those. Note that without this parameter, Horus complains if it cannot find its RRD files for the first time and needs to create them.
-c	–config-file	String filename	Reads the configuration from an alternative config file. Default uses *horus.conf* in the current directory.
-w	–workdir	String directory	Sets the work directory where Horus stores its database files. This does not change where Horus reads the configuration file or input files to process from. Default is the current directory.
-f	None	String filenames	The file(s) to process. Can be any amount of files Horus is to process. Implicitly switches *file mode* on.
-u	–upframe-dir	String directory	Switches Horus into *plugin mode* and selects the directory for use with the UPFrame fifo's.
-?	–help	None	Prints a list of all command line arguments.

Table 5: All available command line arguments in Horus

As an example, the following line starts Horus in *file mode* parsing the file *mydoom-parsed.txt* in the current directory as an *intermediate* file and stores all its output in the */tmp* directory.

```
pc:~$ ./horus -d 2 -f mydoom-parsed.txt -x -w /tmp
```

The following line starts Horus with more verbose message output to process all the files in the */data/netflow* directory and storing the output in */tmp/processed* while not allowing to use more than 10 Megabytes of memory.

```
pc:~$ ./horus -d 7 -m 10 -w /tmp/processed -f /data/netflow/*
```

B Configuration

The configuration file (by default *horus.conf*) can contain directives to configure the behaviour and parameters of Horus. It is read in by Horus after the command line arguments have been parsed but before any processing of data starts.

The directives in the configuration file are name/value pairs, separated by the equality character ("=") and are stored in an internal hash-table. They are directly accessible through the API (see Section D). Leading and trailing spaces are removed from the name and the values. Where a simple yes/no answer is required, the value 0 corresponds to "no" and the value 1 to "yes".

In order to see which values are stored under which names in the internal hash-table, set the debuglevel of Horus to at least 9 which will cause all pairs to be printed to the console. On debuglevel 10, Horus will furthermore display all the lines it ignored in the configuration file.

Table 6 on page 39 lists the names that have already been defined and used by Horus. Any number of additional entries can be added and queried by use of the API.

An example output of Horus when called with debuglevel set to 9 and using a non-empty configuration file:

```
pc:~$ ./horus -d 9
DEBUGLEVEL = 9, limit = 0, output = [(null)], infilecount=0
parsing config file [horus.conf]
inserting [enable.dummy]=[no] into hashtable
inserting [enable.bpcount]=[Yes] into hashtable
inserting [enable.flowcount]=[1] into hashtable
inserting [enable.textoutput]=[0] into hashtable
inserting [enable.memorylogger]=[no] into hashtable
inserting [enable.bpcount.dofuzzy]=[yes] into hashtable
inserting [enable.bpcount.doloopback]=[yes] into hashtable
inserting [enable.bpcount.dominmax]=[no] into hashtable
inserting [enable.flowcount.dofuzzy]=[yes] into hashtable
inserting [enable.flowcount.doloopback]=[no] into hashtable
inserting [enable.flowcount.dominmax]=[yes] into hashtable
inserting [config.portNum]=[25] into hashtable
inserting [config.algo.lookback]=[7] into hashtable
inserting [config.fuzzy.alertscore]=[14] into hashtable
inserting [config.fuzzy.flowfudge]=[5] into hashtable
inserting [config.fuzzy.weekendbonus]=[6] into hashtable
inserting [config.fuzzy.alertwindow]=[5] into hashtable
inserting [config.fuzzy.alertthreshold]=[3] into hashtable
inserting [config.fuzzy.uselinear]=[No] into hashtable
inserting [config.fuzzy.scale]=[2.57515] into hashtable
inserting [config.fuzzy.shift]=[2] into hashtable
inserting [config.lookback.medianwindow]=[5] into hashtable
inserting [config.minmax.threshold]=[2500] into hashtable
parsed text file [horus.conf], processed 104 lines, inserted 23 pairs into hashtable
```

Configuration directives		
Name	Type	Explanation
enable.<*modulename*>	Yes/No	Switches the named Module on or off.
enable.<*modulename*>.dofuzzy	Yes/No	Switches the Fuzzy algorithm on the data of the named Module on or off.
enable.<*modulename*>.dolookback	Yes/No	Switches the Lookback algorithm on the data of the named Module on or off.
enable.<*modulename*>.dominmax	Yes/No	Switches the MinMax algorithm on the data of the named Module on or off.
config.portNum	Integer	Selects the port for the filter. Set to 25 to let the filter pass all data to and from TCP port 25 (smtp).
config.algo.lookback	Integer	Sets the number of days that Lookback and Fuzzy algorithms look back to.
config.fuzzy.alertscore	Integer	Sets the alert threshold (max score) for the Fuzzy algorithm.
config.fuzzy.flowfudge	Integer	Sets the amount by which the alert threshold is lowered when the Fuzzy algorithm processes the flow aspect.
config.fuzzy.weekendbonus	Integer	Sets the amount by which the alert threshold is increased on weekends.
config.fuzzy.alertwindow	Integer	Sets the size of the alert window for the Fuzzy algorithm.
config.fuzzy.alertthreshold	Integer	Sets the number of alerts in the alert window in order to raise an alert condition in the Fuzzy algorithm.
config.fuzzy.uselinear	Yes/No	Selects the scoring mode for the Fuzzy algorithm. If set to ”yes”, linear scoring is used, exponential otherwise.
config.fuzzy.scale	Double	Sets the scale value for exponential scoring in the Fuzzy algorithm. See Section 4.2.2 for a detailed description of this value.
config.fuzzy.shift	Double	Sets the shift value for exponential scoring in the Fuzzy algorithm. See Section 4.2.2 for a detailed description of this value.
config.lookback.medianwindow	Integer	Sets the size of the median-window in the Loopback algorithm.
config.minmax.threshold	Integer	Sets the threshold of the MinMax algorithm.

Table 6: Configuration directives

C Using UPFrame

In order to use Horus in *plugin mode* where it acts as a plugin for the UPFrame framework, you need to start Horus with the command line argument *-u* | *–upframe-dir* and give the directory where the UPFrame FIFO queues reside.

When in *plugin mode*, Horus will not process any additional input files and will also not abort processing as long as UPFrame's mgmt process is still attached. To test the UPFrame functionality, the netflow_replay programm provided with Horus in the *tools directory* can be used. Download UPFrame [17] and compile it first. Note that the library *libupframe.so.0* must be available to Horus. Configure the mgmt and writer as detailed in the UPFrame documentation. Afterwards, a NetFlow file can be sent through UPFrame using as follows:

```
pc:~$ ./netflow_replay -f ./netflowfile1.dat -i 127.0.0.1 -p 19991
```

This will send the content of the NetFlow file as UDP packet stream to localhost (127.0.0.1) to UDP port 19991. If configured correctly, a writer instance of UPFrame receives the packets and hands it to UPFrame, which will in turn send it to Horus in *plugin mode*.

Special care has to be taken that UPFrame is working correctly and that FIFO pipes have the correct modes so that Horus can access them. If the above test is working, Horus is able to receive live network data from routers sent to UPFrame.

D API

Horus exposes many API functions in order to extend functionality easily. The following listing is an overview of the most important structures and API functions grouped by the file where it is defined. The header file of the source file generally has a verbose explanation as well. There are more functions than those detailed here. This is an excerpt of which functions are most likely useful to anyone who wishes to extend the functionality of Horus by adding Modules or Algorithms.

D.1 Structures

D.1.1 Config

The config structure is defined in *global.h*. It contains fields for access to the runtime arguments given to Horus and some other internal structures.

int DEBUGLEVEL	Defines the debuglevel the program runs at. The lower the less debug data should be printet to the console.
FILE *out	Old pointer to an output file. Should no longer be used.
int limit	Holds the number of records that should be processed in file mode, when working on NetFlow files before aborting or 0.
int filecount	Holds the number of input files on the command line.
uint64_t longThreshold	Defines the amount of bytes that a flow needs to exceed in order to undergo long flow processing.
uint16_t portNum	The port number which the filter uses to decide whether to accept a NetFlow record or not.
BLOCKLIST *bucketlist	Pointer to the bucket-list. See also Section 5.3.2.
char * workDirectory	The work directory where all non-input data files should be stored.
char * configFile	The filename of the configuration file.
struct _filename_str *infilelist	List of filenames for reading.
struct _filename_str *outffilelist	List of filenames for writing.
char * memfail	Static buffer for storing an emergency message in case of memory exhaustion.
int p_every	Holds the parameter of the *-e* command line argument.
int p_mod	Holds the parameter of the *-p* command line argument.
int Xnoflush	Flag whether to flush the bucket-list during processing or at the end of reading input data.
int textmode	Flag to indicate that the file to read from is in *intermediate format* and not a NetFlow file.

Table 7: Configuration structure

D.1.2 Module

The module structure contains information about a specific Module. Otherwise it is in essence just a linked list, defining the order of execution of the Modules.

char *name	Name of the Module. Does not need to be the same as the file the Module is defined in.
int enabled	Whether or not this Module should be enabled and used or not.
struct _MODULE *next	Pointer to the next Module following or NULL.
The next entries define the function pointers see *global.h* for the definition	
_moduleInit initFunc	The function to call upon initialization, before starting any processing of data.
_closeAction closeFunc	The function to call upon finalization, before Horus exits the process.
_processSingle singleFunc	The function which performs the processing of a flow record as single flow.
_processLong longFunc	The function which performs the processing of a flow record as long flow.
_storeElementLowPrec storeLowPrecision	The function which analyses and stores the data of a bucket-list element of low precision (after 3 minutes).
_storeElementHighPrec storeHighPrecision	The function which analyses and stores the data of a bucket-list element of high precision (after 15 minutes).

Table 8: Module structure

D.2 Functions

D.2.1 Blocklist

blocklist.c contains the functions for manipulation of a blocklist. The bucket-list as described in Section 5.3.2 is such an instance.

1. Functions for list entries:

   alloc_element Returns an allocated and cleared entry-element for a blocklist.

   clear_element Clears the content of a list entry.

   print_element Prints the content of a list entry to a string buffer.

   add_value_to_element Automatically sums up a value to a specific index in an entry.

2. Functions for list:

   alloc_list Returns an allocated and empty blocklist.

   print_list Prints the list's header data to a string buffer.

   iterate_list Prints the list's entrys' timevalue to a string buffer.

   add_in_front Adds an entry to the list in front (if possible).

   add_element Adds an entry to the list at the correct position.

   remove_oldest Removes and returns the oldest entry in the list.

   remove_element Removes and returns the specified entry in the list, if it exists.

   fetch_element Retrieves the specified entry from the list, if it exists.

   get_element_for_time Returns an entry with the specified time value. If it existed in the list, it is returned. Otherwise a new, empty entry is created, added to the list and returned.

getBlocklistMaxNumIndex Returns the amount of counters an entry has.

registerName Registers a name (for an aspect) in the list and returns the index.

getIndexForName Returns the (formerly registered) index for a given name.

getNameAtIndex Returns the name that was registered for the given index.

D.2.2 Config

config.c contains the functions for querying the configuration file's directives. See also Section B for further information.

getValueFor Returns the value of the specified key as string if it occurs in the hash-table containing the configuration files name/value pairs.

isEnabled Convenience function which returns a true constant if the specified key was found in the hash-table with a value equal to "1" or "yes" (case insensitive). See *config.h* for the defined constants that this function may return.

D.2.3 Elog

elog.c contains functions to log into Horus' error.log.

logError Logs a non-critical error message into error.log.

logError2 Logs an error message to error.log and allows optionally to abort execution of Horus.

logPError Similar to logError2 but appends the output of errno. Replaces perror().

setErrorQuiet Selects whether or not errors are logged to the console also or not. Critical errors which cause execution of Horus to abort are always logged to the console's standard error stream.

D.2.4 Memory

memory.c contains functions to track usage of memory in Horus and to prevent too much memory to be allocated by intercepting the alloc() calls.

get_used_memory Returns the amount of currently used memory.

get_max_memory Returns the amount of memory that has been allocated and not returned to the operating system.

get_configured_max_memory Returns the amount of memory that can be allocated at most.

get_free_memory Returns the amount of memory that can be allocated until the memory library will refuse to allocate more.

init_memory_setting Sets the amount of memory that the library is willing to allocate.

print_status Prints a short string with the amounts of used, free and limit of memory.

D.2.5 RRDlib

rrdlib.c contains the functions for manipulation of RRD files. It is the most complex library in Horus. Before using these functions, make sure you understand how RRD actually works. Information on RRD can be found at the RRDTool page [16].

In order to access a round robin database stored in an *.rrd* file on disk, it is required to create a *context* first. These contexts are stored in a hash-table and can be requested later once they have been created. Each context stores some information about the underlying RRD data file. The filename of the underlying RRD data file is the context name with an appended ".rrd".

1. Functions for manipulation of RRD contexts:

createContext Creates and returns a RRD context known under the given name. If the context has already been created, it is returned.

getRRDContextFor Returns a RRD context that has previously been created under the given name.

destroyContext Destroys the context and removes it from the internal table.

2. Functions for accessing RRD data:

getRRDInfo Returns an info structure with the data similar to calling "rrdtool info <rrd-file>". See *rrd_tool.h* for the type definition.

printRRDinforec Convenience function that prints out the info structure to stdout.

getRRDLast Returns the timecode when the underlying RRD file was last updated. The output is equivalent to "rrdtool last <rrdfile>".

createRRDFile Creates a new RRD file that gets store to disk.

updateRRD Inserts fresh data into the RRD.

fetchSingleRRDValue Fetches one single value from an RRD at the given timecode.

freeRRDDataLine Convenience function that frees all memory occupied by the resulting structure of a fetchSingleRRDValue call.

E File formats

There are two file formats used in Horus: The NetFlow binary format, referred to as *"NetFlow files"* and the *intermediate format* as defined and used by Horus.

E.1 NetFlow File format

The NetFlow file format is simply the payload data of the UDP packets stored on disk. There is no conversion of the data. The specification of the NetFlow format can be found at Cisco's NetFlow page [15]. Each packet contains a flow header, followed by a number of flow records. The exact number of flow records can be read out of the header's count field.

E.2 Intermediate File format

The intermediate file format as used and defined by Horus is a text file. Each line starting with a "#" mark is considered a comment and ignored. The first non-comment line must contain a specially formatted entry in the form of:

```
T:<number of counters>:<counter name 1>:<counter name 2>:...
```

Where T is the literal character "T" and the fields are separated by colons. The second field contains a number specifying the amount of identifiers following. Afterwards the names of the counters are listed with the exact same string as used to register an entry in the bucket-list as detailed in Section 5.3.2. This allows Horus to assign the different counters to the correct Modules even if the order is not the same or some of the fields are not used because the corresponding Module is disabled in the current configuration.

Following this special line are all data lines, starting with the timecode in epoch format and all the specified counters, divided by a colon character. The following example shows the first few lines of a valid file in intermediate format:

```
## output from horus
# field 1 is timestamp in epoch format
# field 2 is byte-counter
# field 3 is packet-counter
# field 4 is flow counter
T:3:byte-counter:packet-counter:flow counter
1059696960:29515:21:1
1059697020:5048435:10346:732
1059697080:7972505:26889:2900
1059697140:8174508:27471:2698
1059697200:11208744:35404:3399
1059697260:5660822:24393:2771
```

The format is easy enough to manipulate the data from hand or by the use of small scripts. Also concatenating several intermediate files into one single file can be done without complex processing.

F Original problem description

Following is a copy of the original task description for this diploma thesis.

F.1 Introduction

The problem

Internet attacks such as massive worm spreading events or denial of service attacks have increased in frequency and impact over the last few years. Attacks that involve several thousand hosts are a reality today.

The setting

In the context of the project DDoSVaX[5] a long-term (>1 year) archive of flow-level Internet backbone border traffic data in the form of Cisco NetFlow v5 records was established. In several offline analyses of Internet attacks, characteristic patterns that possibly could be used for online detection were obtained. An online traffic data analysis framework named UPFrame and several near-realtime plugins were already developed and successfully used to detect worm outbreaks.

F.2 The Task

The usage patterns of certain protocols (such as the amount of e-mail traffic in bytes per hour) show a clearly visible daily rhythm. In case of an attack, this rhythm is disturbed.
The task of the student is split in four major subtasks.

Related Work

A search for publications about algorithms that can be used to track and model seasonal or rhythmic changes in somewhat noisy measurements against a time-line is conducted. Inputs from research about technical measurements in various fields (medicine, weather, network traffic, electricity etc.) and also established economic algorithms (stock market etc.) should be considered.

Specification of anomaly detection algorithms

A small set of promising algorithms for tracking seasonal changes in noisy measurements will be selected, adjusted, extended and/or newly developed. An already existing robust algorithm developed by T. Dübendorfer that detects relative minima and maxima in a noisy signal will also be studied and extended for seasonal anomaly detection.
For each algorithm a specification is written and the algorithm's strengths and limitations are analysed and described. The algorithms should be flexible enough to allow for tracking measurement values and fluctuations at different orders of magnitudes. Anomalies that must be detected are spikes (momentary high values), surges (prolonged high values), faults (momentary blackout), blackouts, sags (momentary low values) and brownouts (prolonged low values). In addition, the seasonal behaviour and deviations from "normal" must be characterized with parameters that are easy to interpret and understand.
The algorithms will be evaluated and tested on network traffic of a restricted set of specific services (e.g. SMTP, HTTP). They must be flexible enough to recognize a "normal" seasonal traffic rhythm and adapt to it. After an adaptation or learning phase, they must be able to detect anomalies in the network traffic of such a service.

UPFrame plugin for NetFlow data

After familiarizing with the DDoSVax cluster "Scylla", the archived CISCO NetFlow traffic data and "UPFrame", a near-realtime plugin will be developed that incorporates the specified algorithms.
For efficient measurements and tests, the plugin must support reading input data through the UPFrame framework as well as directly from the file system. The output produced by the plugin

[5] See `http://www.tik.ee.ethz.ch/~ddosvax/`

in the form of log files will be further processed by independent tools that interact with the user of the plugin. For graphical plots, tools such as gnuplot or RRD can be used. The developed plugin will be released under GPL.
As the plugin will run under tight resource constraints (CPU, RAM) in near real-time, special attention will have to be paid to efficient resource use. It must be possible to restrict the plugin to a fixed amount of RAM usage.

Validation and Parametrization

It is important to reduce false positives in anomaly detection. Therefore a large set of measurements will be carried out with archived DDoSVax NetFlow data. Special attention will have to be paid to outbreak phases of e-mail worms such as Sobig.F or Mydoom.A. A cross-validation on "normal" traffic without outbreaks of larger worms or large network incidents is also vital. This will help to validate, parametrize and optimize the algorithms.

F.3 Deliverables

The following results are expected:

1. *Related work survey* A short but precise survey that explains current approaches in the field of tracking and modelling measurements exhibiting seasonal behaviour.
2. *Specification of various seasonal algorithms* A specification and analysis of at least three different algorithms that proved useful for tracking seasonal measurements and detecting anomalies in the network traffic of a restricted set of specific services (e.g. SMTP, HTTP).
3. *UPFrame plugin* A near-realtime implementation of the seasonal anomaly detection algorithms. The code should be documented well enough such that it can be extended by another developer within reasonable time.
4. *Diploma thesis documentation* A concise description of the work conducted in this thesis (task, related work, environment, algorithms, measurements, results and outlook). The survey and the specification of the algorithms ms are also part of this main documentation.

Further optional components are:

- Alarming mechanisms for important anomalies detected
- Further analysis of the real cause of detected anomalies
- Generalizations of the algorithms used

Presentations and Dates

There will be one informal intermediate presentation before the end of the first half of the total thesis duration. At the end of the thesis, a presentation will have to be given at TIK that states the core tasks and results of this thesis. If important new research results are found, a paper might be written as an extract of the thesis and submitted to a computer network and security conference. This diploma thesis starts on October 25th, 2004 and is finished on February 24th, 2005. It lasts 4 months in total.

Contacts

Tutor: Thomas Dübendorfer, duebendorfer@tik.ee.ethz.ch, +41 1 632 71 9 6, ETZ G95
Co-Tutor: Arno Wagner, wagner@tik.ee.ethz.ch, +41 1 632 70 04, ETZ G95
Supervisor: Bernhard Plattner, plattner@tik.ee.ethz.ch, +41 1 632 70 00 , ETZ G89

References

[1] "Converged Data Networks", The ATM Forum, http://www.atmforum.com/aboutatm/cdn.html

[2] The Spread of the Sapphire/Slammer Worm, http://www.caida.org/outreach/papers/2003/sapphire/sapphire.html

[3] "Botnet attacks increase in October", SecureWorks, http://www.secureworks.com/techResourceCenter/octoberattackchart.html

[4] "Zombie networks fuel cybercrime", Computer Crime Research Center, http://www.crime-research.org/news/04.11.2004/767/

[5] Lukas Ruf, *Latex Essentials – HowTo Create Your LaTeX-based Documentation*, TIK, ETH Zuerich, 2002.

[6] J. Brutlag: USENIX, Proceedings of the 14th Systems Administration Conference (LISA 2000).

[7] P. Barford, J. Kline, D. Plonka, and A. Ron, "A Signal Analysis of Network Traffic Anomalies" in *ACM SIGCOMM Internet Measurement Workshop*, (Marseilles, France), Nov., 2002.

[8] Internet Anomaly Detection Resource Page, http://www-personal.engin.umich.edu/~npatwari/iadrp.html

[9] "Snort - the de facto standard for intrusion detection/prevention", http://www.snort.org

[10] Paxson, Vern, "Bro: A System for Detecting Network Intruders in Real-Time", Lawrence Berkeley National Laboratory Proceedings, 7'th USENIX Security Symposium, (San Antonio TX) Jan. 1998

[11] Mahoney, M., P.K. Chan, "PHAD: Packet Header Anomaly Detection for Identifying Hostile Network Traffic", Florida Tech. technical report 2002-08, http://cs.fit.edu/$\sim$tr/

[12] Mahoney, M., P.K. Chan, "Learning Nonstationary Models of Normal Network Traffic for Detecting Novel Attacks", Edmonton, Alberta: Proc. SIGKDD, 2002, 376-385

[13] Mahoney, M. "Network Traffic Anomaly Detection Based on Packet Bytes", Proceedings of the 2003 ACM symposium on Applied computing, March 2003, Melbourne, Florida

[14] MSN TV, http://www.webtv.com

[15] "Cisco IOS Software NetFlow", Cisco Systems, http://www.cisco.com/warp/public/732/Tech/netflow/

[16] RRD Tool Homepage, http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/

[17] UPFrame Homepage, http://www.tik.ee.ethz.ch/~ddosvax/upframe/

[18] "A look into IDS/Snort Whole thing by QoD", AntiOnline, http://www.antionline.com/showthread.php?s=&threadid=253920

[19] C. C Holt (1957), "Forecasting seasonals and trends by exponentially weighted moving averages", ONR Research Memorandum, Carnigie Institute 52. P. R. Winters (1960) "Forecasting sales by exponentially weighted moving averages", Management Science 6

[20] Yiming Gong, http://www.securityfocus.com/infocus/1796, http://www.securityfocus.com/infocus/1802

[21] A. Lakhina, K. Papagiannaki, M. Crovella, C. Diot, E.D. Kolaczyk, and N. Taft "Structural Analysis of Network Traffic Flows". In ACM SIGMETRICS, 2004

[22] A. Lakhina, M. Crovella, C. Diot "Diagnosing Network-Wide Traffic Anomalies" SIGCOMM, 2004

[23] M. Crovella and E. Kolaczyk, "Graph Wavelets for Spatial Traffic Analysis". In IEEE INFOCOM, 2003.

[24] M. Roughan, T. Griffin, Z.M. Mao, A. Greenberg, B. Freeman, "IP Forwarding Anomalies and Improving their Detection Using Multiple Data Sources", SIGCOMM'04 Workshops

[25] "What is exponential Smoothing?", Engineering Statistics Handbook, http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc43.htm

[26] Dübendorfer, T. 1999. "RespiG" Messumgebung für Atemmuster. Arbeitsgruppe Arbeit + Gesundheit. IHA, ETH Zürich

[27] Encyclopedia Mythica: Egyptian mythology, http://www.pantheon.org/areas/mythology/africa/egyptian/articles.html

[28] Dr. Zahi Hawass, "Horus The Falcon God", http://www.guardians.net/hawass/horus.htm

[29] Encyclopedia Mythica, http://www.pantheon.org/areas/gallery/mythology/africa/egyptian/horus.html