



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich



Institut für  
Technische Informatik und  
Kommunikationsnetze

Stefan Keller

# Entropy Plug-in and Visualisation Tool

Semester Thesis SA-2006-25  
April 2006 to July 2006

Supervisor: Arno Wagner  
Co-Supervisor: Bernhard Tellenbach  
Professor: Bernhard Plattner

## Abstract

The thesis is part of the DDosVax project which has the motivation to detect distributed denial of service (DDoS) attacks and to develop some countermeasures against them. Therefore, this thesis has the goal to implement a tool-set for near real-time processing which visualises the characteristic of the Internet traffic that is forwarded by the border gateway routers of the Swiss Education and Research Network SWITCH. The characteristic is determined by an estimation of the entropy of the NetFlow what is calculated by compression algorithms like bzip2 and lzo1x-1. The entropy gives information about the Internet traffic behaviour what means that the characteristic of a worm outbreak could be detected because it has an abnormal behaviour in comparison to the normal Internet traffic. The calculation of the entropy is implemented in several ways but every method results in the same characteristic. That is important because a very efficient method, named iterative compression, is implemented that calculates the entropy very fast. Currently, five per cent of the Internet traffic in Switzerland and crossing Switzerland is forwarded by the border gateway routers. If that number of included NetFlow traffic increases, there will be the possibility to involve much more NetFlow data and these can still be processed by the tool if iterative compression is used.

For the visualisation of the entropy, a Common Gateway Interface (CGI) program is implemented that also efficiently generates the plots in real-time and visualises them in a web browser.

The whole tool-set is a very efficient and fast processing system that is not restricted for worm detection only. Every anomaly of the NetFlow is detectable that is an effect of different distribution in the Internet traffic.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Task Description . . . . .	2
1.3	Overview . . . . .	2
1.4	Related Work . . . . .	3
<b>2</b>	<b>Entropy</b>	<b>4</b>
2.1	Definition . . . . .	4
2.2	Compression Algorithms . . . . .	5
2.3	Compression Variants for the Plug-in . . . . .	6
2.3.1	Block Mode vs. Streaming Mode . . . . .	6
2.3.2	Sliding Window . . . . .	6
2.3.3	Iterative Compression . . . . .	7
<b>3</b>	<b>Entropy Plug-in</b>	<b>8</b>
3.1	Collect the Data . . . . .	9
3.1.1	Bit Mask . . . . .	10
3.1.2	Hash Table Size . . . . .	10
3.2	Time Analysis . . . . .	11
3.2.1	Maximum Flow Time . . . . .	11
3.2.2	Time Constrains . . . . .	12
3.3	Processing . . . . .	13
<b>4</b>	<b>Visualisation Tool</b>	<b>15</b>
4.1	The Normalized Entropy in a Time Plot . . . . .	15
4.2	CGI Script . . . . .	16
4.2.1	Processing a Request . . . . .	16
4.2.2	”Life Monitoring” . . . . .	17
4.2.3	Zooming . . . . .	17
<b>5</b>	<b>Validation</b>	<b>19</b>
5.1	Compressors in Comparison . . . . .	19
5.1.1	Compression Algorithm . . . . .	20
5.1.2	Block Compression vs. Iterative Compression . . . . .	20
5.1.3	Changes in Block Size . . . . .	22
5.2	Distinction of the Transport Protocols . . . . .	25
5.3	Conclusion . . . . .	26
5.4	Future Work . . . . .	28

---

5.5 Acknowledgement . . . . .	28
<b>A Installation of the Plug-in</b>	<b>29</b>
A.1 Requirements . . . . .	29
<b>B Configure the Plug-in</b>	<b>30</b>
B.1 Argument List . . . . .	30
B.2 The Config File . . . . .	31
B.3 Converter . . . . .	31
<b>C Installation of CGI Scripts</b>	<b>33</b>
<b>D Entropy Plots</b>	<b>35</b>

# List of Figures

2.1	The usage of a sliding window. . . . .	7
2.2	Iterative compression with sliding window. . . . .	7
3.1	The Plug-in reads the data from a buffer of UPFrame which stores the UDP stream of SWITCH for a short time. . . . .	8
3.2	The hash table that contains an array filled with several lists. . .	10
3.3	The binary representation of the flow start time in UNIX time is logically ANDed with a bit mask of eight ones and twelve zeros. After a shift operation, the result is the position of a list where the record with that time must be added. . . . .	11
3.4	In best-case, a flow with maximum flow time starts before an export time. In that case the delay is 15min. . . . .	12
3.5	In worst-case, a flow with maximum flow time starts immediately after an export time. In that case the delay is 30min. . . . .	12
3.6	Time constrains . . . . .	13
3.7	Data flow graph of the plug-in. . . . .	14
4.1	Overview . . . . .	15
4.2	Data flow of the CGI interface. . . . .	17
4.3	Zoom in with the mouse. . . . .	18
5.1	Block compressed by lzo1x-1 with a block size of 5min and a sliding window size of 1min. The plots are normalized. That means that the size of each compressed block is divided by its uncompressed block size. . . . .	21
5.2	Iterative compressed by lzo1x-1 with a block size of 5min and a sliding window size of 1min. The plots are normalized. That means that the size of each compressed fragment is divided by its uncompressed fragment size. . . . .	21
5.3	Block compressed by bzip2 with a block size of 5min and a sliding window size of 1min. The plots are normalized too. That means that the size of each compressed block is divided by its uncompressed block size. . . . .	23
5.4	Iterative compressed by bzip2 with a block size of 5min and a sliding window size of 1min. The plots are normalized. That means that the size of each compressed fragment is divided by its uncompressed fragment size. . . . .	23

---

5.5	Block compressed by lzo1x-1 with a block size of 10min and a sliding window size of 2min. The plots are normalized. That means that the size of each compressed block is divided by its uncompressed block size. . . . .	24
5.6	Block compressed by bzip2 with a block size of 3min and a sliding window size of 1min. The plots are normalized. That means that the size of each compressed block is divided by its uncompressed block size. . . . .	25
5.7	Witty with no distinction of the used transport protocol. . . . .	26
5.8	The characteristic of the UDP traffic during the Witty outbreak. . . . .	27
D.1	Block compressed by lzo1x-1 with a block size of 5min and a sliding window size of 1min. . . . .	36
D.2	Iterative compressed by bzip2 with a block size of 5min and a sliding window size of 1min. . . . .	37
D.3	The characteristic of the UDP traffic during the Witty outbreak calculated with lzo1x-1 using iterative compression. . . . .	38

# List of Tables

5.1	Compression speed of lzo1x-1 and bzip2. . . . .	20
5.2	The relative variation of block compression to iterative compression using lzo1x-1. . . . .	22
5.3	The relative variation of block compression to iterative compression using bzip2. . . . .	24
5.4	The relative variation of block compression with a block size of 5min and 10min using lzo1x-1. . . . .	25
5.5	The relative variation of block compression with a block size of 5min and 3min using bzip2. . . . .	26
B.1	Argument list of the plug-in . . . . .	31

# Chapter 1

## Introduction

### 1.1 Motivation

For a long time, distributed denial of service (DDoS) attacks were not known by the most people and commercial organisations. There were a few non-commercial establishments, like the ETH, that became a victim of such an attack but that was not of a public interest.

Around the year 2000, DDoS attacks have become an important threat to Internet services after such attacks damaged businesses like ebay.com or amazon.com. Thus, the interest was coming up to analyse and understand these attack methods with the goal to detect them and to protect the attacked organisations for reducing the damage.

**DDoSVax project** As a consequence of the DDoS attacks and the increasing public interest, the ETH created a project with the operator of the Swiss Education and Research Network SWITCH<sup>1</sup>. The name of this project is DDoSVax [2] that has started the research with the goal to develop detection algorithm and countermeasures against DDoS attacks. This cooperation allows the researchers at the ETH to analyse real flow-level Internet traffic data, that is forwarded by the four SWITCH border gateway routers. The collected statistical traffic data of each flow are send to the ETH in more or less periodical time intervals using Cisco NetFlow V5 [3]. Because of the data protecting act, the NetFlow V5 packets contain only reduced information of a flow, whereby the methods for analysing the traffic become more complicated.

**This Thesis** A detection of an anomaly in the NetFlow data, received by the SWITCH border gateway routers, would be a sign that there may be same bad traffic. For detecting the anomaly, there has been found a way to calculate the distribution of IP addresses and port numbers that is called entropy. As part of the DDoSVax project this thesis has the subject to calculate and to visualize the entropy of the collected NetFlow data in near real-time. The visualised entropy can be analysed by human and should give the information if an Internet worm has outbroken.

---

<sup>1</sup>See <http://www.switch.ch/>



## 1.2 Task Description

A real-time UDP NetFlow processing framework called UPFrame [4] was developed in a diploma thesis [5], that collects the exported NetFlow data of the border gateway routers and forwards them to data processing plug-ins. These plug-ins are all independent of each other and were controlled by a watchdog process of UPFrame.

The task for this thesis is split into three major subtasks: Entropy Plug-in, Visualization and Validation.

**Entropy Plug-in** In a first step there is to implement a near real-time plug-in for UPFrame, that calculates the entropy of the collected data. It must satisfy some time constraints and it should be possible to include all received NetFlow data what means that no NetFlow packet should get lost in the UPFrame buffer. Therefore, data processing must be implemented in an efficient way because the plug-in has to collect a huge number of data. The plug-in includes several algorithms with different behaviour for calculating the entropy.

**Visualisation** The second step is to implement a visualisation tool that generates a picture of the entropy over a specified time range. The request for a picture comes from a web client over a secure SSL connection. Therefore, it must be implemented a (Common Gateway Interface) CGI script that processes the web requests and plots the entropy of that time range in real-time.

**Validation** At the end, there has to be done some tests to validate which alternatives are the favorites for calculating the entropy. The used criteria are efficiency of the entropy plug-in and the results of the generated pictures. Therefore, several plots are compared and the advantages and disadvantages are discussed.

## 1.3 Overview

The structure of this thesis is the following:

In chapter 2 there is defined what entropy is and which algorithms are used to estimate the entropy. It also describes the two methods that are used to determine the entropy with the included algorithms and how these methods are applied to a NetFlow data sequence. Chapter 3 shows how the entropy plug-in works and what has been done for an efficient processing of the buffered NetFlow data. It describes the use of a hash table for collecting the relevant NetFlow data that must be sorted after the start time of the flows. The chapter also discusses the time constraints that the plug-in must include for the decision when to compress the collected data. In chapter 4 the CGI interface is explained with all its features. The validation is discussed in chapter 5 that shows the characteristic of the Blaster Worm and the Witty Worm with the calculated entropy.

## 1.4 Related Work

The paper 'Entropy Based Worm and Anomaly Detection in Fast IP Networks' [1] by Arno Wagner describes the method to calculate the entropy with compression algorithms like bzip2, lzo1x-1 and gzip. It is illustrated how the characteristic of a worm outbreak is detectable using the entropy of the Net-Flow.

In a diploma thesis with the title 'Near Real-Time Detection of Traffic Usage Rhythm Anomalies in the Backbone' [8] by René Gallati is discussed how to detect a worm-spread that is sent by e-mail. Therefore, the e-mail traffic is analysed and a daily rhythm of that traffic is included in the detection algorithm.

In another diploma thesis 'Scan Detection Based Identification of Worm-Infected Hosts' [7] there is described how a worm scan detection can be integrated in an intrusion detection system. This method is thought for office networks for a fast reaction on a worm infection.

## Chapter 2

# Entropy

The goal is to detect an outbreak of a worm with the statistical NetFlow data. The problem with detecting a fast Internet worm in near real-time from an observed traffic is, that there is a large traffic but the traffic of the worm is only a small fraction on the whole. Thus, there must be a method that can show the small changes in a network traffic - changes that could be made by a worm. But what information are in the Cisco NetFlow V5 packets that could help to detect such an outbreak?

Because there are no information about the data of an IP packet, there must be another way to detect it. The question is, what will distinguish normal traffic from a traffic of a worm. The differences are in the distribution of the source IP address, destination IP address, source port and destination port. Normal Internet traffic will have a special characteristic what will be shown in chapter 5 with the entropy of these four parameters. If there is an abnormal traffic included it will change the characteristic in an obvious way. A worm outbreak has a behaviour that is very different in comparison with the normal Internet traffic because a worm will normally scan an IP address range to infect some weakly protected hosts. Thus, one or a few IP addresses will send many packets with different destination IP addresses. There is also a new and abnormal characteristic in the use of the ports. The attacker may scan an IP address range on a special or on several destination ports and the same is recognizable for the used source ports. Furthermore, an infected host may change its Internet traffic behaviour and tries to infect other hosts. For more details about worm traffic characteristic see [3], [6] and [7].

Thus, the calculation of the entropy is a way to analyse the distribution of the NetFlow traffic and their changes over the time.

### 2.1 Definition

Entropy is a measure how random a given data-set is. The more random a sequence is, the larger is its entropy. In [3] the entropy is estimated by a data compression algorithm. It only estimates the entropy, because data compression algorithms are not perfect in a mathematical sense, but the estimated entropy is good enough for analysing the Internet traffic.

The entropy is not determined for every packet that is forwarded by the border

gateway routers, but it is determined for every flow. A flow is defined by the source/destination IP address/ports and some other information that are stored in the Cisco NetFlow V5 packets [3]. That is also the reason why worm outbreaks has such an influence on the resulted entropy since a worm outbreak will have a huge number of flows but the number of transfered packets and the number of transfered data is negligible in comparison with the normal Internet traffic.

## 2.2 Compression Algorithms

There are many compression algorithms and every one has its own behaviour what in general will result in a trade-off between compression speed and compression ratio. In [3] the used compression algorithms are:

1. bzip2 (libbz2-1.0)
2. gzip (zlib1g 1.2.1.1-3)
3. lzo1x-1 (liblzo1 1.08-1)

This three algorithms are used in this thesis too and for more flexibility it is extended by lzo1x-999.

With this variety the user can freely choose his favourite or can do situation-oriented choice.

All of this four has its advantages and disadvantages that are explained below.

**bzip2** This compression algorithm has a very good compression ratio and the ratio between worst-case and average-case compression time is in the region of 10:1 - for the versions 0.9.5 and above [9]. It is based on the Burrows-Wheeler block sorting text compression algorithm and on Huffman coding [3]. The internal block size is the number of bytes that will be compressed together. If more data is available it starts a new block and compresses it independently of the other blocks. An interesting point is that the block size is variable and can be set between 100 and 900 kilobytes and the allocated memory  $m_{bzip2}$  for compression is determined for every block by the formula:

$$m_{bzip2} = 400kB + 8 * blockSize. \quad (2.1)$$

The block size can influence the compression ratio what means that a larger block size could have a better compression ratio than a smaller one but the difference in the compression time should be negligible.

The drawback of bzip2 is its long compression time and in comparison with the other compressors it needs more memory for compression.

**gzip** This compression algorithm is the GNU zip compressor and is based on the Lempel-Ziv method that still has a good compression ratio (not so good like bzip2) and it is more than three times faster than bzip2. In contrary to bzip2 it has a fix block size of 256kB what will also result in a fix determined memory usage [11] [3].

**lzo1x-1** The Lempel-Ziv-Oberhumer compressor is the fastest used algorithm. It was especially made for real-time application and is nearly 25 times faster than bzip2 and still nearly ten times faster than gzip. But as consequence of its fastness it has a worse compression ratio and uses only a block size of 64kB [10].

**lzo1x-999** This compression algorithm is of the same family as lzo1x-1 but it has a compression ratio and a compression speed like gzip. It has also a block size of 256kB that is not changeable [10].

## 2.3 Compression Variants for the Plug-in

The discussion above shows how entropy is defined and what kind of compression algorithms are used. Now, there is the question how to compress the four values (source/destination IP address and source/destination port) without including undesired effects. Undesired effects are short-term effects which are dominant for a short time but is not really an anomaly in the NetFlow traffic. A short-term effect is for example when several hosts try to connect the same server what means that many different source IP addresses contact the same destination IP address on the same destination port. That can change the characteristic of the NetFlow but because it is no attack, the effect is vanished after a short time.

### 2.3.1 Block Mode vs. Streaming Mode

In general one can distinguish two different compressing mode:

1. **Block Mode:** In this mode a large block of data is compressed at once in one step. A block can be defined as all source IP address with a start flow time in the interval  $[t_0, t_1]$ , as an example.
2. **Streaming Mode:** In this mode the data is processed at the time it is read. In practice it is like block mode with smaller block size.

The advantage of block mode is that its compressed result is over a longer time period what means that more values are integrated for the entropy. This fact reduces wrong results in case of short-term effects. The advantage of a longer time interval is also its biggest disadvantage: The entropy plug-in needs to collect more data before it can compress them what has the consequence that more memory is used and more time for processing and compressing the data is needed.

Streaming mode can not be used because it detects every small changes. That means that all short-term effects will be included what is not desired for detecting a worm which results in a long-time behaviour.

So there is a trade-off between the needed processing and compression time and a not sensitivity of short-term effects in the estimated entropy.

### 2.3.2 Sliding Window

Because of the interest using block mode for calculating the entropy it is used in combination with a so called sliding window. In figure 2.1 is illustrated how

the concept with the sliding window works.

After compressing the first block of all flows within the interval  $[t_0, t_1]$  a part of that block is still used for the next block in  $[t'_0, t'_1]$ . Thus, only a small piece of the beginning part is neglected (one sliding window) in the next block and only a small new part is added. The big advantage is that the sensitivity of short-term effects is extremely reduced because only a small part is new and therefore, it can not be the dominant part. So only if the effect is present over a longer time it will be introduced in the estimated entropy.

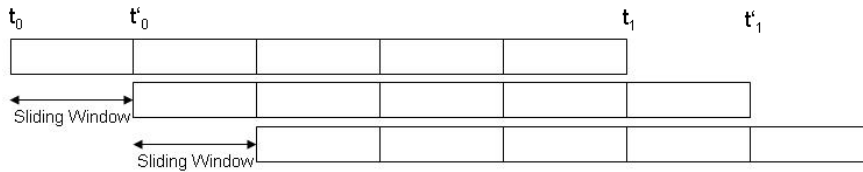


Figure 2.1: The usage of a sliding window.

But this method has also a drawback: Although the block size can be reduced by using sliding window it is not a very efficient way, because every piece is compressed several times in different blocks. How often every flow is compressed depends on the used block and sliding window size. In general, the block size is a multiple of the sliding window size with the consequence that every flow is compressed as often as the sliding window finds place in one block.

### 2.3.3 Iterative Compression

The combination of block mode and sliding window is one concept how the entropy is determined. But because of the drawback of additional compression times, there is an alternative that will need much less time.

Iterative compressing splits a block in several fragments, compresses them independently and adds the results together. Each fragment is of the size of a sliding window what has the advantage that the fragments can be reused for the next block. That will work very fast and seems to be a good choice for a near real-time application. But iterative compression has the question if the sum of the compressed fragments will be much larger than the value of a compressed block and if short-term effects have more influence on the resulted entropy. These questions are discussed in chapter 5.

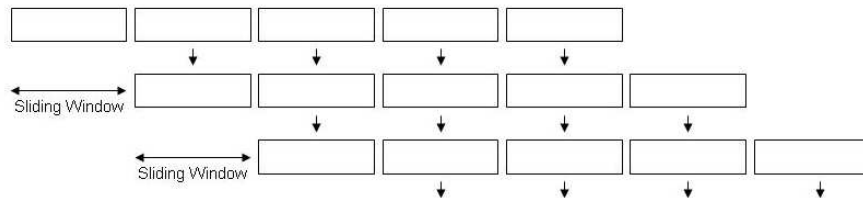


Figure 2.2: Iterative compression with sliding window.

## Chapter 3

# Entropy Plug-in

This chapter explains how the entropy plug-in works and what the difficult parts are. The goal is to implement a plug-in that calculates the entropy of the NetFlow data in near real-time. It is only possible in near real-time because the NetFlow packets are not sent immediately by the router what means that there is always a small delay. In general, the routers export the NetFlow data every 15 minutes, but there is the possibility that they export them before that time if their local memory is full. In practise there is shown that there can also be some delay until the routers export the NetFlow data. It is also possible that a started flow can not be exported if it has not finished yet.

But as soon as some NetFlow data are available by the plug-in, it starts with processing. It reads the data from a shared memory that is managed by UPFrame which buffers the NetFlow packets for a short time. (Exactly: How long the packets are buffered is not defined in time but the buffer has a fix size which is configureable. For this thesis the default value is used.)

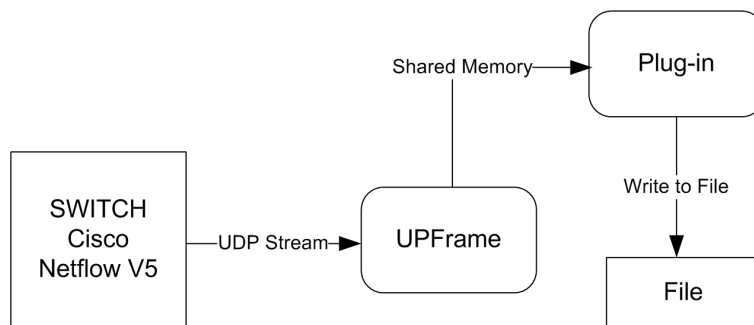


Figure 3.1: The Plug-in reads the data from a buffer of UPFrame which stores the UDP stream of SWITCH for a short time.

If the plug-in is used with its default parameter, the block size will contain all flows with a start time within five minutes and the sliding window will be one minute. Then the plug-in must collect about 1,000,000 NetFlow records to fill a whole block and in iterative compression it must collect about 200,000 NetFlow records to fill one fragment. These numbers vary and there were peak

values that reach 2,000,000 NetFlow records for one block.

Thus, the plug-in should collect and manage the data in a very efficient way with the aim to read all data from the buffer and no packet gets lost. That does not mean that all packets must be included for the estimated entropy of the NetFlow, but it should be possible.

Keep in mind that the plug-in supports two different possibilities to include the entropy of the NetFlow in its calculation.

- It calculates the entropy of the NetFlow with no distinction of the used transport protocol.
- It calculates the entropy of the NetFlow of TCP, UDP, ICMP and all other protocols separately.

The reason, why the second item is implemented, is that there exists some worm attacks which do not change the characteristic of the NetFlow traffic significantly. One example, as discussed in chapter 5, is the Witty Worm who uses UDP as transport protocol. Its influence is almost not detectable with no distinction of the used transport protocol and its special characteristic is much better visible if only the entropy plot of all UDP flows is analysed.

For simplicity, the following description is focused on processing the data with no distinction of the used transport protocols because the essential schedule is the same for both cases.

## 3.1 Collect the Data

The first step, when new NetFlow packets arrive, is to read the data and to store the relevant information. The information that must be stored and builds a record are:

- source IP address
- destination IP address
- source port
- destination port
- flow start time

The first four items of the record are needed to calculate the entropy of these parameters as described in chapter 2. The last item is used to sort the data that depends on the start time of the flow. That means before the plug-in can compress the data it needs to sort them in increasing order in condition of the start time.

An issue is that there is to collect a huge number of data before the plug-in can compress them. The goal is to reduce the time to sort this data sequence that will be too large if one list is used. It seems to be a good choice to use several lists to collect the data rather than one because the time for sorting increases exponentially with the number of elements. Therefore, a hash table is implemented.



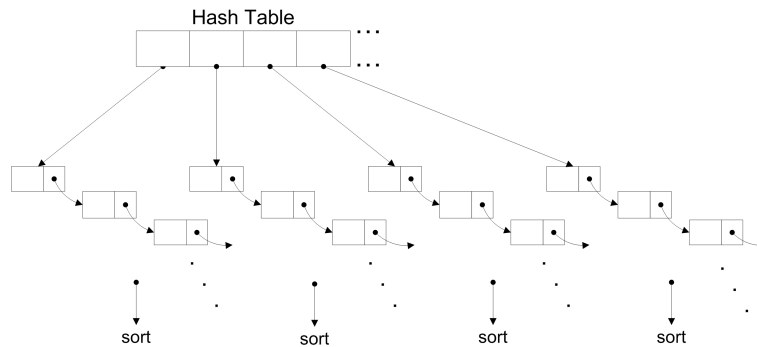


Figure 3.2: The hash table that contains an array filled with several lists.

The result is a table with some lists, where every list is filled by data within a specified time range. The decision to which list a record has to be added is made by a bit mask. If the time has arrived for compressing the data, the hash table itself must be sorted first and in a second step every list must be sorted independently of each other.

### 3.1.1 Bit Mask

Over a bit mask it is decided to which list a record must be added what contains two bit operations: a logical AND and a shift operation.

The bit mask has several ones at the most significant bits (MSB) and several zeros at the least significant bits (LSB). The following numbers are used as default values but they are all configurable to avoid every kind of restrictions. The hash table contains 511 lists where every list collects data of a time range of about four seconds. The relevant record entry is the flow start time which contains the number of milliseconds since 1. January 1970 at 00:00 (UNIX time) and is a 64 bit value. Thus, the bit mask contains twelve zeros at LSB and eight ones at MSB because at position twelve there is a resolution of about four seconds ( $2^{12} = 4096ms$ ). The eight ones are used to generate the hash by a logical AND operation with the start time of the flow. The resulted hash must be shifted to right by twelve bits and the output is the list number where that record must be added. An example shows the operations in figure 3.3.

With the use of a hash table the time for sorting can be reduced by a factor of ten in comparison to a collection with one list only.

### 3.1.2 Hash Table Size

With the hash table, the plug-in has a fast algorithm to manage its stored data, but there is a time condition that must be hold because there is a possibility that data are added to a list that is not responsible for that time range - not yet.

The lists, as explained above, contains all flows with a start time within four seconds and there are 511 lists. Thus, the hash table can collect data of  $511 * 4096 = 2093056ms \simeq 34.88min$ . With more than  $30min$  the hash table

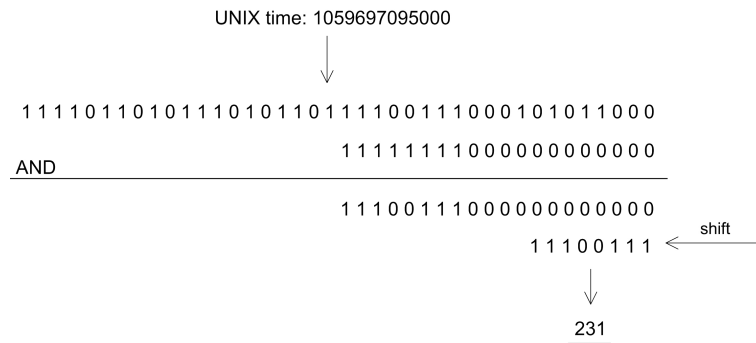


Figure 3.3: The binary representation of the flow start time in UNIX time is logically ANDed with a bit mask of eight ones and twelve zeros. After a shift operation, the result is the position of a list where the record with that time must be added.

is large enough to include all NetFlow data as discussed in the next section. But if the hash table contains 127 lists only, it can collect data of 8.7min what will be a problem if more data are collected before compression. Because if a NetFlow packet arrives that contains a flow with a start flow time that is out of that range of 8.7min, it will be added to a list that is not responsible for that time range.

Thus, one must keep care of the size of the hash table but in general a size of 511 will be the right choice. With that size there are no flow time problems and the time for sorting is very fast.

## 3.2 Time Analysis

Another issue is that data can arrive which have a flow start time that is not in the range of the actual compression block. This is a problem if its flow start time is too old and is part of a block that has already been compressed. In that case the flow must be neglected. But there are some restrictions how long a flow can be in maximum.

### 3.2.1 Maximum Flow Time

The router exports all flows after they has terminated. But sometimes there are some flows which have a very long flow time, whereby the maximum flow time is approximately 15min. The question now is what the maximum delay can be of such a flow until it is exported.

The best-case is very simple: If a flow starts immediately before the router has exported and ends exactly before the router exports again. Then the delay will be 15min what is shown in figure 3.4.

In the worst-case, the delay can be 30min. That happens when a flow starts immediately after an export time and ends after the next export time, what is illustrated in figure 3.5.

In that situation the router can not export the flow with the second export time because the flow is still active and its flow time is not yet 15min. As

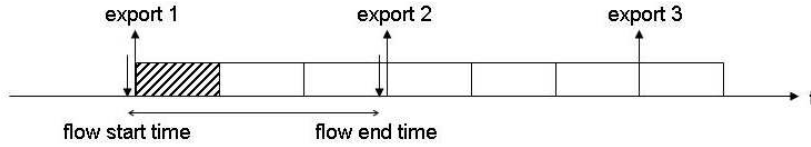


Figure 3.4: In best-case, a flow with maximum flow time starts before an export time. In that case the delay is 15min.

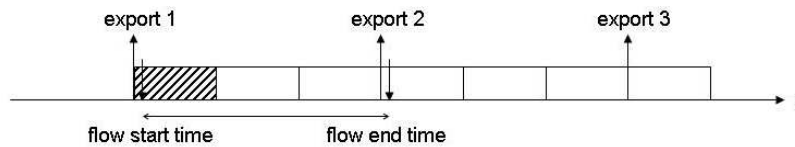


Figure 3.5: In worst-case, a flow with maximum flow time starts immediately after an export time. In that case the delay is 30min.

consequence it must wait the third export time where it will export the flow in every case.

### 3.2.2 Time Constrains

The discussion above has shown that a flow time is restricted by 15min what will result in a maximum delay of 30 min. Thus, some time conditions must be set if the plug-in should include all flows in its calculation.

Therefore, the following time points must be defined:

**setupTime** It is the lower bound and determines what the actual minimum start time of a flow can be. If a flow arrives which has an older start time, it must be neglected.

**deadline** The deadline is the upper bound and defines the time point where the plug-in must compress the collected data. That time has arrived if an export time of a router is later than that deadline.

**compressTime** This time point is the end point of a block that must be compressed. That means that the block size is defined by the interval [setupTime, compressTime]. This interval is different in iterative compression because only one sliding window is compress at the time where in block compression the whole block will be compressed.

**waitTime** A waitTime is included for long flow times and determines the time that the plug-in still must collect some data although the compressTime is already achieved. If it should include all NetFlow packets the waitTime must be of the following size:

$$waitTime > 15 \text{ min} - (compressTime - setupTime) \quad (3.1)$$

This is true because the plug-in first collects the data and compares the export time with the deadline. It does not really matter if it compares the export time before or after the data collection, because the plug-in must wait for the next export time to compare the times. Thus, there is no big time difference if it also collects and includes the arrived NetFlow packets. The plug-in can not do a calculation when the next export time will be because it can be less than 15 min if the local memory of the router is full before that time, as an example.

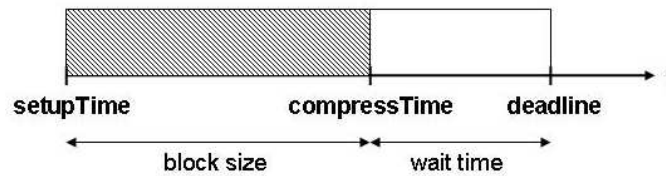


Figure 3.6: Time constraints

So, if the waitTime satisfies the condition (3.1), it is possible to include all NetFlow packets that are exported. But flows with a maximum flow time are very seldom and currently, the flow of a worm has a very short flow time. That means that the waitTime must not satisfy the condition (3.1) and the resulted entropy characteristic of the NetFlow will still be the same. Two different waitTimes were tested during the collection of the Blaster Worm. If the waitTime is set to ten minutes, the plug-in had to neglect about 0.1% of the flow because of a too old start flow time. In the second test the waitTime was set to one minute with the consequence that 1% of the flows had to be neglected.

### 3.3 Processing

The important details of the plug-in are explained in the sections before and now, the whole schedule can be discussed.

Figure 3.7 shows the process from receiving some NetFlow data until they can be stored.

The UPFrame calls a callback function that awakes the plug-in of its wait state. Then, the plug-in begins with collecting the NetFlow data and stores the relevant information in its hash table as long as the time constraints are satisfied. If the constraints are not satisfied any more, the plug-in starts to compress the data after it has sorted them. Finally, it can store the normalized entropy and it can carry on with collecting if some data are still available, otherwise it goes in the wait state.

It is possible that no data are available while the plug-in collects the NetFlow data and therefore, it also goes in the wait state and waits until new data is available. That situation may be reached after the plug-in has read all NetFlow data of one export. Then it must wait for the next export before it can continue with processing.

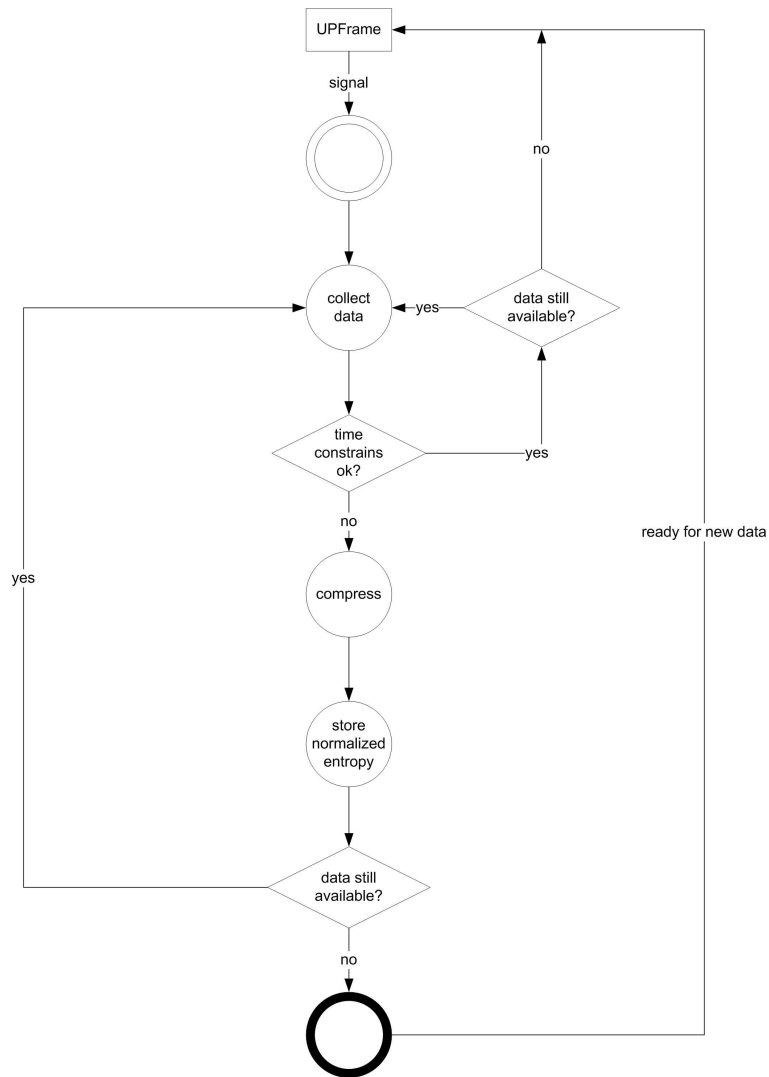


Figure 3.7: Data flow graph of the plug-in.

## Chapter 4

# Visualisation Tool

For plotting the estimated entropy a visualisation tool is implemented that shows the calculated entropy in a time plot. The visualisation tool is encapsulated from the plug-in and is an independent process that uses the generated files with the entropy values of the plug-in. For location independency, the tool can process web requests and the plotted results are shown in a web browser.

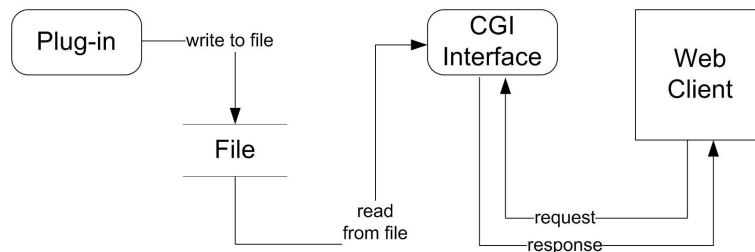


Figure 4.1: Overview

### 4.1 The Normalized Entropy in a Time Plot

The plot shows the entropy over the time that contains the characteristic of the NetFlow. During the calculation of the entropy the values are normalized to a number between zero and one. The normalized entropy is a much better value for plotting because the number of flows changes from one calculated block to another what will result in very different entropy values for every compressed block. Thus, the estimated entropy itself says nothing about the characteristic of the NetFlow traffic because the more data is included in the calculation of the entropy the more entropy the result will contain. That means that there are two facts that can be responsible for the value of the entropy:

- the distribution of the used data and
- the number of data that is included in the calculation.

For making the estimated entropy independent of the second item, it must be normalized by the number of input bytes for the calculation. That will result in the wished characteristic that can be used for analysing.

## 4.2 CGI Script

For processing web requests a CGI script is implemented with the script language Perl. The visualisation tool uses the binary files with the normalized entropy values that are generated by the plug-in and converts them to a well formatted text file that can be read by gnuplot [12]. Gnuplot is a tool that is used to plot the entropy in a time diagramm and stores the plot as a JPG file.

### 4.2.1 Processing a Request

If a client sends a request there are four parameter that are send with it:

- the start time
- the end time
- if the client want to know the most used ports and the number of packets that are send in that time range
- if the client want the distinction between TCP, UDP, ICMP and all other protocols what will result in four plots

Then the CGI script receives the request and starts with processing. The processing part includes three phases for responding a request what is also shown in figure 4.2:

**Phase 1** First, it must collect all files that are needed for that time range. Therefore, the script must translate the start and end time to UNIX time because the file names contain the time range of the stored data and the timestamps in the files are in UNIX time. If no files for this time range are available, the script will return a message to the client that there is no data for the requested time range. Otherwise it continues with processing and generates three file names: one for the picture, one for the gnuplot data file and one for the statistical data file (number of packets and the most used ports). It must generate these file names because the CGI script itself does not create these files but it must include the results of the data content for the web page. The file names contain a sequence of random generated number with the goal that no file get overwritten or appended with new data.

**Phase 2** In this phase, the CGI script executes a program that is written in C and converts the binary files. This is done with the language C because its execution time depends extremly on the requested time range and there must be done many times the same operations what is performed much faster by C. In general, the program reads the binary files and writes two text files: the gnuplot data file and the statistical data file.

**Phase 3** Last, the CGI script executes a bash script with the gnuplot commands that generates the JPG file with the plot of the normalized entropy. The CGI script includes that picture and the statistical data file, if requested, in an HTML page and sends the document back to the client.

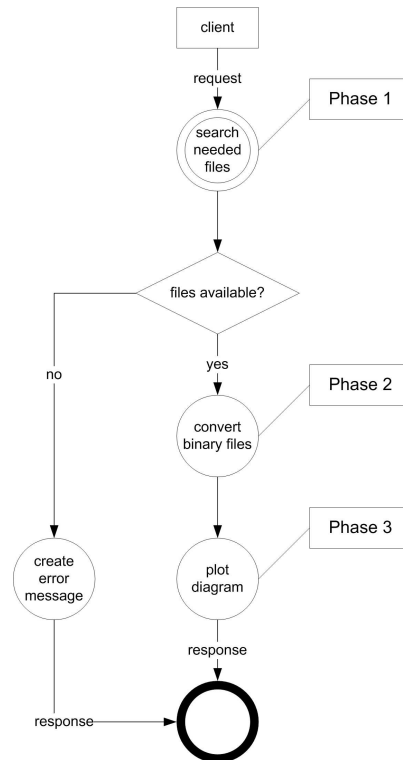


Figure 4.2: Data flow of the CGI interface.

### 4.2.2 "Life Monitoring"

For near real-time analysing there is a possibility to view always the last 24 hours in a "life monitoring" way. The HTML page reloads itself every minute and new values are included - if available. The words "life monitoring" are in double quotes because there is a delay. The delay is still a consequence of the router export times and of the processing time of the plug-in that first has to collect a sequence of data before it can compress them and furthermore, there might be included a wait time what additionally increases the delay.

### 4.2.3 Zooming

If a time range is plotted and the client starts with analysing the resulted characteristic, he may detect an anomaly that he wants to study in more detail. Therefore, a zoom functionality is implemented with the goal that the client do not need to type the whole date a second time for plotting. He can select with the mouse a time range he wants to zoom in. Since processing a request



is very fast, although a big time range is requested, the zoom functionality is implemented in the same way: When the client wants to zoom in the plot, a new request is sent with the new time and amplitude range.

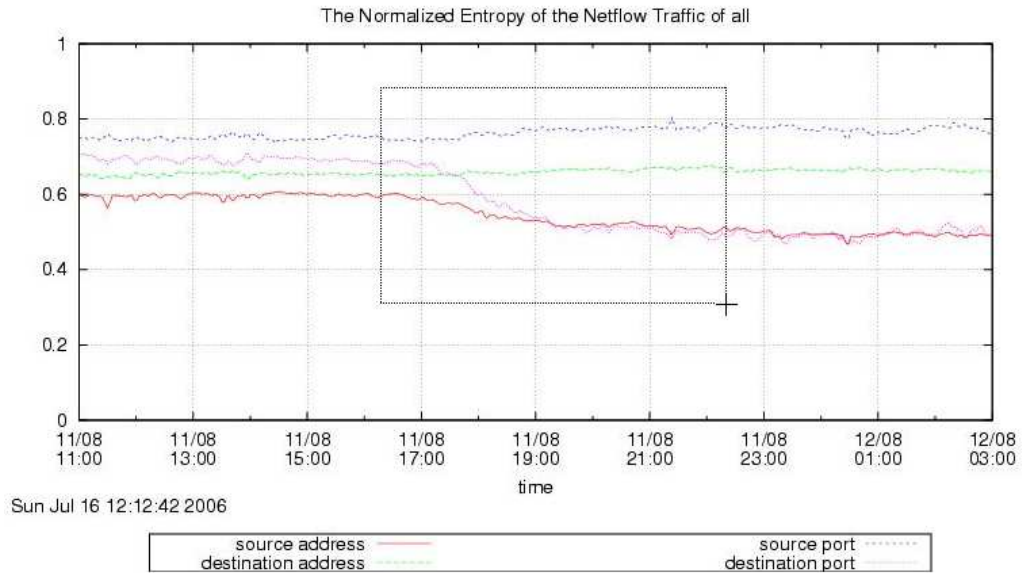


Figure 4.3: Zoom in with the mouse.

The zoom in function is implemented in Javascript that calculates the relative position to the pictures and transforms the coordinates to time points. This can be done very easily because both, the coordinates and the time, have a linear behaviour. That feature makes analysing easier and more user-friendly instead of typing the whole date a second time.

## Chapter 5

# Validation

This chapter will show the results of that thesis and compares the used compression algorithms and the two alternatives block compression and iterative compression. It will also show the difference if the transport protocols are analysed separately or not.

The entropy plots that are used for the validation will show two different types of worm: the Blaster Worm and the Witty Worm. They have different behaviours to infect a host and while the Blaster Worm uses TCP as transport protocol, the Witty Worm uses UDP.

The Blaster Worm has the characteristic that one or few hosts will scan a range of addresses. That means that the source addresses will contain less entropy than normal traffic and the destination addresses should contain a bit more entropy. And because the Blaster Worm looks for a leak on a special port by using different source ports for every flow, the source port will contain more entropy while the destination port will contain less entropy than usually.

The Witty Worm has another characteristic than the Blaster Worm because it was used to attack a weakness in a specific firewall product. It randomly scans with a fixed source port and a variable destination port what will result in a distribution of source ports that contains less entropy and in a distribution of destination ports that contains more entropy. Because of a randomly scan, the destination IP address will contain much more entropy and the source IP address a bit less than usual.

The plug-in was tested with a program that simulates a router and exports the NetFlow packets in a real fashion. Therefore, it uses the locally stored NetFlow data and sends them to a specified IP address. For these tests, the program is executed on the same computer as the plug-in processes the data. There is the possibility to send the packets in shorter time intervals than in real what is used to find the restrictions of the plug-in.

### 5.1 Compressors in Comparison

A feature of the plug-in is that it allows to calculate the entropy with block compression and iterative compression by using different compression algorithms in both cases. The following sections will discuss these alternatives and will show the generated plots and their differences. The Appendix D shows some plots

which are calculated with lzo1x-1 and with bzip2 with a bigger figure size for a detailed analysis.

### 5.1.1 Compression Algorithm

In general, every compression algorithm can be used for calculating the entropy of the NetFlow data. The algorithm bzip2 is the slowest used compressor and it still can calculate the entropy in time what means that no packets in the UPFrame buffer get lost. The simulation of the outbreak of the two worms (Blaster Worm and Witty Worm) has shown that the simulation can process one a half times faster than in real and still no packets get lost. In the case of the Blaster Worm, the simulation can process 1.8 times faster with the same result. But the compression time is very different what depends on the used algorithm. For example, lzo1x-1 compresses a block faster than bzip2 a fragment. Table 5.1 shows the used time for compression of these two algorithms. There are only a reference values but it gives a feeling how long the compression time is. The values are from the simulation of the Blaster Worm with a block size of five minutes and a sliding window size (fragment size) of one minute.

compressor	iterative compression (seconds)	block compression (seconds)
lzo1x-1	0.1	1.2
bzip2	2.0	8.0

Table 5.1: Compression speed of lzo1x-1 and bzip2.

Under the condition that no NetFlow packets get lost during data processing of the entropy plug-in, it does not matter which algorithm is used. The visualised characteristic of the Internet traffic will be the same with the exception that the entropy values will have different sizes. The following pictures will show that the normalized entropy depends on the used compression algorithm and its compression ratio but it has no influence on the resulted characteristic. The changes in the entropy over the time is always the same for all algorithms! Thus, every compression algorithm can be used to recongize the NetFlow characteristic for calculating the entropy.

### 5.1.2 Block Compression vs. Iterative Compression

Besides the usage of different compression algorithm, there are the two alternatives block compression and iterative compression which are explained in chapter 2. There is no question that iterative compression is the faster method to calculate the entropy, but in that thesis the first test is done for determining the entropy with that alternative.

The following two figures show the characteristic of the Blaster Worm - figure 5.1 was made with block compression and figure 5.2 with iterative compression.

The entropy is calculated with lzo1x-1 and a block size of five minutes by a sliding window size of one minute is used in both cases. The Blaster Worm had its outbreak a bit after the time 17:00. Before that time, the entropy values contain the characteristic of the normal Internet traffic and show the changes as consequence of the outbreak. The plots are pretty identical and shows the same

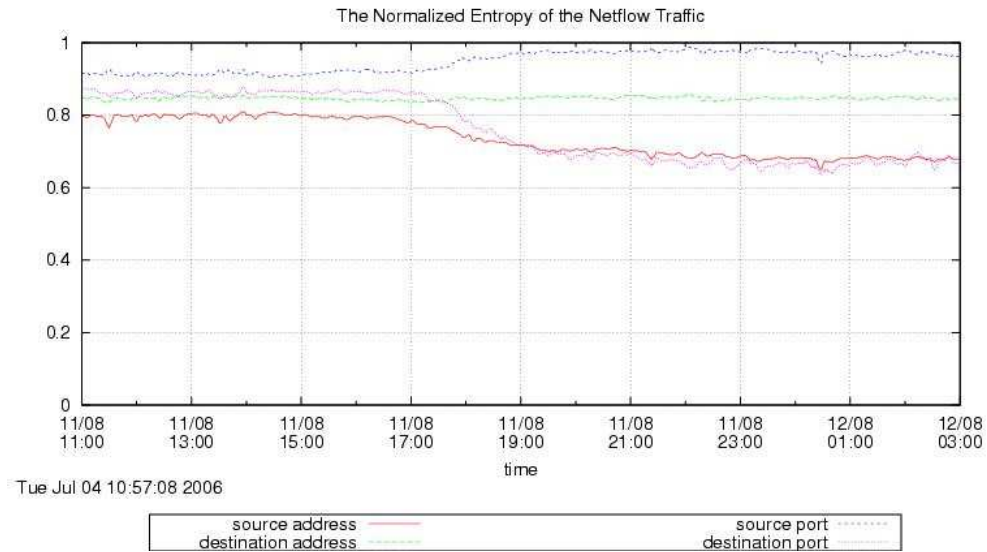


Figure 5.1: Block compressed by lzo1x-1 with a block size of 5min and a sliding window size of 1min. The plots are normalized. That means that the size of each compressed block is divided by its uncompressed block size.

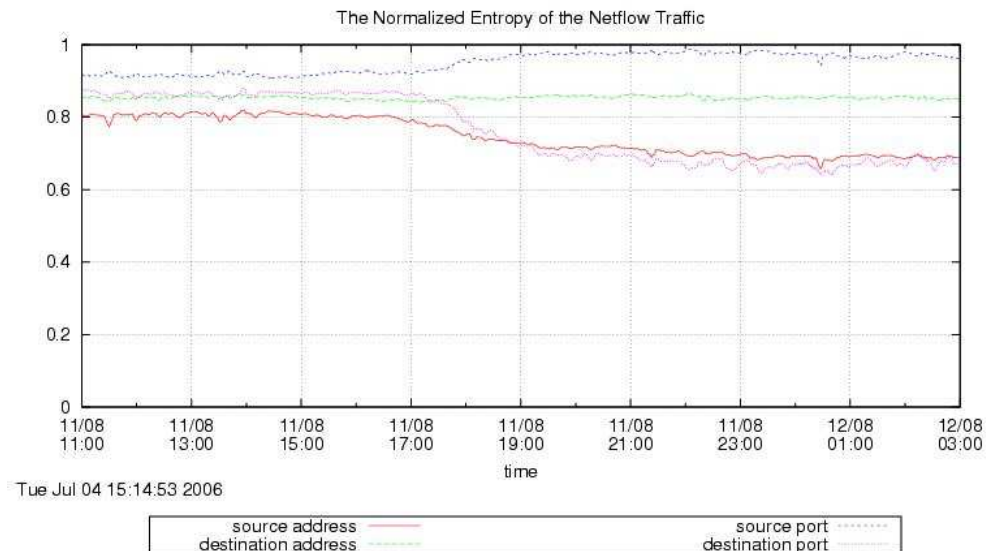


Figure 5.2: Iterative compressed by lzo1x-1 with a block size of 5min and a sliding window size of 1min. The plots are normalized. That means that the size of each compressed fragment is divided by its uncompressed fragment size.

characteristic. It is recognizable that the source IP address and the destination IP address contains a bit more entropy in iterative mode. But the same effect is not really visible for the ports. Therefore the reason is that IP addresses are four bytes values while the ports are two bytes values. As described in chapter 2 the

entropy of . . .	value in %
source IP address	0.98
destination IP address	0.72
source port	0.14
destination port	0.28

Table 5.2: The relative variation of block compression to iterative compression using lzo1x-1.

compression algorithms themselves use a block size internally for compressing the input data. As a consequence a compressor can fill an internal block with more ports than with IP addresses what has the effect that it uses less internal blocks for compressing the ports than for compressing the IP addresses. That means that short-term effects will have more influence over the IP addresses because less addresses are included in one internal block than over ports.

An other reason, why there is a variation between block and iterative compression, is that in general the last internal block of the compressor is not completely filled. Because in iterative mode several fragments are compressed independently every fragment will have a last internal block of the compressor that is not completely filled.

In (5.1) the relative variation in average of block compression to iterative compression is determined where  $e_b(t_k)$  is an entropy value at the time  $t_k$  that is calculated using block compression while  $e_i(t_k)$  is an entropy value at the time  $t_k$  that is calculated using iterative compression.

$$v = \sum_{k=1}^n \frac{|e_b(t_k) - e_i(t_k)|}{n} \quad (5.1)$$

The relative variation is less than one per cent for all four values in average what is shown by table 5.4.

The next two figures show the Blaster Worm too, but the entropy is calculated with the slowest compressor algorithm - with bzip2. The block size and the sliding window size are the same as above.

The first obvious point is that the values contain less entropy than with lzo1x-1 because of its better compression ratio. But the main characteristic is still the same, although bzip2 and lzo1x-1 use very different algorithms to compress the input data. That is visible by the entropy of the source port which does not increase so much as with lzo1x-1 but still shows the same behaviour.

If block compression and iterative compression are compared again, they are more identical than the plots before, because of its bigger internal block and its more efficient compression algorithm. For these plots the internal block size contains 900kB what is the maximum internal block size of bzip2.

The most variation is visible in the range of 19:00 and 01:00 if the source IP address and the destination port are compared. The relative variation for bzip2 is also a bit better what is shown in the following table.

### 5.1.3 Changes in Block Size

The plots above have shown that block compression and iterative compression results in the same characteristic. But what will happen when the block size

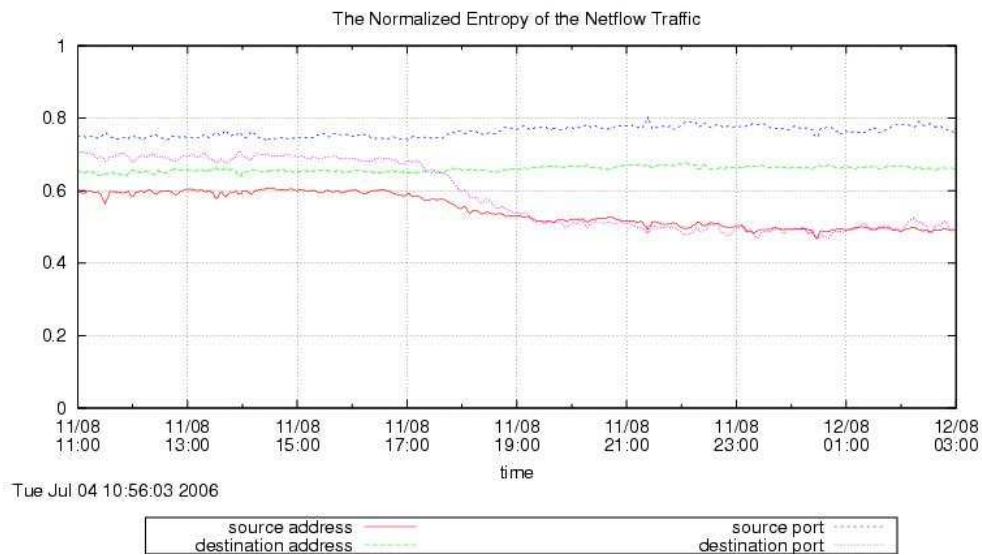


Figure 5.3: Block compressed by bzip2 with a block size of 5min and a sliding window size of 1min. The plots are normalized too. That means that the size of each compressed block is divided by its uncompressed block size.

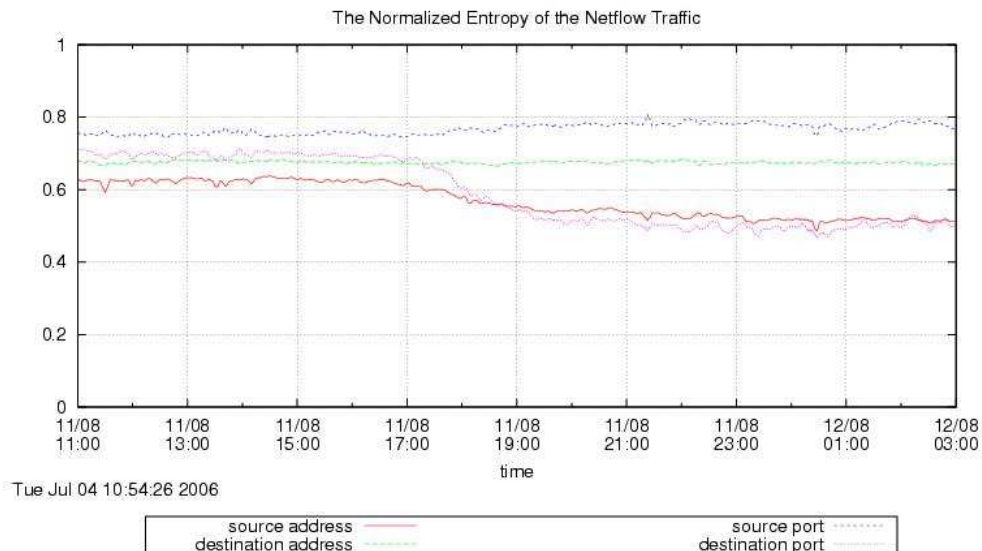


Figure 5.4: Iterative compressed by bzip2 with a block size of 5min and a sliding window size of 1min. The plots are normalized. That means that the size of each compressed fragment is divided by its uncompressed fragment size.

is shorter or longer? Because lzo1x-1 is so extremely fast, a test was done with a block size of ten minutes and a sliding window size of two minutes. There is still no problem for the plug-in to process the data without losing any packets. Figure 5.5 shows the entropy plot with that block size. The main difference is

entropy of . .	value in %
source IP address	0.40
destination IP address	0.24
source port	0.10
destination port	0.11

Table 5.3: The relative variation of block compression to iterative compression using bzip2.

in the peaks which are not so extrem with a longer block size.

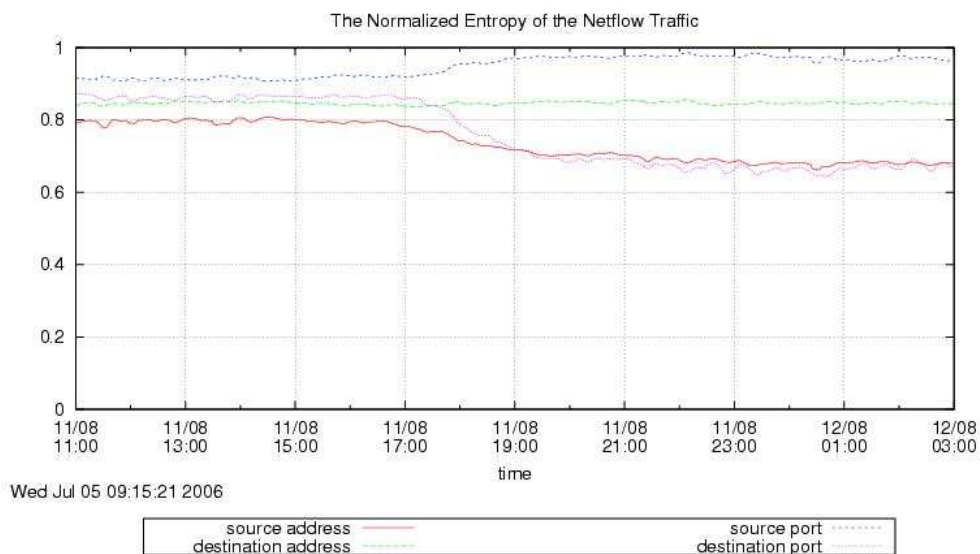


Figure 5.5: Block compressed by lzo1x-1 with a block size of 10min and a sliding window size of 2min. The plots are normalized. That means that the size of each compressed block is divided by its uncompressed block size.

The reason, why the peaks are not so distinctive, is that short-term effects are more reduced and the resulted entropy are smoother over the time. But because the main characteristic is still the same (compare with figure 5.1) there is no need to increase the block size because the plug-in needs more resources for processing and there is no significant advantage with a longer block size. The relative variation is very small as shown in the table 5.4 and is mainly a result of the smoothed values.

Because the sliding window size in figure 5.5 is two minutes and in figure 5.1 is one minute, only every second entropy value of the figure 5.1 is included in the calculation.

Another possibility is to reduce the block size what would be desirable by bzip2 in block mode. In figure 5.6 the used block size is three minutes with a sliding window size of one minute. With such a short block size, the peaks are more pregnant and the plot loses the smoothness because of the influence of short-term effects.

Figure 5.6 includes more short-term effects what is a result of its smaller

entropy of . .	value in %
source IP address	0.27
destination IP address	0.17
source port	0.23
destination port	0.40

Table 5.4: The relative variation of block compression with a block size of 5min and 10min using lzo1x-1.

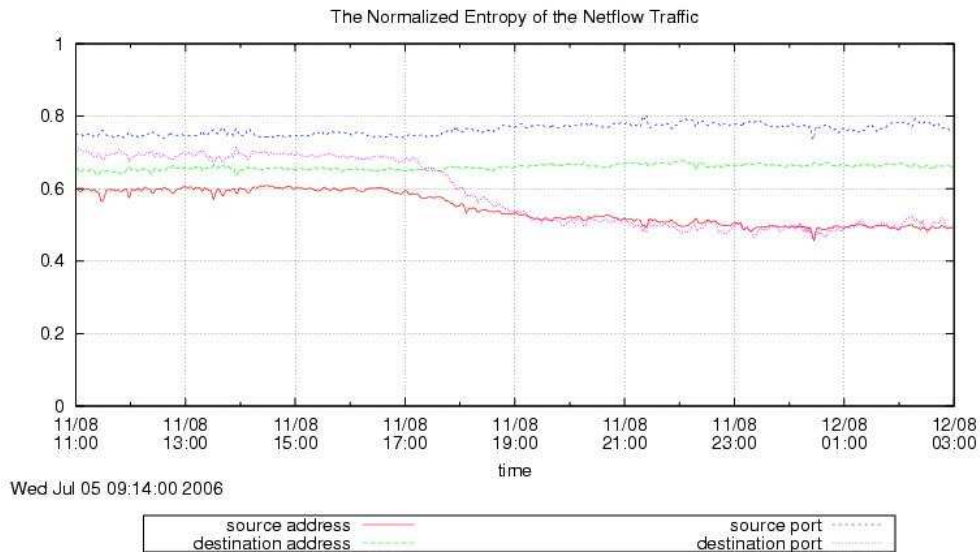


Figure 5.6: Block compressed by bzip2 with a block size of 3min and a sliding window size of 1min. The plots are normalized. That means that the size of each compressed block is divided by its uncompressed block size.

block size. The peaks are more distinctive and shows that the entropy is very variable if small block sizes are used. The characteristic is still present but the peaks become more dominant. A block size of three minutes will be the lower bound but five minutes seems to be a good choice. Again, the relative variation is very small as shown in table 5.5.

## 5.2 Distinction of the Transport Protocols

The Blaster Worm infected about 500,000 hosts during the initial outbreak [3] what is recognizable in the plots above. But there are other worms which look for a specific weakness of a specific product. The Witty Worm is such a worm and, as shown in figure 5.7, it is not so easily detectable like the Blaster Worm since it only infected about 15,000 hosts [3]. Figure 5.7 is made with the fast lzo1x-1 compressor by a block size of five minutes and a sliding window size of one minute.

There is a little change recognizable in the range of four and five o'clock but it is very difficult for analysing. Only the changes in the port distribution indicates



entropy of. . .	value in %
source IP address	0.16
destination IP address	0.12
source port	0.16
destination port	0.22

Table 5.5: The relative variation of block compression with a block size of 5min and 3min using bzip2.

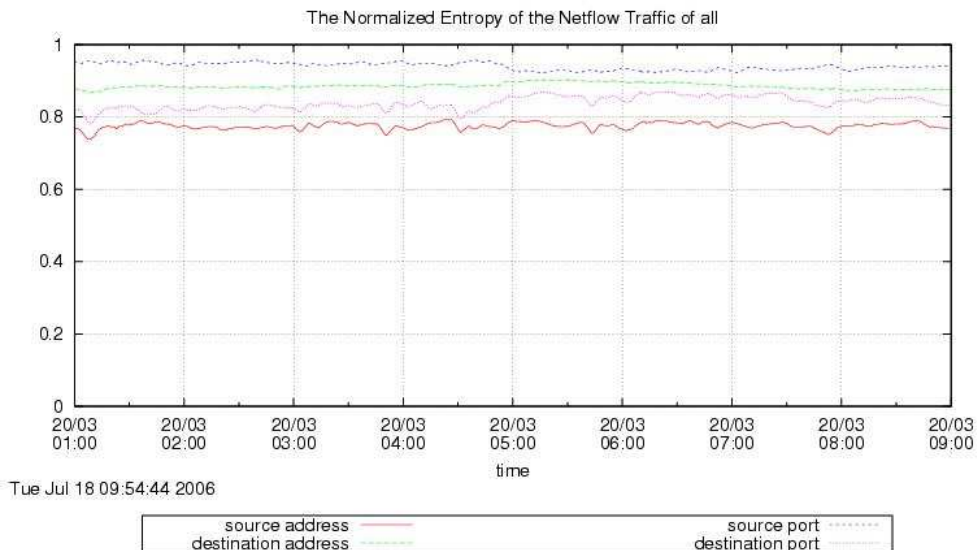


Figure 5.7: Witty with no destination of the used transport protocol.

that something has happen but the characteristic is more or less the same. The outbreak is much more recognizable if only the UDP packets were analysed, what was the transport protocol Witty used. This is shown in figure 5.8.

The entropy is calculated with the same compressor and the same block and sliding window size as in figure 5.7 but because of the distinction of the transport protocols, the UDP traffic shows the characteristic of the Witty Worm and can be easily detected.

### 5.3 Conclusion

Currently, only five per cent of the traffic in the Swiss backbone are forwarded by its border gateway routers and both compression alternatives, block compression and iterative compression, can be used with all included compression algorithms without losing any packets. Both mode can still process the NetFlow data if the NetFlow packets arrive one a half times faster than in real. But there will be the tendency for using iterative compression with a fast compression algorithm like one of the lzolx family. Iterative compression reuses already compressed fragments what makes it more efficient than block compression and reduces the compression time and therewith the whole processing time.

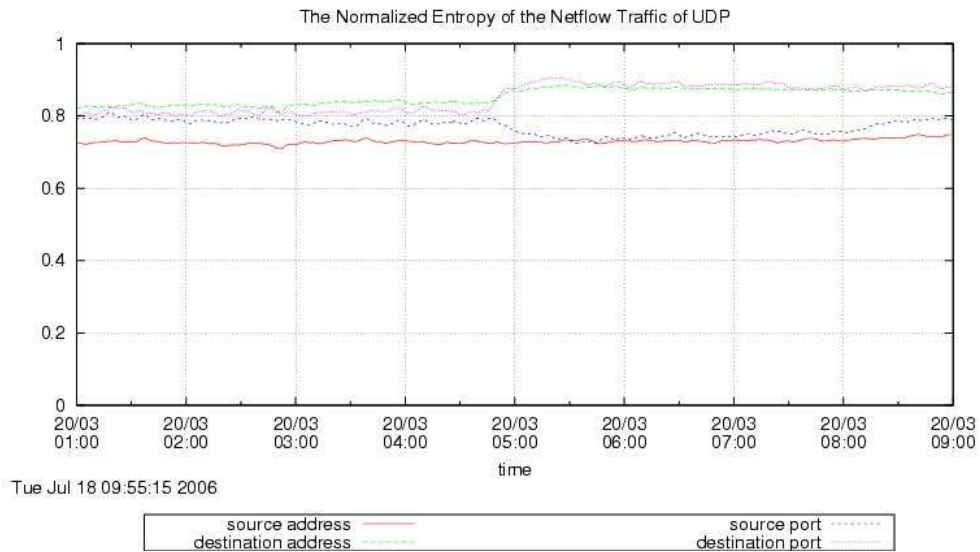


Figure 5.8: The characteristic of the UDP traffic during the Witty outbreak.

The reduced compression time depends on the used block and sliding window size because that are the indicator how often every part is compressed in block mode. Iterative compression has also the advantage that it must collect less data before the entropy can be calculated with the consequence that the time for sorting the hash table is shorter. That is an important point too because the time increases exponentially with the number of elements that must be sorted. Iterative compression needs also less memory but that is not a very important point since in block mode it collects about one million data records that corresponds to about twelve megabytes what the plug-in must store. The tests has shown that only the needed CPU time and the CPU speed can be a problem for the plug-in because of the compression time. But with iterative compression one is on the safe side.

Thus, in general iterative compression is much more resource-friendly and as the plots above have shown the results are almost the same. The most important fact is that the characteristic is always the same because that is what will be analysed.

But there is also shown the restrictions of the entropy plug-in: A worm must infect many hosts and has to generate many flows before it is detectable. The number of needed flows can be reduced if a distinction of the transport protocols is made what is shown with the Witty Worm because the calculated entropy is more sensitive on some changes in the Internet traffic. That does not mean that short-term effects have more influence on the resulted characteristic since the entropy is still determined in the same fashion as there is made no distinction of the transport protocols. There is almost no difference in the processing time if the distinction of the transport protocols is made. The plug-in needs a bit more time for managing and collecting the data because there are four independent hash tables. But since every hash table contains less record entries the time for sorting is shorter what compensates the loss of time during collecting.

## 5.4 Future Work

Now, there exists a tool-set that calculates the entropy in near real-time and plots the resulted characteristic for being analysed by human. The next step will be a concept that analyses the entropy in near real-time without someone must watching the generated plots. Therefore, the characteristic of a worm outbreak must be well understood in order that a tool can be implemented that analyses the stored normalized entropy and starts an alarm if an outbreak is detected. This is also still possible in near real-time only because together with the delay of the export time of the router and the delay of the plug-in there is a delay that must be waited to be sure that the recognized effect is a worm outbreak and not a short-term effect. Therefore the question is if the calculated entropy gives enough information or if other values in the Cisco NetFlow V5 packets must additionally be used to detect it in near real-time.

This thesis developed a tool that is very useful for analysing the NetFlow traffic posteriorly and is a first step to something like an "early-warning system". But it is only a first step because if a worm outbreak is detected, it is still too late and many host will already be infected. But the plug-in is implemented in a very efficient fashion, so that there is the possibility to include much more NetFlow data than currently is considered and can show the Internet traffic behaviour of a big backbone.

The entropy model and the plug-in can be used for every anomaly detection and is not restricted for worm detection only. Thus, they can be used in other fields that analyse the behaviour of the Internet traffic.

## 5.5 Acknowledgement

At that point I want to reciprocate my supervisors, Arno Wagner and Bernhard Tellenbach, for their constructive inputs and for their invested time. Furhter-more, I want to thank Marianne Michel who has designed the layout of the web interface and Theus Hossmann who helped me with the Javascript for the zoom in functionality.

# Appendix A

## Installation of the Plug-in

The following should help to install and execute the plug-in. Consult also the added CD that contains some example files.

### A.1 Requirements

For installing the plug-in on a Linux or UNIX based system the following libraries must be available:

1. libbz2-1.0
2. zlib1g 1.2.1.1-3
3. liblz1 1.08-1
4. libm-2.3.2

The plug-in only works with UPFrame because UPFrame distributes the NetFlow packets and the plug-in needs several header files and some shared libraries (libupframe.so, libnetflowutil.so) of it. On the CD there is an example file, named "Makefile", for compiling the plug-in. For installing UPFrame see [4] where you can download the framework and some other plug-ins and where you can find a documentation about UPFrame.

The used file for the plug-in are available on the CD and are:

**entropy.c** This is the core part of the plug-in that collects and manages the data and stores the calculated entropy in a binary file.

**entropy\_tools.c** In this file all needed compressors are contained for block and iterative compression.

**linked\_collection.c** That file contains contains a class like program that is needed by the plug-in for the collection.

**cfile\_tools.c** This file is not directly used by entropy.c but is needed by the others.

There are also some header files which are available on the CD too. Entropy.h is the header entropy.c uses and there every function is documented.

## Appendix B

# Configure the Plug-in

### B.1 Argument List

The plug-in can set several parameters and is very configurable. In the list below every argument is specified and explained what could be very helpful for the first use.

argument	description
-b < <i>number</i> >	It defines the internal block size of bzip2 that can be a number between 1 and 9 and stands for 100kB and 900kB respectively. The default value is "9".
-c < <i>string</i> >	With the argumet -c the config file is determined. The absolute path must be included.
-e < >	Determines that the plug-in should process the data with the distinction of the transport protocols (TCP, UDP, ICMP, Others). If this argument is not used, then the plug-in makes no distinction.
-f < <i>string</i> >	That parameter defines the name of the log file what also includes the absolute path.
-h < <i>integer</i> >	It is used to set the size of the hash table. The number should be an integer x that satisfies the condition: $x = 2^n - 1, n \in \mathbf{N}$ . In combination with the bit mask it defines the time range that can be collected. The default value is "511".
-i < >	This argument with no parameter says that iterative compression should be used. If the argument is not included, block compression is used.
-l < <i>integer</i> >	That defines the block size that is compressed at once in block compression. It must also be set in iterative compression! The size is defined in number of minutes and should be an integer. The default value is "5".

-m < <i>integer</i> >	It specifies the number of the bits of that the bit mask should be shifted. The default value is "12" ( $2^{12} = 4096$ ms).
-n < <i>integer</i> >	That defines the waitTime parameter. After a block is full and ready for compressing the plug-in continues with collecting until an export time is waitTime minutes later, then it starts with compressing. The default value is "10".
-o < <i>string</i> >	This argument specifies which output channel should be used for the logging entries. The string can be: - stdout: prints the log to the standard output. - file: prints the log to the file specified by -c. - syslog: syslog is used for the log outputs. The default value is "stdout".
-s < <i>integer</i> >	It determines the size of the binary files where the calculated entropy is stored. The number is defined in hours. The default value is "24" (one day).
-t < <i>string</i> >	That specifies which compressor algorithm should be used. Possible is: - lzo: stands for lzo1x-1. - lzo999: stands for lzo1x-999. - gz: stands for gzip. - bz2: stands for bzip2. The default value is "lzo".
-w < <i>integer</i> >	This defines the sliding window size that should can divide the block size. It has also the unit minutes. The default value is "1".

Table B.1: Argument list of the plug-in

## B.2 The Config File

The config file is one of many arguments for the plug-in. It determines the directory where the entropy files should be stored and what the file end name should be.

Do not change the order or the names of the first word at each line!

There is also an example file on the CD named "entropy.cfg".

## B.3 Converter

After compressing the data, the estimated entropies and some other values must be stored in a file for the visualisation tool. An input parameter for the plug-in is how big such a file should be. Therefore, the parameter defines how many hours should be stored in one file.

The created file is a binary file that contains the following information for every entry:

- timestamp in milliseconds (8 bytes)
- entropy of the source IP address (4 bytes)

- entropy of the destination IP address (4 bytes)
- entropy of the source port (4 bytes)
- entropy of the destination port (4 bytes)
- the number of packets of all flows for these estimated entropies (8 bytes)
- the ten most used ports of these flows  
( $10 * 2$  bytes +  $10 * 8$  bytes = 100 bytes)

The reason why to write a binary file is that the plug-in will be used for long-time observation and since it is thought for near real-time analysis, it will be in use the whole time, ideally. That means that it will generate a huge number of data that will need much disk space. If a binary format is used in contrary to a text format, the needed disk space can be reduced by more than 36%. Thus, it seems to be a good choice to use the binary format and take the little time to convert the data if they are used.

The binary files can be converted to a text file very fast using the C program `binary2textForHuman`. It generates a file with a tabluar list of the four entropy values and the statistical data at the end of the file. For the execution use the following syntax:

```
binary2textForHuman < filename > < binaryfile > ...
```

filename: contains the name of the generated text file.

binaryfile: is the binary file that is wished to compress.

There can be added more than one binary file,  
but the results will be added to the same  
text file.

If more than one binary file are converted, the text file still contains one table with the four entropy values what means that the values of a new binary file are added at the end of the entries of the files before. There is still one entry with statistical data since it addes them together.

## Appendix C

# Installation of CGI Scripts

For the CGI scripts, there must be installed a web server. During the thesis XAMPP was used what can be downloaded by [14]. The following files should be placed in the directory "cgi-bin" of the web server:

1. index.pl
2. start.pl
3. picRequest.pl
4. picRequestProtocol.pl
5. lifeMonitor.pl
6. lifeMonitorProtocol.pl
7. zoomin.pl
8. zoominProtocol.pl

Both zoom in scripts need some Javascript files that must be placed in the directory "htdocs" like usual HTML code. The Javascript files are:

1. div.js
2. stifu.js
3. stifuProtocol.js

Furthermore, there are three other files that must be placed but the location is user-defined. The first file is plotPicture.sh what is the Bash script that starts gnuplot and generates the picture. The location must be specified in the entropy\_perl.cfg file. The second one is the C program that converts the binary files to text files formatted for gnuplot and is called binary2textConverter. The last one is the entropy\_perl.cfg which is the config file for all CGI scripts. During the thesis it was placed in the same directory like the Perl script files. If you choose another location you must change the directory in every Perl script file (except index.pl and start.pl). In that config file the following paths are defined:



**shellScriptPath** That defines the location of the Bash script plotPicture.sh.

**picturePath** This location is used for the generated picture and is in "htdocs/images" by the standard configuration of xampp.

**gnuplotDataFilePath** There are the generated text files that contain the entropy stored by binary2textConverter.

**cFilePath** That is the path of the C program binary2textConverter.

**counterFilePath** The files in that location contain the statistical information of the NetFlow data which are also generated by binary2textConverter.

**entropyConfigFile** That defines the location of entropy.cfg what is the config file of the plug-in. With the information of that file, the CGI interface determines the location of the binary files.

## Appendix D

# Entropy Plots

In the following there are some plots, whereby every plot is on one page. Thus, more details can be analysed.

- Figure D.1 shows the Blaster Worm and is calculated with the default values what means a block size of five minutes and a sliding window size of one minute using block compression.
- Figure D.2 also shows the Blaster Worm with the default values but it is compressed with bzip2 using iterative compression.
- Figure D.3 shows the Witty Worm but only the UDP traffic. It is created using the default values and is compressed by lzo1x-1 with iterative compression.

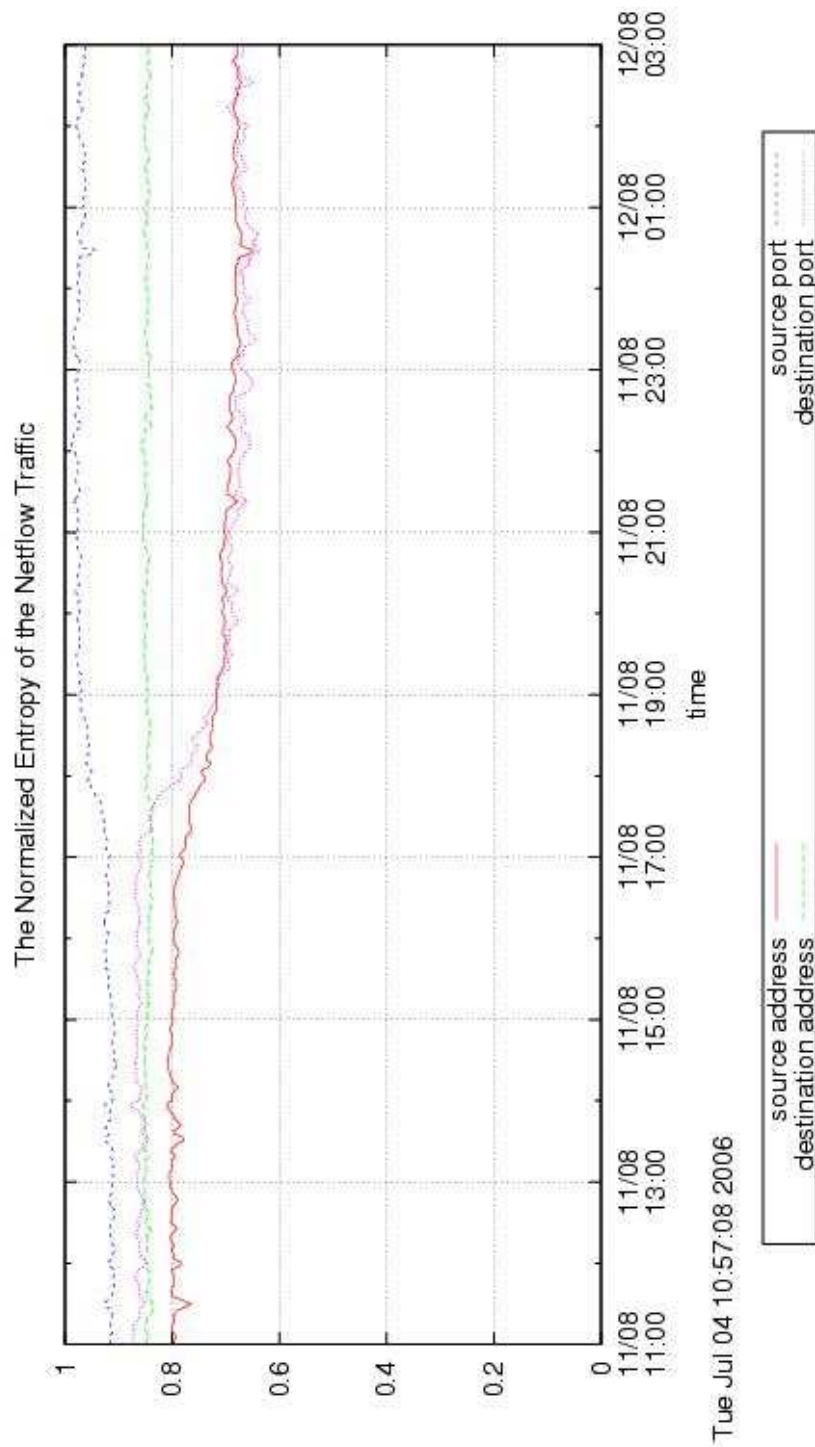


Figure D.1: Block compressed by lzo1x-1 with a block size of 5min and a sliding window size of 1min.

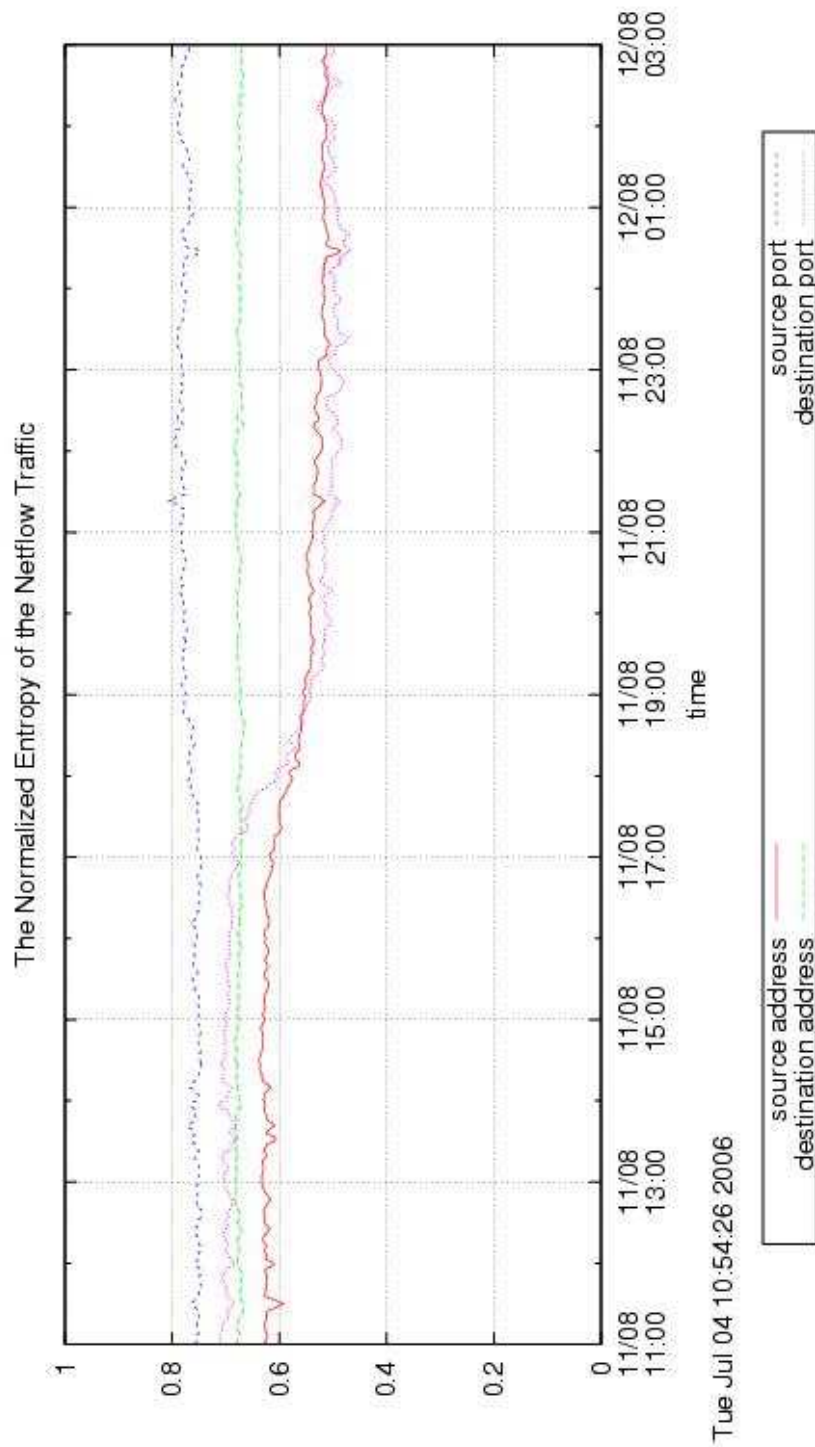


Figure D.2: Iterative compressed by bzip2 with a block size of 5min and a sliding window size of 1min.

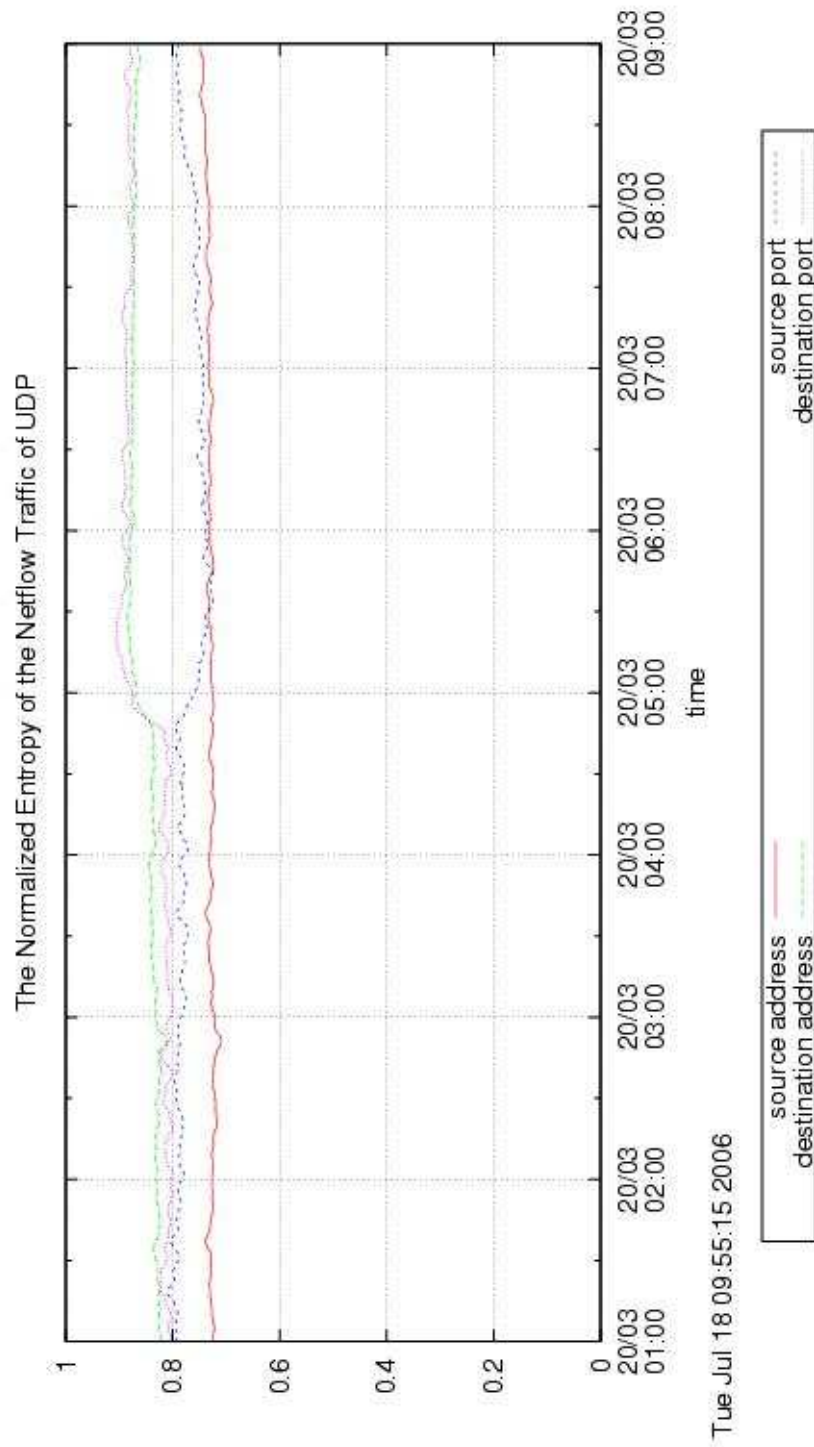


Figure D.3: The characteristic of the UDP traffic during the Witty outbreak calculated with lzo1x-1 using iterative compression.

# Bibliography

- [1] A. Wagner. *Entropy Based Worm and Anomaly Detection in Fast IP Networks*, 2005
- [2] Thomas Dübendorfer, Arno Wagner. DDoSVax.  
<http://www.tik.ee.ethz.ch/~ddosvax/>
- [3] A. Wagner. *Entropy-based Worm Detection in Backbone Networks*, to appear
- [4] Caspar Schlegel. UPFrame - An Extendible Framework for Reception and Processing of UDP Data <http://www.tik.ee.ethz.ch/~ddosvax/upframe/>
- [5] Caspar Schlegel. *Real-time UDP Data Processing Framework*, 2003.  
<http://www.tik.ee.ethz.ch/~ddosvax/sada/>
- [6] Theus Hossmann. *Analysis of Sobig.F and Blaster Worm Characteristics*, 2004. <http://www.tik.ee.ethz.ch/~ddosvax/sada/>
- [7] Christoph Göldi, Roman Hiestand. *Scan Detection Based Identification of Worm-Infected Hosts*, 2005. <http://www.tik.ee.ethz.ch/~ddosvax/sada>
- [8] René Gallati. *Near Real-Time Detection of Traffic Usage Rhythm Anomalies in the Backbone*, 2005.  
<http://www.tik.ee.ethz.ch/~ddosvax/sada>
- [9] Julian Seward. bzip2 - Documentation.  
<http://www.bzip.org/1.0.3/bzip2-manual-1.0.3.html>
- [10] Markus Franz Xaver Johannes Oberhumer. lzo - Documentation.  
<http://www.oberhumer.com/opensource/lzo/lzodoc.php>
- [11] Jean-loup Gailly, Mark Adler. The gzip home page. <http://www.gzip.org/>
- [12] Petr Mikulik. gnuplot homepage. <http://www.gnuplot.info/>
- [13] Anakin Tatooine. gnuplot - not so frequently asked questions.  
<http://t16web.lanl.gov/Kawano/gnuplot/intro/working-e.html>
- [14] Kai Seidler. apache friends - xampp.  
<http://www.apachefriends.org/de/xampp.html>