



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich



Institut für  
Technische Informatik und  
Kommunikationsnetze

Jonas Bolliger  
Thomas Kaufmann

# Detecting Bots in Internet Relay Chat Systems

Semester Thesis SA-2004.29  
May 2004 to July 2004

Supervisor: Thomas Duebendorfer  
Co-Supervisor: Arno Wagner  
Professor: Bernhard Plattner

## Abstract

It is well known that Internet Relay Chat (IRC) is used not only by humans for harmless chatting. IRC can also be misused as a simple means to send commands to malicious programs (called bots) running on compromised hosts (so called zombies). A “master” can log into a specific IRC channel, where hundreds or even thousands of bots are listening to, and e.g. enter the IP address of a machine. Shortly thereafter, a distributed denial-of-service (DDoS) attack against the announced host will be launched.

This Semester’s Thesis analyzes two different bot types. First the “*longtime standing connection bot*”, a bot that is logged in for a long period of time, and secondly the “*fast joining bot*”, a bot that is involved in a fast increase of the number of logged in IRC clients.

Finally we implement a small toolbox that is able to give information about an IRC connection (if the IRC server is known). Further the tool can detect the fast joining trait and make some further investigations about their origin (i.e. subnet analysis).

This detection is based on the analysis of Cisco NetFlow data, in other words, recorded network traffic with highly reduced information content.

# Contents

<b>1</b>	<b>Introduction and Problem Description</b>	<b>6</b>
1.1	IRC Network	6
1.2	Cisco NetFlow Data	6
1.3	Bots	6
1.4	DDoS in General	7
1.5	Internet Relay Chat and DDoS	7
1.6	The DDoSVax Project	7
1.7	The Task	7
1.7.1	Information Gathering	8
1.7.2	IRC Traffic Measurement	8
1.7.3	Algorithm Development	8
1.7.4	Validation	8
1.8	Deliverables	8
<b>2</b>	<b>Examined Bot Types</b>	<b>9</b>
2.1	Introduction	9
2.2	Bots with Longtime Standing Connections	9
2.2.1	Ping-Pong Signature	10
2.3	Fast Joining Bots	13
2.3.1	Number of Logged in IRC Clients	13
2.3.2	Detect Increase in the Number of Logged in IRC Clients	13
2.3.3	Subnet Analysis	14
<b>3</b>	<b>Implementation</b>	<b>15</b>
3.1	Offline Analysis	15
3.1.1	Introduction	15
3.1.2	Sliding Window	15
3.1.3	“Longtime Standing Connection Bots” Implementation	17
3.1.4	“Fast Joining Bots” Implementation	18
3.2	Online Analysis	22
3.2.1	Introduction	22
3.2.2	“Longtime Standing Connection Bots” Implementation	22
3.2.3	“Fast Joining Bots” Implementation	23
<b>4</b>	<b>Results</b>	<b>28</b>
4.1	Description of Measurement	28
4.2	“Longtime Standing Connection” Analysis	28
4.3	“Fast Joining Bots” Analysis	29
<b>5</b>	<b>Summary</b>	<b>32</b>
5.1	Conclusion	32
5.1.1	Longtime Standing Connection Bots	32
5.1.2	Fast Joining Bots	32
5.2	Outlook	33
5.2.1	Longtime Standing Connection Bots	33
5.2.2	Fast Joining Bot Detection	33
5.3	Acknowledgment	33

---

<b>A</b>	<b>Deployment</b>	<b>35</b>
A.1	Introduction . . . . .	35
A.2	Framework in a Nutshell . . . . .	35
A.2.1	longtime.pl . . . . .	35
A.2.2	fastjoin.pl . . . . .	36
A.2.3	NetflowTools.pm . . . . .	36
A.2.4	longtime_plot.pl . . . . .	38
<b>B</b>	<b>Pseudo Code “Offline Longtime Standing Connection” Algorithm</b>	<b>39</b>
<b>C</b>	<b>Pseudo Code “Offline Fast Joining Bot Detection” Algorithm</b>	<b>41</b>
<b>D</b>	<b>Output of Fast Joining Bot Analysis</b>	<b>44</b>
<b>E</b>	<b>Pseudo Code “Online Longtime Standing Connection” Algorithm</b>	<b>45</b>
<b>F</b>	<b>Starting Bots on a Huge Number of Hosts Using PSSH</b>	<b>47</b>

# List of Figures

2.1	Ping-Pong . . . . .	10
2.2	Point scale . . . . .	11
2.3	Definition of frame when breach occurs . . . . .	13
2.4	Prefix matching of two addresses . . . . .	14
3.1	Sliding window . . . . .	15
3.2	Flow chart for sliding window . . . . .	16
3.3	Implementation of the offline longtime standing connection bots algorithm . . . . .	17
3.4	Flowchart of the offline longtime standing connection bots algorithm . . . . .	19
3.5	A plot of the number of logged in IRC clients on the 9th of May 2004 . . . . .	21
3.6	Online implementation of the longtime standing connection bots algorithm . . . . .	22
3.7	Flow chart of the online longtime standing connection bots algorithm . . . . .	24
3.8	Structure of the online fast joining bots analysis . . . . .	25
3.9	Flow chart of the data flow in the online analysis of the fast joining bot . . . . .	26
3.10	Detail view of the analysis part in the online analysis of the fast joining bot . . . . .	27
3.11	Realization of the data collection using a ring buffer . . . . .	27
4.1	Setup of our testing environment . . . . .	28
4.2	Analysis of IRC connections . . . . .	29
4.3	Zoom of peak in Figure 4.2 . . . . .	30
4.4	Result of measurement . . . . .	30

# List of Tables

2.1	Result of IRC channel measurements . . . . .	12
2.2	Addendum to Table 2.1 . . . . .	12

# Chapter 1

## Introduction and Problem Description

This chapter will give a brief overview about the environment our thesis took place in and introduces the data our work was based on. It will then present the given task and the way we established our results.

### 1.1 IRC Network

IRC (Internet Relay Chat) is a protocol defined 1982 in Oulu, northern Finland. With this protocol it is possible to easily set up servers to produce complex networks spread all over the world to provide chat services for a big number of users. Within the IRC network it is possible to log in without a registration as it is usual in ICQ, IM, MSN or similar networks. Users can open new separate channels to talk within and they are even able to restrict access to these channels, as every user can be an operator of a channel.

### 1.2 Cisco NetFlow Data

The border routers of the SWITCH network produce Cisco NetFlow Records. This data structure was built to be able to log a huge amount of traffic with a useful CPU usage and disk space, it only contains information about source and destination of a connection and some information about duration and the amount of transported data. It contains no information about the *payload* of a connection. These NetFlow Records were the basis of our investigation. This means that we were not able to gain insight into packet payload, but that we had to gather information out of the context in a broader view. The data monitored in a NetFlow Record are port and IP address for source and destination, the protocol type, the starting and ending time of a connection (and therefore also the duration), the amount of transported data and the number of packets. If a connection lasts longer than 15 minutes, the record will be closed and a new begins; a connection is assumed to be finished after being idle longer than 4 seconds.

### 1.3 Bots

Because the IRC Network is very open and easy usable, it is often used by computer programs, too. Programs installed on different hosts can log onto an IRC server, open new channels and communicate through these channels. These programs are called bots (an abbreviation of Robots). There exists a big numbers of bots that serve for every possible purpose, eg. weather news, FTP messaging, file sharing synchronization, etc. We call these bots "harmless bots". Unfortunately, since the protocol is open to every kind of user, there exist also persons who misuse this openness for malicious purposes. So the possibility to communicate can also be used to coordinate SPAM delivery, DDoS attacks, etc. These so called "malicious bots" can spread,

be installed and run, unrecognized by a user, for a long time, just waiting to perform a DDoS attack.

## 1.4 DDoS in General

Distributed Denial of Service (DDoS) attacks are a threat to Internet services ever since the widely published attacks on ebay.com and amazon.com in 2000. ETH itself was the target of such an attack six months before these commercial sites were hit. ETH suffered repeated complete loss of Internet connectivity ranging from minutes to hours in duration. Massively distributed DDoS attacks have the potential to cause major disruption of Internet functionality up to severely decreasing backbone availability.

## 1.5 Internet Relay Chat and DDoS

It is well known that Internet Relay Chat (IRC) is used not only by humans for chatting but can also serve as a means to send commands to malicious programs (the “bots”) running on compromised hosts (the “zombies”). A person (the “master”) can log into a specific IRC channel, which hundreds or even thousands of bots are listening to, and issue a command such as e.g. *attack <IP address>* that is received and executed by the bots. In this way, the IRC service can be abused to coordinate and launch DDoS attacks.

## 1.6 The DDoSVax Project

In the joint ETH/SWITCH research project DDoSVax<sup>1</sup> abstract Internet traffic data (Cisco NetFlow) is collected at all border gateway routers operated by SWITCH. This data contains information about which Internet hosts were connected to which others and how much data was exchanged over which protocols. For this thesis the DDoSVax research team has established a contact to an administrator of a frequently used IRC system that is temporarily located in the SWITCH network.

## 1.7 The Task

Based on tests with real IRC bots, literature research, the results of a previous thesis and traffic measurements on a real IRC server and on routers in the Internet backbone, algorithms that detect bots abusing an IRC system will be developed and validated.

The three following approaches to detect bots and botnets will be considered. Further approaches are optional to this thesis.

- Bots that are installed by a worm will join shortly after each other into the same IRC network. Such *fast joining bots* should be detected in the DDoSVax NetFlow data.
- Bots normally stay in an IRC system for a long time. Long standing connections to an IRC server should be detected in the DDoSVax NetFlow data.
- Bots are usually not talkative in an IRC channel. Therefore, IRC connections that consists mostly of IRC Ping-Pong traffic and no real conversation should be detected.

The thesis is divided into four main parts, namely information gathering, IRC traffic measurements, algorithm development and validation.

---

<sup>1</sup>See <http://www.tik.ee.ethz.ch/~ddosvax/>



### 1.7.1 Information Gathering

Studying the thesis “Analysis of Internet Relay Chat Usage by DDoS Zombies” written by Stéphane Racine is the first step to familiarize with IRC and its possible misuse by bots.

Further literature research on bots and botnets using IRC, studying real bot code, and setting up an own IRC server will give further insights.

### 1.7.2 IRC Traffic Measurement

By using the worldwide distributed computers of Planet-lab, a setup with our own bots connecting to a (test) IRC server will be installed and its traffic measured on the server (tcpdump) and in the backbone (DDoSVax NetFlow data). The attack part of these bots (e.g. for denial of service attacks) will be disabled to prevent any misuse. The focus of interest lies on the use of IRC to communicate between a master and the bots.

Other IRC traffic measurements, which provide bot-like traffic patterns and that can be used to design and validate bot detection algorithms, will be conceived and executed.

Time consuming analysis, especially that of large amounts of DDoSVax NetFlow data, will be done on the TIK experimental cluster “Scylla”<sup>2</sup>.

### 1.7.3 Algorithm Development

With the information gathered and the measurements done, various algorithms to detect characteristic IRC bot traffic (based on the three approaches) will be developed.

Offline algorithms will be run on the cluster “Scylla” or on a workstation. Online algorithms will run as plugins in the UPFrame UDP processing framework provided by the DDoSVax project.

### 1.7.4 Validation

The last step will be to thoroughly test the online and offline detection algorithms and to adjust important parameters to reduce false positives.

## 1.8 Deliverables

1. *IRC bot traffic signatures* The bot signatures obtained from literature research, own measurements and tests that describe which characteristics specific bot traffic has.
2. *Offline IRC bot detection algorithms* Design decisions and implementation of the various offline IRC bot detection algorithms.
3. *Online IRC bot detection algorithms* Design decisions and UPFrame plugin implementation of the various online IRC bot detection algorithms.
4. *Thesis Documentation* A concise description of the work conducted in this thesis (task, related work, environment, measurements, results and outlook).

---

<sup>2</sup>see <http://www.tik.ee.ethz.ch/~ddosvax/cluster/>

## Chapter 2

# Examined Bot Types

This chapter describes which bot types we examined and why the behavior of these types of IRC clients seem to be especially typical for bots.

### 2.1 Introduction

As it is generally difficult to determine whether a flow originates from a human IRC user or whether a malicious bot caused an IRC connection, we defined two scenarios how a bot will behave within an IRC network. The simplest possibility is that a bot is running permanently on a host without the knowledge of the user. Then the bot will log into the server, as soon as the user connects to the Internet; the pattern then can be observed on the server as a connection that will last for a long time (if the bot is installed on an office computer, the connection will most likely last for eight or more hours). This leads to the first examined bot type - *bots with longtime standing connections*. The following section will further describe how we proceeded.

Another scenario how a bot would behave, is a worm traveling through a local area network and infecting hosts that are not appropriately protected. Since infections like this can happen within a short period of time, we can assume that a bigger number of hosts will log into an IRC server within a short period of time. This would result in an abrupt change in the number of logged-in clients at an IRC server. All these bots can be considered to be bots with long standing connections afterward.

### 2.2 Bots with Longtime Standing Connections

Imagine an Internet user with a host, infected with a malicious bot, so called “Zombie”, going online in the morning and stands online until after the evening. The bot will log into the IRC channel and wait there for an order. This is a common scenario of IRC usage. Therefore it is justifiable to analyze *“bots with longtime standing connections”*.

To analyze this bot type, the following properties were taken (each per connection):

- duration of the connection
- amount of data sent from the IRC client to the IRC server (only IRC traffic!)
- amount of data sent from the IRC server to the IRC client (only IRC traffic!)
- number of IRC Ping-Pong's

To categorize these properties, a rating with a point scale was developed. The point scale has to be flexible, because of further trimming. Therefore we take a parable to distribute points. To evaluate how to set the parameters of the parables, we made a measurement of real IRC channels. We logged into several channels on June 24th 2004 13:00-14:00 MET (Swiss local time) and made a full capturing of the IRC Port 6661. Then we exported the traffic as Cisco

NetFlow to compare the values of the connections. The results found are listed in Table 2.1, the corresponding addendum in Table 2.2. The values are the ones sent, respectively received by our IRC client. Because we only logged into the channel without chatting, the values in the column “Sent” aren’t representative, however the values in the column “Received” are very useful. Considering the Table 2.2 we can get a feeling what is human, or even more, we can see what is possible to read for humans. For example, how big the amount of data can be. Our measurement only shows how a channel is represented in the flows. Taking these values, we can try to set the parable parameters. We derive the following:

- The *duration of the connection* is simple to rate. The longer the duration of a connection is, the more suspicious it is. We remind that here we are searching for longtime standing connections! We use a linear scale. For one day online we give 100 points. See Figure 2.2(a).
- For the *amount of data sent to the IRC sever* we can consider Table 2.1. For example the #mp3\_collective is a channel, which is used for mp3 file sharing purposes. That means, that there are only few human chatters in it. Because of file sharing bots the amount of data is very high, too high to be human. If the amount of data gets bigger, then we can be quite sure that it is not a human chatter. Further, if the amount of data is very high, it is not so important, what the other properties are. Therefore, we took a parable. To find the parameters of it we took these values to set the parameters of our point scale shown in Figure 2.2(b).
- To examine the *amount of data sent to the IRC client*, we adopt a new value: The *quotient*. It is defined as  $\frac{\text{amount of data sent to the IRC Client}}{\text{amount of data sent to the IRC Server}}$ . Imagine an IRC client logged into an IRC channel, doing nothing, would have a very big quotient if the others in the channel would say something. This would be a non-talkative longtime standing connection. On the other hand, if the others are also quiet, the quotient is not big. Another example is, three or four clients chatting in a channel. The quotient would not be big. Therefore, it is very difficult to set this value. We tried it in Figure 2.2(c).
- The *Ping-Pong’s* are again simple to rate. The more Ping-Pong’s we count, the more suspicious the connection is, the more likely it’s a non-talkative longtime standing connection. Therefore we give 100 points for one whole day Ping-Pong’s. See Figure 2.2(d).

### 2.2.1 Ping-Pong Signature

In the section before, we spoke about Ping-Pong’s. But what are Ping-Pong’s and why search for them?

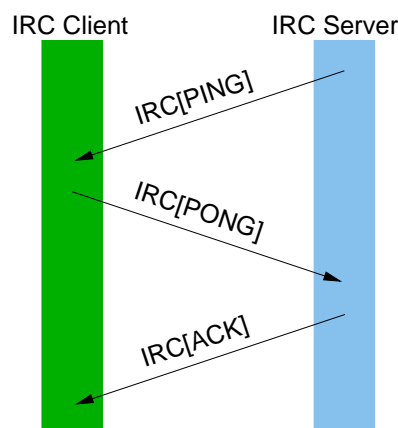


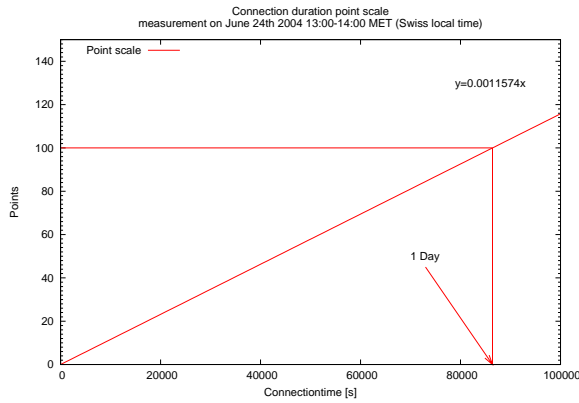
Figure 2.1: Ping-Pong

If an IRC client is idle for a longer time, then the IRC server has to know, if the IRC client is still online and for this he sends regularly<sup>1</sup> an IRC-Ping to the IRC client. The IRC client has to

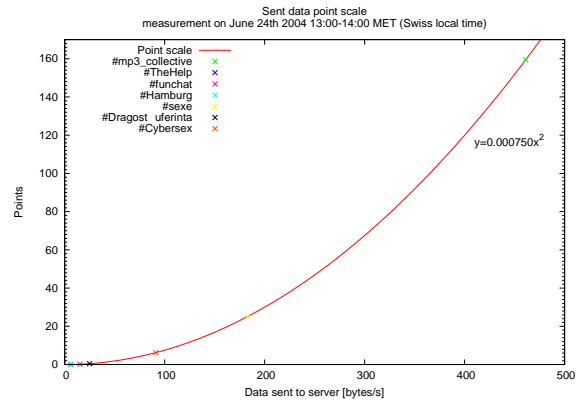
<sup>1</sup>the Undernet IRC server sends a Ping every 180s, if Client is idle

answer this IRC-Ping with an IRC-Pong. The IRC server then sends an Ack to the IRC client. See Figure 2.1.

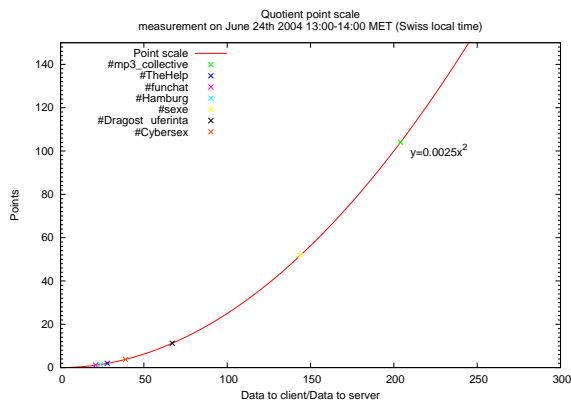
Now, it should be clear, why we want to search for Ping-Pong's: Because Ping-Pong's are a relatively sure indication for bots with "non-talkative longtime standing connections".



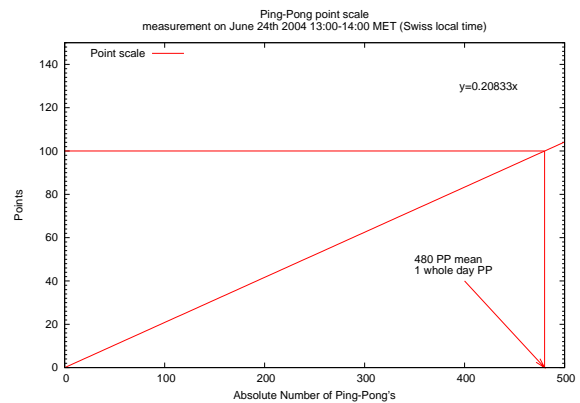
(a) Duration



(b) Amount of Data sent to Server



(c) Quotient



(d) Ping-Pong's

Figure 2.2: Point scale

Channel	Sent [ $\frac{bytes}{s}$ ] to IRC server	Received [ $\frac{bytes}{s}$ ] from IRC server	Packets sent to IRC server	Packets received from IRC server	Con.-time [ <i>min</i> ]	$\frac{Received\ data}{Sent\ data}$
#mp3_collective	2.25	461	209	11120	115	204
#TheHelp	0.215	6	8	56	45	28
#funchat	0.709	15.3	23	108	38	21
#Hamburg	0.242	6	8	31	40	24
#sexe	1.24	183	29	1024	25	144
#Dragost&Suferinta	0.356	25	10	155	39	67
#cybersex	2.32	91	59	704	29	39

Table 2.1: Result of IRC channel measurements

Channel	Explanation
#mp3_collective	a very frequented channel, with file sharing bots in it. Amount of data too high, to be human readable.
#TheHelp	a normal channel, but very low amount of data.
#funchat	a normal channel, but low amount of data.
#Hamburg	a channel with no traffic.
#sexe	a normal channel, some users are chatting. Agreeable to read everything and not boring.
#Dragost&Suferinta	a normal channel, but low amount of data.
#cybersex	a normal channel, some users are chatting. Agreeable to read everything and not boring.

Table 2.2: Addendum to Table 2.1

## 2.3 Fast Joining Bots

In Chapter 2.1, the analysis of fast joining bots was justified. In order to detect these bots, some more detailed examinations are necessary:

- how to determine the number of logged in IRC clients on an IRC server
- how to detect fast increases in this number
- how to detect whether such bots log in from the same subnet

We developed methods to solve the above mentioned problems and successfully implemented them in the analysis tools.

### 2.3.1 Number of Logged in IRC Clients

The Undernet servers are programmed to expect a signal (or a packet) from each client at least all three minutes. If a client is inactive during this time interval, the server will send an IRC-Ping and receive an IRC-Pong, if the IRC client is still running. For us this means that from every active client, we will register a NetFlow flow record at least all three minutes. To be sure to consider lost records, we defined a time slot with a length of seven minutes. If a host produces a NetFlow flow record with the IRC server as destination address within this time, we assume that he is logged in. That reduces the problem significantly; what our program has to do, is to examine, whether a client was already active in the actual time slot, and if not, to increase the temporary number of logged in clients. After the time slot is finished, we know the number of logged in clients during that time.

### 2.3.2 Detect Increase in the Number of Logged in IRC Clients

Since the method above gives the number of logged in clients at an IRC server, we need to detect, whether a significant change of this number occurs. We basically defined a frame, where the number of clients is allowed to move within. The frame consists of a height ( $\Delta hosts$ ) and a width (*timewindow*). Figure 2.3 shows the case, when the number of logged in IRC clients changes suspiciously rapid (marked with "BREACH").

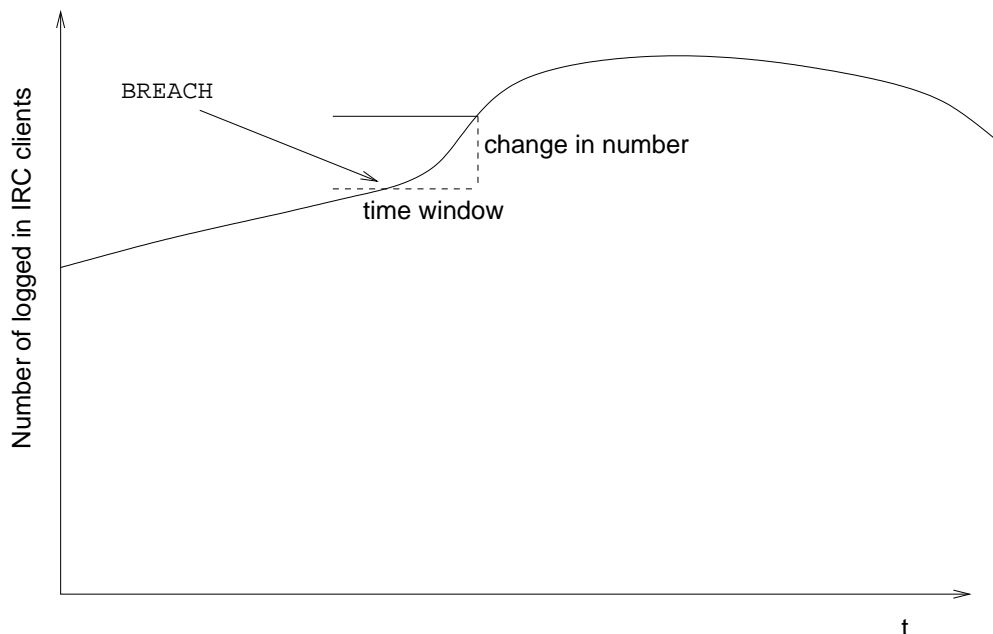


Figure 2.3: Definition of frame when breach occurs

The frame moves as time goes forward, and reports all hosts that were part of the breach. The reported addresses are the ones that were reported as new in the time window.

### 2.3.3 Subnet Analysis

A problem is the fact, that the reported addresses do not obviously have to be bots. We measured a fluctuation of about 20% of the logged in clients every seven minutes. Assuming a total of approx. 6000 logged in clients, we will detect 1200 false clients apart from the 200-300 bots. To sort out the bots, we had to develop an additional search pattern. As already mentioned above, we assume the bots to usually origin from within the same subnets. Therefore we will compare the addresses of the whole list of reported clients. A prefix matching of the (converted) binary addresses gives the number of the matching bits (see Figure 2.4. These results are accumulated for every address. In a case, when a subnet of some dozens of hosts accrues a breach, these hosts will give a much higher number than casual clients.

```
129.132.57.75  => 10000001100001000011100101001011
129.132.57.115 => 10000001100001000011100101110011
                  <----->
                  26 Bits match
```

Figure 2.4: Prefix matching of two addresses

# Chapter 3

## Implementation

In this chapter the online and offline implementations of the two detection algorithms will be described in detail. The offline tool is a Perl script, the online tool is written in C.

### 3.1 Offline Analysis

#### 3.1.1 Introduction

The offline tool takes the NetFlow one hours files from the disk archive. We processed two such files for each hour. The first file contained data of a single router, the second file contained data of three different routers. We developed a framework, to handle this various number of routers in the two files. Fortunately there was a tool<sup>1</sup>, provided by the DDoSVax group, that was able to display the NetFlow data in a human readable manner. Further we could use the Perl scrips from Stéphane Racine, who already wrote some functions, which handles NetFlow files. We made a collection of useful functions, that they are easy to use in different scripts. This library called "NetflowTools" will be explained in Appendix A.2.3.

Because this analysis is offline, we have the advantage to "look" into the future. Therefore we can compose the whole connection first, and then, when the connection is finished, analyze it.

#### 3.1.2 Sliding Window

If we would analyze some hours and days and compose every flow, until the end of our analysis, it will end in a memory problem. Therefore, we cannot analyze as long as we want without cleaning up the memory. So we had to search a solution to clean up the memory at the right time. We decided to take advantage of the "sliding window" technique. The basic idea is to analyze until a predefined point, and then look, which connections are terminated, so that we can report them and free memory. The idea is shown in Figure 3.1.

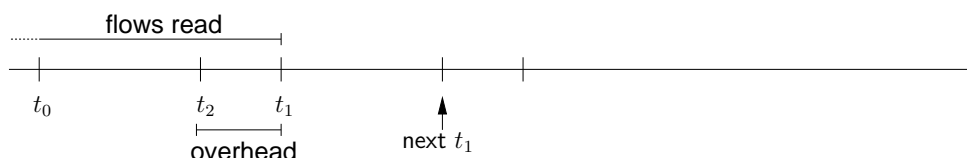


Figure 3.1: Sliding window

As we know from Chapter 1.2, the flows are exported at the least every fifteen minutes after beginning. Furthermore, let us define the overhead as  $t_1 - t_2 = 20 \text{ minutes}$ .

If we compose the flows (see more about composing flows in Chapter 3.1.3) and we arrive at  $t_1$  (the end of the file), we can be sure that every connection that was finished before  $t_2$  is really finished. So we can analyze the connection, report it, and free memory. Surely we can also do the analysis somewhere in the file. Then we only have to adjust  $t_1$  (and also  $t_2$ ) the right way. How the data flow looks like is show in Figure 3.2.

<sup>1</sup>netflow\_to\_text



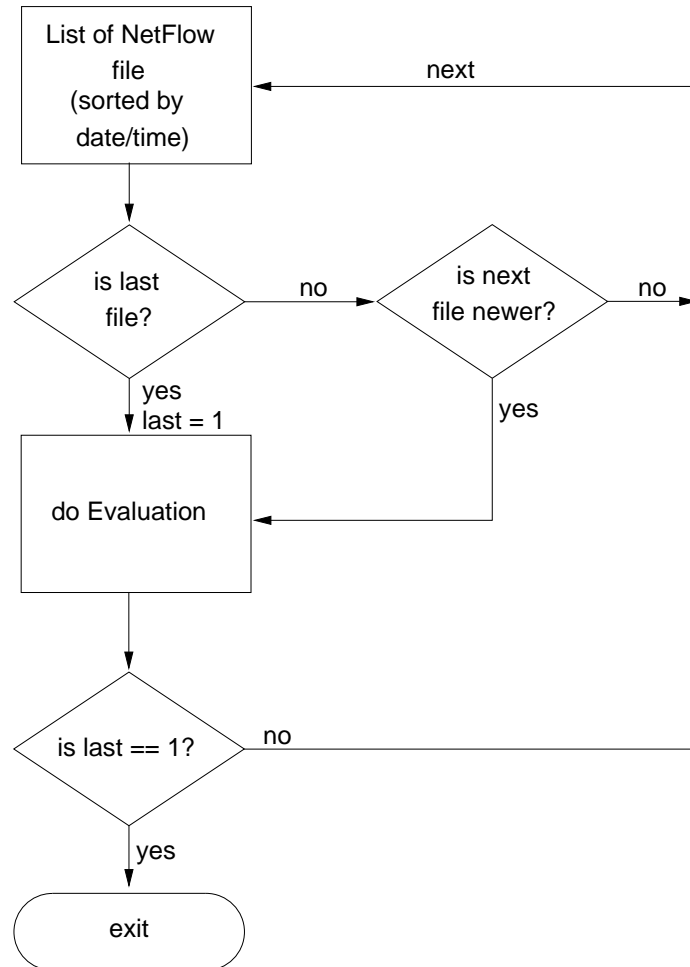


Figure 3.2: Flow chart for sliding window

### 3.1.3 “Longtime Standing Connection Bots” Implementation

The NetFlow data is saved on the disc in the archive from where we read it with the `netflow_to_text` program. The data flow is shown in Figure 3.3.

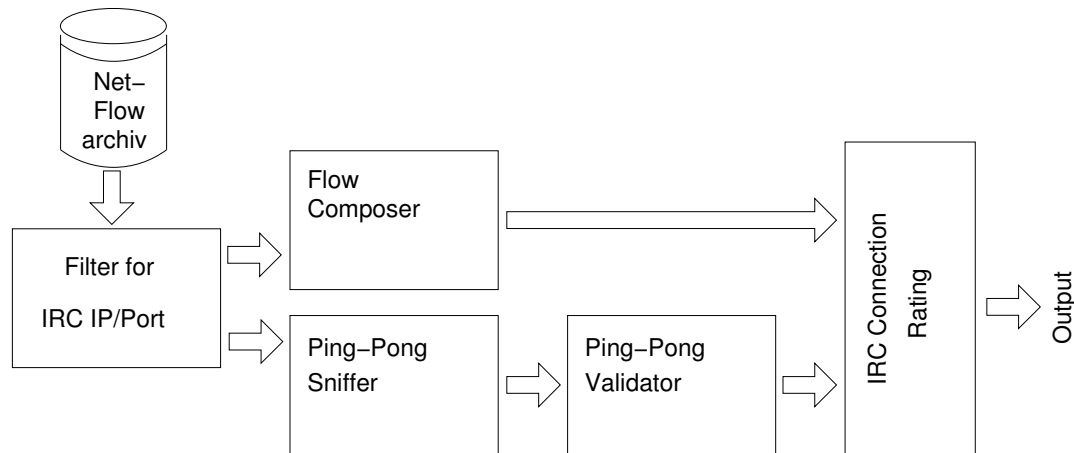


Figure 3.3: Implementation of the offline longtime standing connection bots algorithm

#### Definition of Connection

First let us define what we mean with a connection. A connection is represented by the double **ClientIP:ServerPort**. Everything with the same double, belongs to the same connection.

#### Filter for IRC Connections

Only flows are passed through, that are from the specific IRC server on the specific ports<sup>2</sup>, respectively are going to the specific IRC server and specific ports.

#### Flow Composer

The flow composer takes every flow and checks, if it is a *Server to Client* or a *Client to Server* flow. A new connection is made, if the double is new, and updated if the double already exists. Normally, a flow belonging to an existing connection, is newer and therefore we can update the end time of the existing connection with the end time of the flow. Unfortunately there are some flows delayed and so, from time to time, we have to update the start time of the connection.

#### Ping-Pong Sniffer/Validator

Every flow coming to the Ping-Pong Sniffer is analyzed, if it could be a Ping or a Pong. To find a Ping or a Pong, we took the patterns, which Stéphane Racine found in his Master’s Thesis. Let us define two parameters:

- *l*: length of flow (Bytes)
- *p*: number of packets in flow

A Ping has the pattern:

$$p \geq 2 \text{ and } l \geq 86$$

and a Pong has the pattern:

$$l \geq 46 \text{ and } l \leq 121 \text{ and } p == 1 \text{ or}$$

$$l \geq 86 \text{ and } l \leq 173 \text{ and } p == 2$$

<sup>2</sup>6660-6669, 7000, 7777, 8000

If it could be one of them we try to search the complement of it. If we can find it, we have successfully detected a Ping-Pong. Again, there are some delayed flows, and so we have to search for a Pong, even though the end time of a Ping normally is younger (see Figure 2.1).

The *Ping-Pong Sniffer* also makes sure, that a Ping-Pong only can appear in a defined distance<sup>3</sup>. If there are suddenly two Ping-Pong's in the same time slot, none counts.

### IRC Connection Rating

In this block the rating is made. See Chapter 2.2 for more details.

For a deeper understanding, see the flowchart in Figure 3.4.

### Output

The found connections are written to a file. Each line represent a connection with all the found properties. Furthermore, the points are also reported. The file format looks like<sup>4</sup>:

```

#
# Result of offline plugin 'longtime'
# File written at: 05.07.2004 14:18:34
# Using the following scheme:
# CLIENTIP SERVERPORT BYTES_TO_SERVER BYTES_TO_CLIENT PACKETS_TO_SERVER PACKETS_TO_CLIENT ←
# CONNECTIONS_TO_SERVER PINGPONGS POINTS[] STARTTIME ENDTIME
# POINTS[CON_TIME BYTES_TO_SERVER QUOTIENT PINGPONG]
#
#
2 x.x.x.x 6667 720 0 15 0 2 0 4 1 0 0 ←
   1087924924.3660 1087926620.6720
10 x.x.x.x 6667 288 0 6 0 2 0 2 1 0 0 ←
   1087925170.4440 1087926256.3200
x.x.x.x 6667 144.0 0 3.0 0 1 0 0 1 0 0 ←
   1087926431.2970 1087926440.1290
12 x.x.x.x 6667 288 0 6 0 1 0 1 1 0 0 ←
   1087924567.4390 1087924917.2000
x.x.x.x 6667 288 0 6 0 1 0 0 1 0 0 ←
   1087923577.8140 1087923713.4950
14 x.x.x.x 6667 311961 390418 6013 6943 1 0 0 69112 13 0 ←
   1087927185.5900 1087927192.8230
x.x.x.x 6667 128.0 473 3.0 4 1 0 0 1 37 0 ←
   1087925722.7570 1087925733.9570
16 x.x.x.x 6667 73888 83128 1373 840 1 0 8 1 11 0 ←
   1087923253.3350 1087926683.4570
x.x.x.x 6667 448760 550995 8507 9858 1 0 0 766352 12 0 ←
   1087927189.0480 1087927192.1210
18 x.x.x.x 6667 3523 4277 59 48 1 0 6 1 12 0 ←
   1087924090.2640 1087926894.4630
x.x.x.x 7000 3896 4428 66 51 1 0 6 1 11 0 ←
   1087924087.1260 1087926891.6480

```

A pseudo code of the offline longtime standing connection bots algorithm can be found in Appendix B.

### 3.1.4 "Fast Joining Bots" Implementation

The implementation of the fast joining bot analysis follows the description given in Chapter 2.3. The first thing to implement was the bucket counting for number of connections. After the number is evaluated, we can check for possible breaches in our defined frame. Whenever the number of IRC clients at our server increases more than our variable `boundary_top` within the `boundary_time` timeslots, a breach is reported, accompanied by the suspicious IP addresses. After this file is written, a second program called `postanalysis` is run, that will perform the subnet analysis and print the results in a fancy format. This program is the same as for the online analysis and expects equivalent inputs for both methods (online/offline) and gives therefore the same results.

### Data Collector

This part checks every incoming flow for an appropriate destination address (IRC server) and destination port (IRC ports). Afterwards the start and end time is taken and examined, whether

<sup>3</sup>the Undernet IRC server send a Ping every 180s, if a Client is idle

<sup>4</sup>IP addresses marked with "x.x.x.x" are faked for security reasons

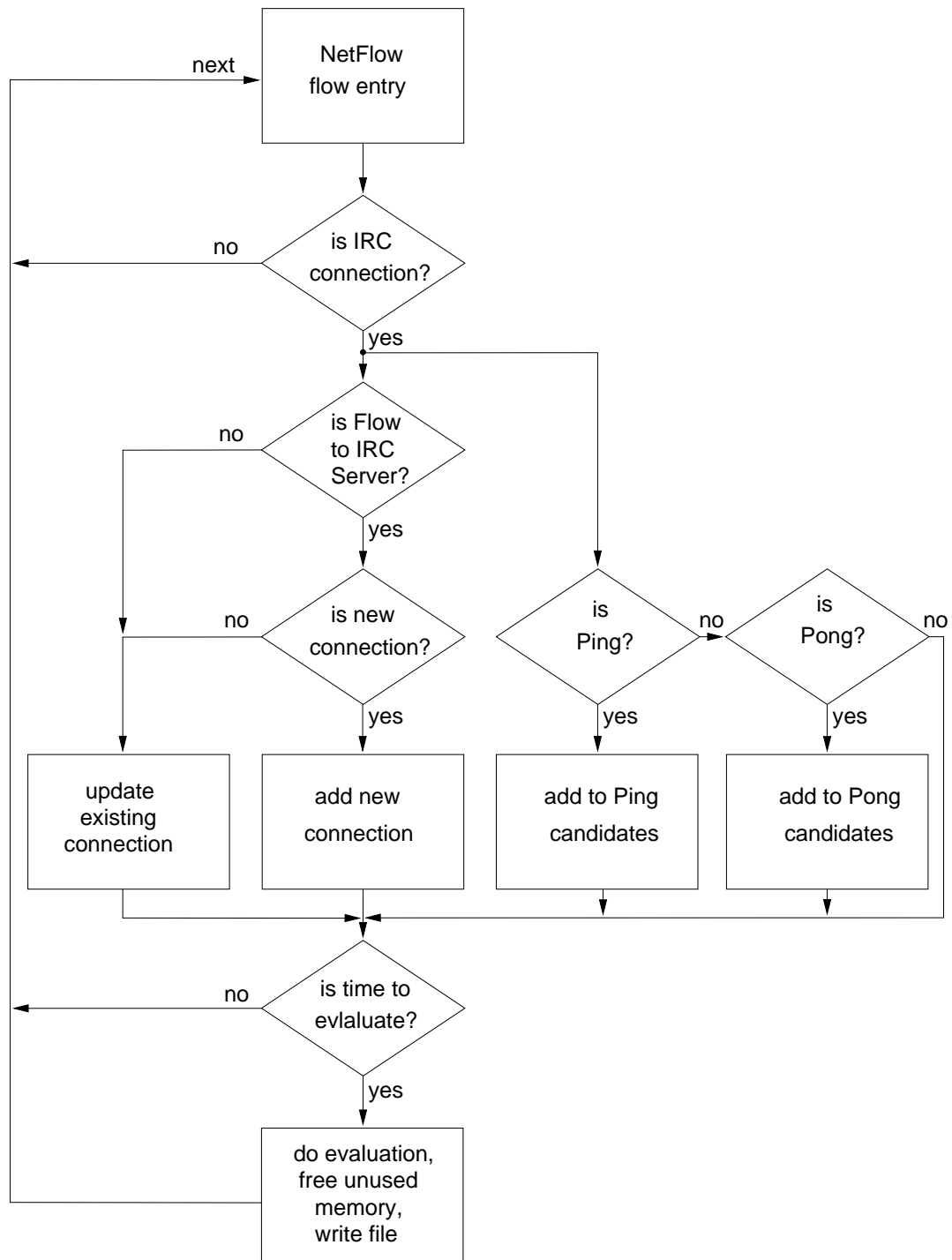


Figure 3.4: Flowchart of the offline longtime standing connection bots algorithm

this NetFlow record fits in the actual time slot. If yes, the address is checked, whether it already performed a connection and the counter increased for every time slot the flow appears, if it is new. If the record is newer, the actual time slot increases and the the same action is performed as described above. If the record comes late, it is entered one time slot in the past. Records delayed more than one time slot are ignored.

After this, a moving average calculation is performed. Therefore, depending on the size of the average window, several bucket values in the future must be known. This is the reason, why this calculation is done on the offline analysis only. The now flattened values are given to the breach detector.

### Breach Detector

Since we know the recent numbers of logged in IRC clients, we step backwards in time until we reach the end of our frame and search for breaches. When we detect a breach, we search for the IP addresses, that are active in this slot and were not in the last one; these addresses are written into the new hash. After we successfully found breaches all found addresses are written to `#basename.pdat` and the post analysis tool is called. Additionally we write the number of logged in IRC clients into `#basename.number` every time a time slot is finished.

Typical outputs of this program can be seen here<sup>5</sup>:

```
# Result of offline tool 'fastjoin'
2 # File written at 07.07.2004 09:19:36, contains data of 14:30:00
# Using the following sheme: ADDRESS      NUMBEROFCONNECTS      LAST TIME OF CONNECT
4 x.x.x.x      25      1089027660
x.x.x.x      5      1089028080
6 x.x.x.x      5      1089027240
x.x.x.x      5      1089021360
8 x.x.x.x      10     1089027240
x.x.x.x      5      1089021780
10 x.x.x.x      5      1089021780
x.x.x.x      10     1089027660
12 x.x.x.x      5      1089021360
x.x.x.x      10     1089027240
```

```
# Number of Connections
2 # Format: TIME NUMBER
4 12:03:00-2004-07-05      5295.6
12:10:00-2004-07-05      5905.8
12:17:00-2004-07-05      5903
6 12:24:00-2004-07-05      5908.6
12:31:00-2004-07-05      5950.4
8 12:38:00-2004-07-05      6046
```

### Post Analysis

The post analysis tool first reads the list with the suspicious addresses, then performs a subnet analysis. This analysis uses every IP address, and does a prefix matching as described in Chapter 2.3 with every other address. This leads to a problem with the complexity of  $O(n^2)$ .

At first, this calculation was made using Perl. Surprisingly we were not able to get a proper result within a useful amount of time. Using more than 1000 addresses resulted in calculation times of 15 minutes and more. Although we tried extensive optimization and analysis of the algorithm, we couldn't reduce the calculation time. As we tried to do the same in C, the (almost) same code needed about two seconds for 10000 addresses. Perl seems to be extremely slow when using big hash tables (this observation was made at several other points, by other people also).

### Output

In order to visualize our results, we add the suspicious addresses with their results for each time grid point. The running algorithm writes several files:

- a file with the extension `#basename_data.pdat` that contains a list with all the addresses, the corresponding result of the subnet analysis, the number of breaches it occurred in, and the time it breached last.

<sup>5</sup>IP addresses marked with "x.x.x.x" are faked for security reasons

- a file with the extension `#basename_plot.pdat` that counts the number of addresses that occurred in a breach at each time grid point.
- a file with the extension `#basename_plot.gnuplot` that sets every setting for GNUPlot to perform a nice plot in the file `#basename.eps`.
- finally a file called `#basename_number.gnuplot` that prints the number of logged in IRC clients. A nice plot can be seen in Figure 3.5. The obvious peak is the result of a DDoS attack of the Undernet IRC server. First the bots gather at the Undernet Server (increase of numbers) and then the IRC server breaks down.

Examples of these files can be found in Appendix D. A pseudo code of this implementation is in Appendix C. In despite to the longtime standing connection bot analysis we are counting IP addresses instead of the double `ClientIP:ServerPort`, since the number of hosts connected contain information enough to detect a bot during a fast joining sequence.

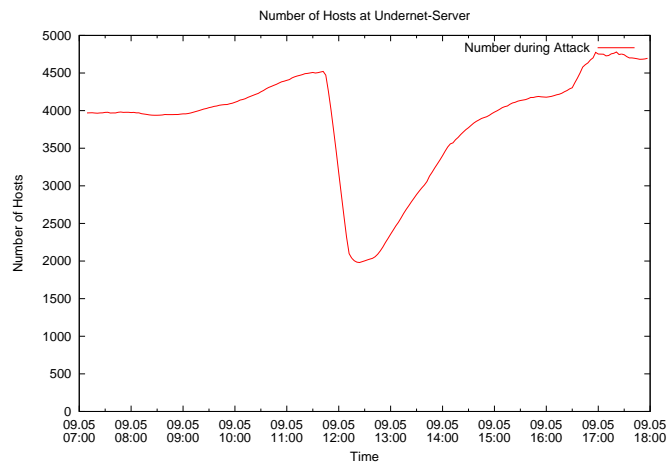


Figure 3.5: A plot of the number of logged in IRC clients on the 9th of May 2004

## 3.2 Online Analysis

### 3.2.1 Introduction

The Online Analysis should, in contrast to the Offline Analysis, be able to operate in real time. The framework “UPFrame” to perform these operations was provided by the DDoSVax group. UPFrame means UDP NetFlow Processing Framework and provides the necessary functionality to directly process incoming NetFlow records. What we had to do was to implement our functions using the provided netflowtools (`netflow_v5`). The incoming records are filtered for IRC ports and a predefined IRC server. Afterwards the records are processed differently for the long time standing connection bots and the fast joining bots.

### 3.2.2 “Longtime Standing Connection Bots” Implementation

The online implementation of the algorithm discussed in Chapter 3.1.3 is at first view more or less the same. But there are some differences. The basic functionality is the same, but how the data is processed is different. The flow chart is shown in Figure 3.3.

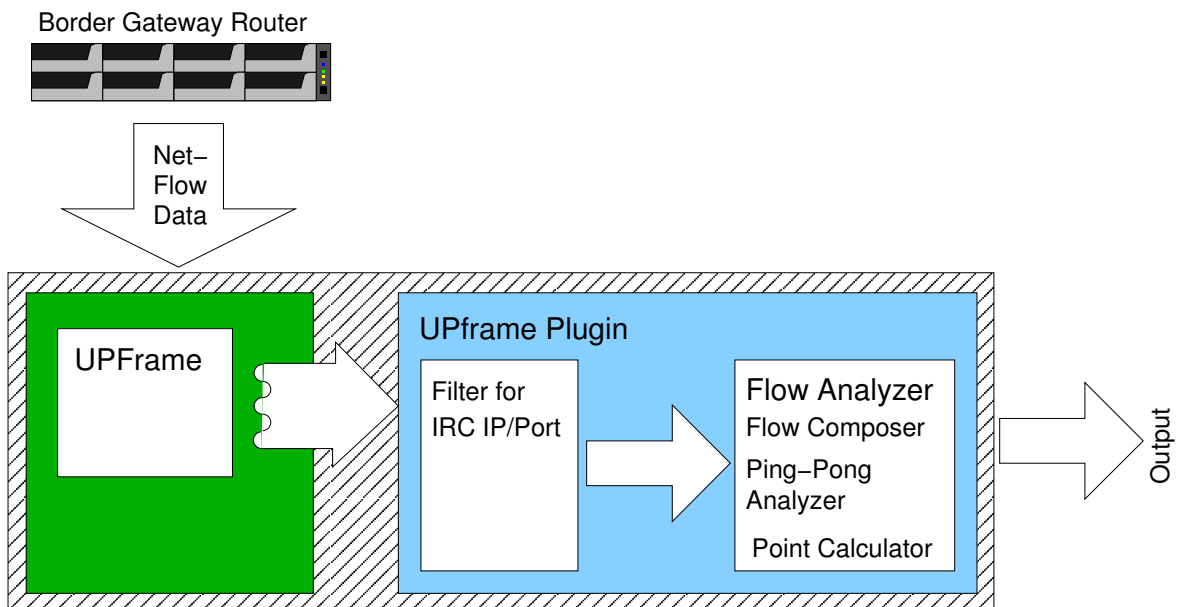


Figure 3.6: Online implementation of the longtime standing connection bots algorithm

### Definition of Connection

A connection is represented by the double **ClientIP:ServerPort**. See also Chapter 3.1.3.

### Filter for IRC Connections

Only flows are passed through, that are from the specific IRC server on the specific ports, respectively are going to the specific IRC server and specific ports<sup>6</sup>.

### Flow Analyzer

The flow analyzer takes every flow and checks, if it is a *Server to Client* or a *Client to Server* flow. A new connection is made, if the connection (identified by a double) is new, and updated if it already exists. Normally, a flow belonging to an existing connection, is newer and therefore we can update the end time of the existing connection with the end time of the flow. Unfortunately,

<sup>6</sup>6660-6669, 7000, 7777, 8000

there are some flows delayed and so, from time to time, we have to update the start time of the connection. Furthermore, every flow is checked, if it could be a Ping or Pong. If so, the flow analyzer tries to find its complement. Exactly like in the offline analysis, this is made for Ping's and Pong's. For more see Chapter 3.1.3.

How this works is shown in Figure 3.7.

Here we have a causal system, i.e. we cannot look to the future. The points have to be calculated before the connection is finished. Therefore, the points are calculated immediately.

For further understanding see the flow chart in Figure 3.7.

## Output

The output would be the same like in the offline analysis

A pseudo code of the online longtime standing connection bots algorithm can be found in Appendix E.

### 3.2.3 “Fast Joining Bots” Implementation

The implementation of the online analysis should perform the same actions as the offline analysis, with the restriction to not acquire a high latency, since every packet should be processed fast due to a restriction of the buffer size. Additionally the latency of the program as a whole should not be too big; we want to be able to obtain results exactly when they appear.

This made it impossible to make use of the moving average to flatten the results. It is predictable that the measurements with the online tool for the fast joining bots will not be as exact as those with the offline tool.

## Structure

Figure 3.8 shows how the online tools is structured. The implementation is basically the same as the one in the offline tool, the NetFlow flow records are processed (added to the bucket) and the results sent to the post processor. This module is the same as the one used for the offline analysis, also the output of all the modules is the same. (An example can be seen in Appendix D).

## Implementation

The data path of a NetFlow flow record can be seen in the flow chart in Figure 3.9, the data analysis is described in Figure 3.10.

Data is collected into a ring buffer as shown in Figure 3.11. This data type was chosen because it makes the breach detection much easier. In contrast to the Offline Analysis where the whole frame is searched for breaches, here we only compare the newest with the oldest ring buffer entry. If we detect a breach, the list of new IP addresses is already stored at the actual position of the ring buffer; this step was taken to reduce latency (the NEW list is refreshed with every incoming flow).

After seven minutes (when the bucket is filled) the post processor is called and performs its operation. Therefore an external process is started, this will, especially when applied on the DDoSVax Scylla Cluster, reduce load problems because the system will be able to give the post processor a lower priority than the data collector.

## Output

The output of the online analysis tool is the same as in Section 3.1.4, and some samples can be seen in Appendix D.



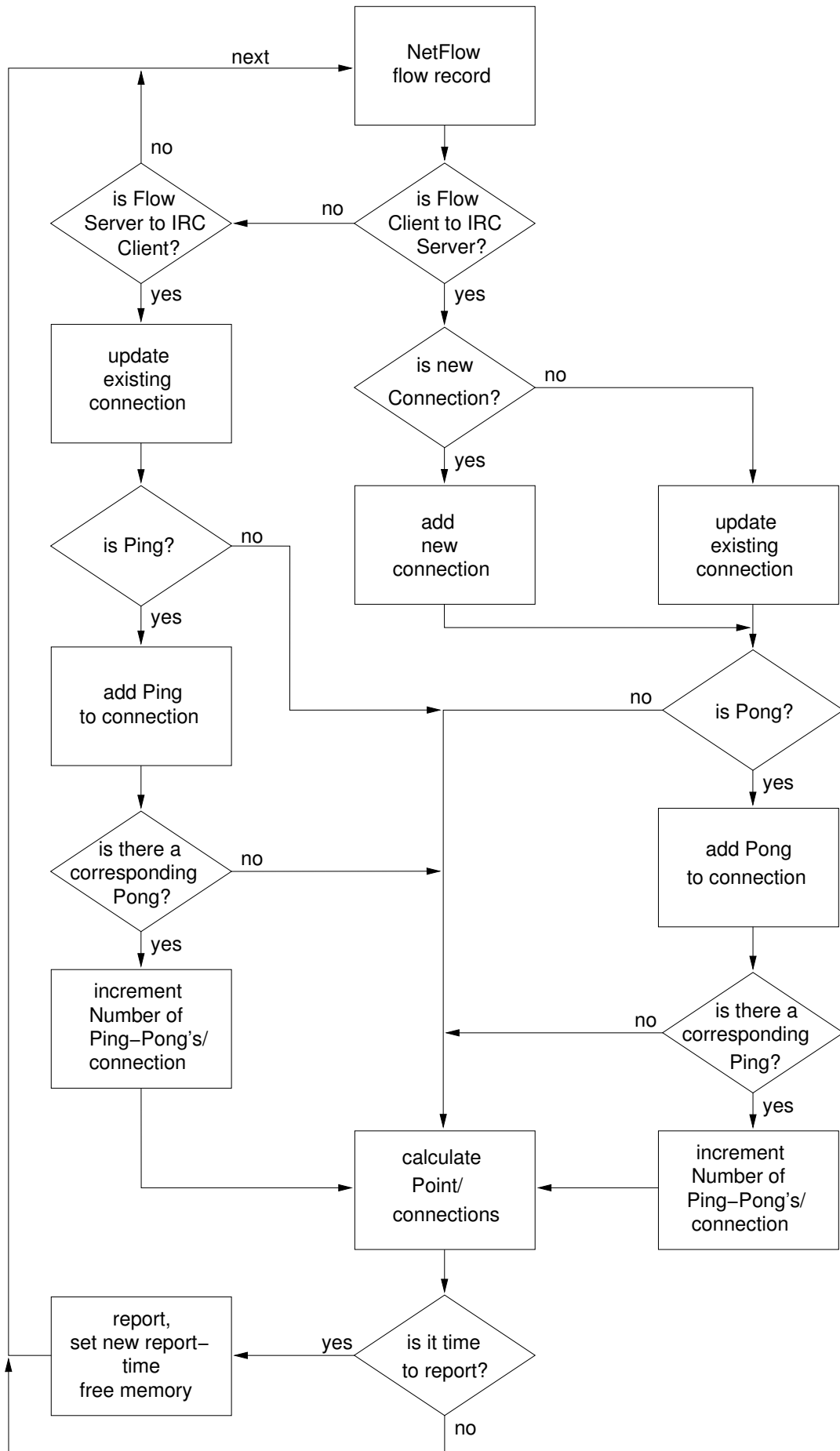


Figure 3.7: Flow chart of the online longtime standing connection bots algorithm

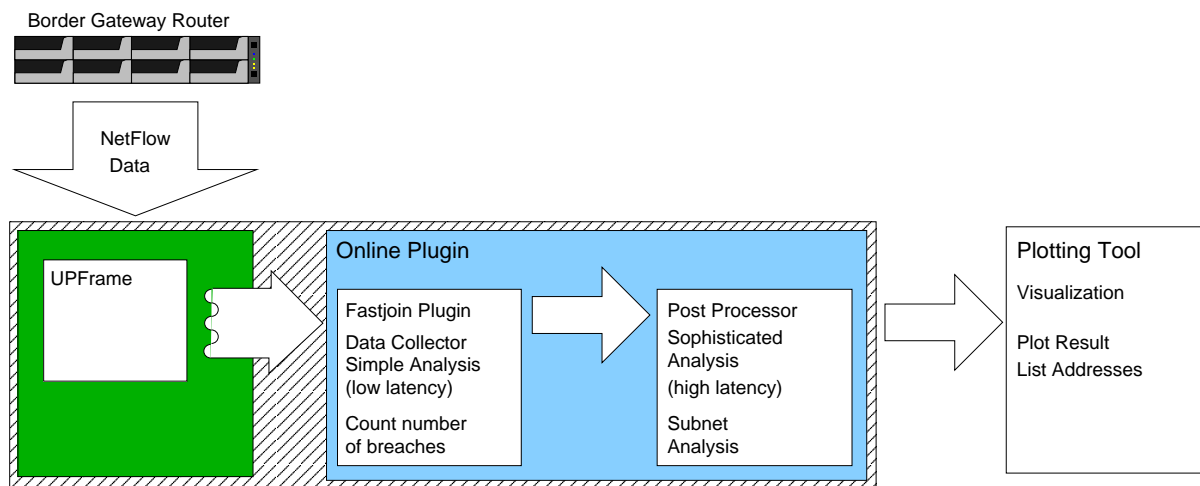


Figure 3.8: Structure of the online fast joining bots analysis

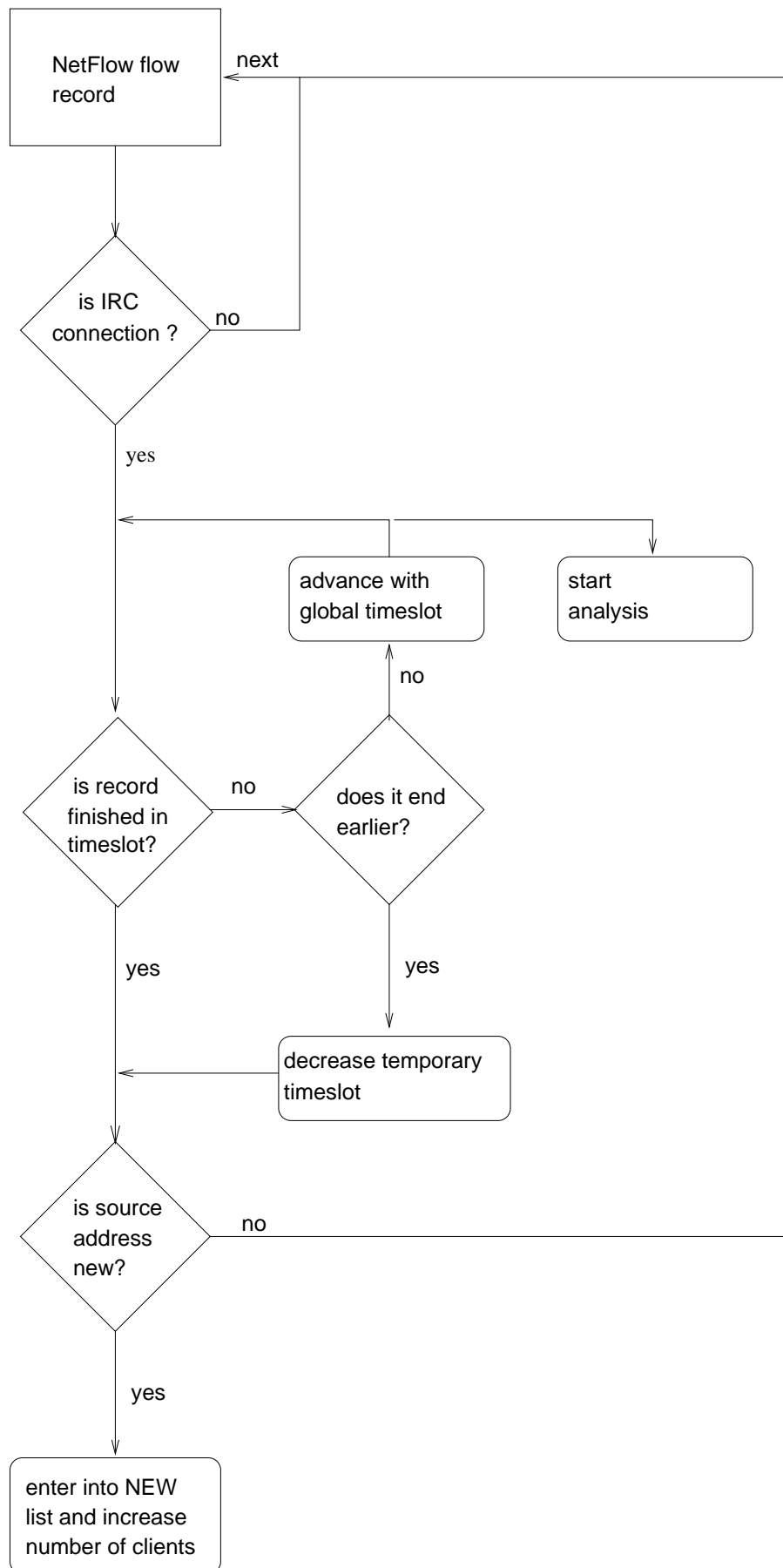


Figure 3.9: Flow chart of the data flow in the online analysis of the fast joining bot

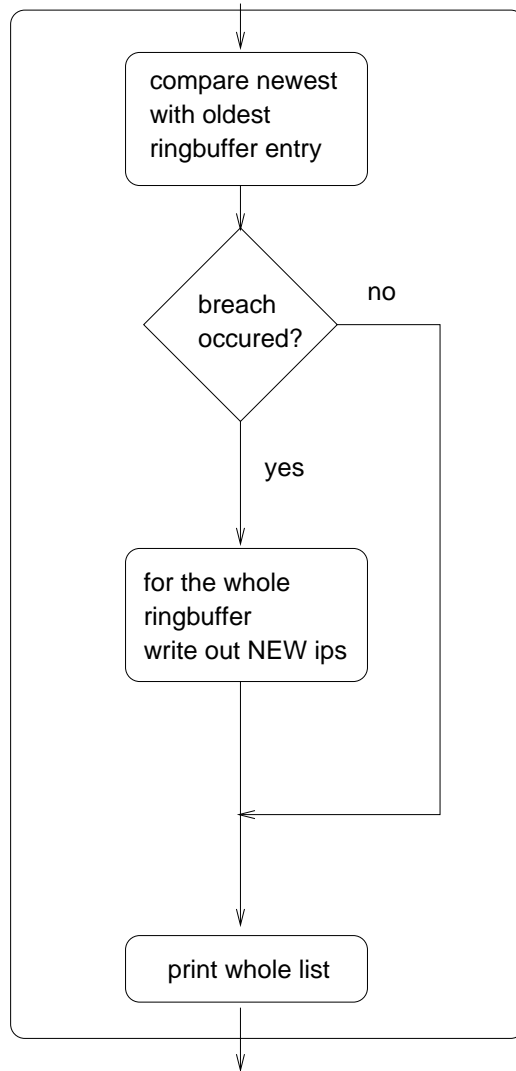


Figure 3.10: Detail view of the analysis part in the online analysis of the fast joining bot

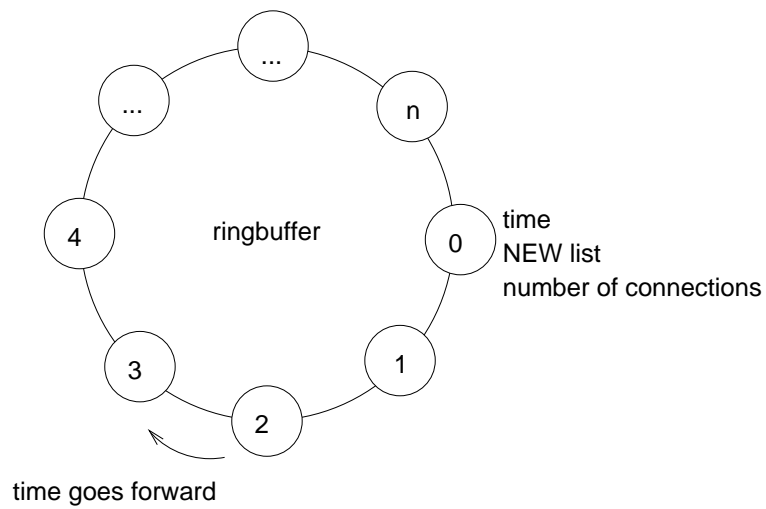


Figure 3.11: Realization of the data collection using a ring buffer

# Chapter 4

## Results

To validate our algorithms, we needed a testing environment. Fortunately we were able to use the “PlanetLab” network, where the DDoSVax Group is a part of. This project basically provides more than 400 hosts, distributed all over the world, where we could install and run different programs for testing purposes. Figure 4.1 shows the application of this network. We were able to simulate a fast joining scenario as well as long time standing connections.

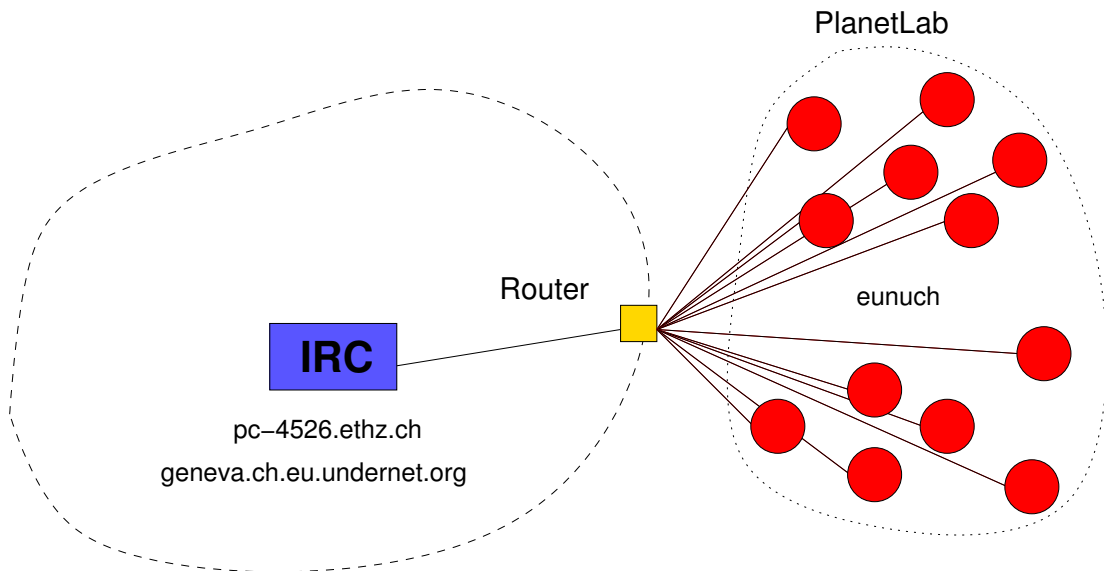


Figure 4.1: Setup of our testing environment

The program installed on the PlanetLab nodes was a modified version of the malicious “kaiten” bot, an IRC bot written in C. We removed the attack and malicious command support and called it “eunuch”. We control them using PSSH (see Appendix F).

### 4.1 Description of Measurement

The results described below mainly refer to a measurement taken at noon of the 5th of July 2004. At this time, we started our 150 bots, and sent them to the Undernet IRC server. The goal of the measurement was to first detect fast joining scenarios and analyze the longtime standing connection characteristics afterward.

### 4.2 “Longtime Standing Connection” Analysis

As said in Chapter 3.1.3, the output of the offline and the online analysis is a tabular. To give a fast overview over the analyzed time, or to see irregularities, we made a Perl script, that takes

this table and writes a gnuplot file. Such a plot is shown in Figure 4.2.

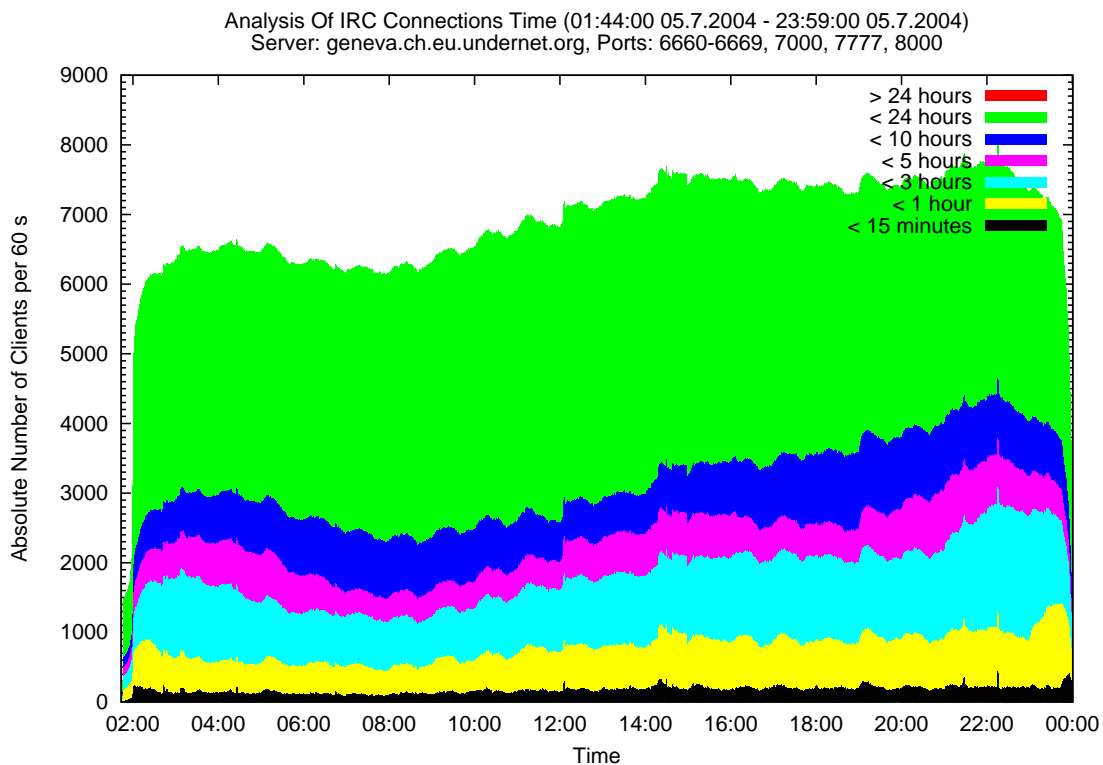


Figure 4.2: Analysis of IRC connections

In this plot, we can see, how many clients are online and (for) how long. Only the clients that are logged in on the geneva.ch.eu.undernet.org server are shown. The script takes an analyzed connection and draws it into the plot everywhere it is online. That means that if an IRC client is online for two hours, he would be blue for the whole two hours he is online. This is again an acausal analysis! We look several hours into the future in this case.

If we pay further attention to the plot in Figure 4.2, we can see, that there are some peaks. For example there is one at noon. There we have an abrupt rise of short connections. This piece of the plot is zoomed in in Figure 4.3.

This peak is two times bigger than the normal level. This is an appearance of a fast joining action, which is shown in the next chapter. Such a “longtime” plot can give further information about elapsed attacks, or are a good tool to proof a made supposition.

### 4.3 “Fast Joining Bots” Analysis

Concerning the measurement, the analysis should give some clear results:

- Detect Breach at the moment the bots were activated
- Find irregularities with help of the subnet analysis (the PlanetLab network consists of a compound of usually four up to ten or more nodes within one subnet)

The results obtained from the analysis are plotted in Figure 4.4. The plot contains several informations: on the left scale there is the accumulated number of hosts that were involved into a breach during this time. On the right hand side is the accumulated value gained through the subnet analysis, divided by the number of involved clients.  $(\frac{\sum P_{t_{subnet}}}{\#clients})$

This plot should give information about the importance of the breach (number of involved clients) and give a guess whether this breach contained hosts from the same subnet, or whether the

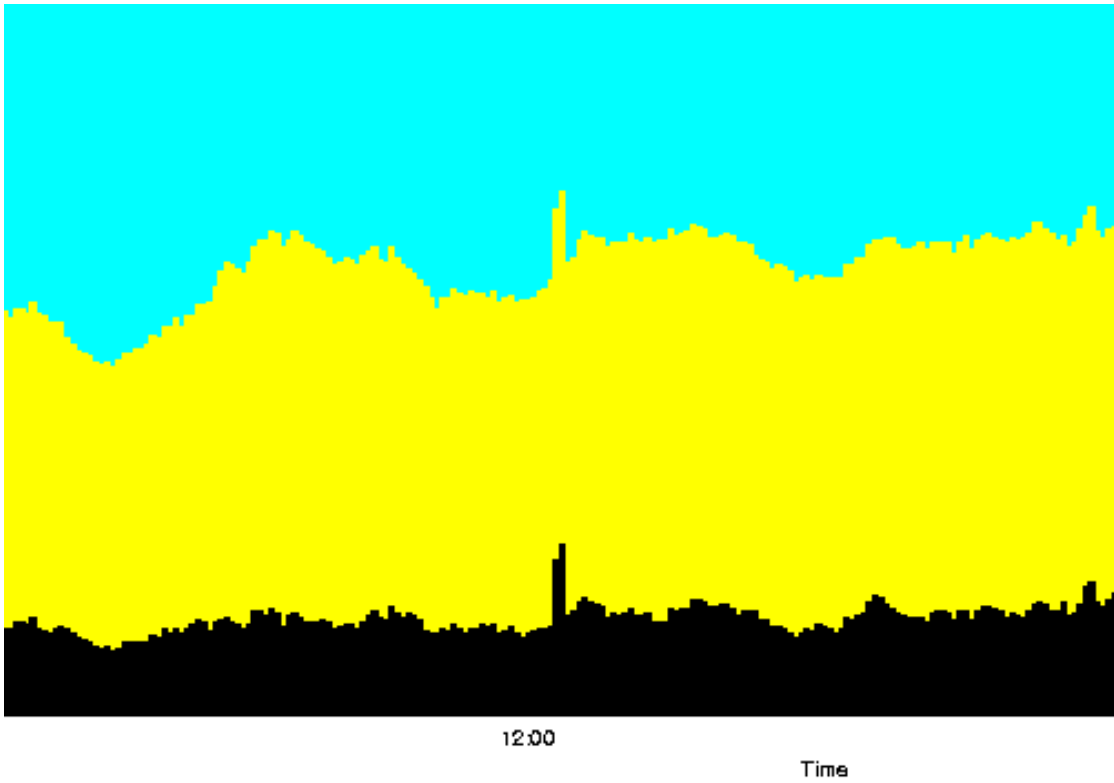


Figure 4.3: Zoom of peak in Figure 4.2

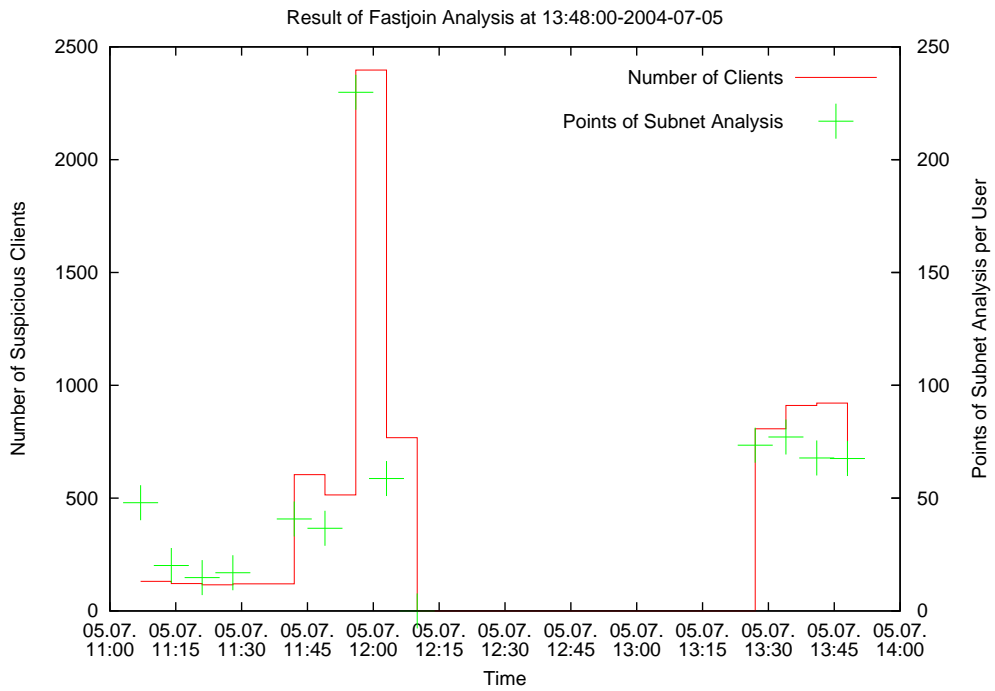


Figure 4.4: Result of measurement

nodes were randomly distributed all over the Internet. This case would result in a low result of the subnet analysis.

Taking a closer look at the plot of our measurement of the 5th of July gives a clear result: the arrival of the bot network was detected, the peak of the red graph shows a significant increase in this number. Additionally it results in a remarkable higher value of the subnet analysis. This shows that the analysis detected the relationship between the logged in clients that were involved into the breach!

Below listed is an extract of the data file given by the analysis<sup>1</sup>:

	Address	Subnet Analysis	Time
2	169.229.50.10	399	1089021780
4	169.229.50.11	399	1089021780
	169.229.50.12	398	1089021780
6	169.229.50.13	398	1089021780
	169.229.50.15	397	1089021780
8	...		
	x.x.x.x	0	1089021360
10	x.x.x.x	0	1089027660
	x.x.x.x	0	1089027660
12	x.x.x.x	0	1089021360

The clear difference between the upper and the lower addresses inspired us to further investigation about the origin of these hosts. A DNS lookup of the first five addresses showed, that they all came from the PlanetLab network, where our bot were installed.

<sup>1</sup>IP addresses marked with “x.x.x.x” are faked for security reasons



# Chapter 5

## Summary

### 5.1 Conclusion

In this chapter we try to conclude the results of our analysis and to outline the possibilities of the examined methods to detect the two Bot types.

#### 5.1.1 Longtime Standing Connection Bots

##### Results

We have developed a tool, that is able to analyze IRC connections in Cisco NetFlow data, if the IRC server is known.

Our algorithm can show, everything included in the NetFlow data for each connection. That especially is:

- the *online time* of each IRC client
- the *amount of data*

Further we can show, if an IRC client is idle or not and how many Ping-Pong's were sent. We can also proof the results by comparing the output of this algorithm with the one of the fastjoining tool.

##### Limits

As said in the chapter before, we are able to display nearly everything about the connection, that was saved in the NetFlow data. But with all this, we haven't found the malicious bots.

If for example, we have a "human chatter" that logs into an channel in the morning, and then stays online, waiting for some action in the channel, then we will have a typical non-talkative longtime standing connection. This sample is far away from special. IRC often is used for a kind of IM (Instant Messenger), or last but not least, for some kind of Groupware what would end in extremely long time standing connection (several weeks!) with nearly no traffic!

On the other hand, IRC also often is used for good-natured bots like file sharing purposes. Because these are also bots with "non-human" characters, it will be nearly impossible to make a difference between *malicious* and *good-natured* bots.

Considering this, we will have a very high number of false positives, if we would try to detect malicious bots with the defined properties. Nevertheless, the properties give knowledge for further analysis.

#### 5.1.2 Fast Joining Bots

##### Results

During our work on the fast joining algorithm we developed several algorithms and methods:

**Number counter:** The developed algorithm is able to count the number of logged in clients out of NetFlow flow records. This means that we were able to concentrate the data and gain new information out of it.

**Breach detection:** This method can detect the breach out of a given frame. It will report all the addresses that were involved in a breach.

**Subnet Analysis:** The last method shows the relationship between addresses, whether they belong to the same subnet. This is a reliable help to decide, whether hosts may contain malicious Bots or whether they accidentally joined an IRC server at the same time as a gathering of Bots occurred.

This methods were successfully evaluated and can serve as a tool to detect breaches.

### Limits

Although the success of our measurement, there are some obvious limits. At first, the clients at the IRC server fluctuate approx. 20% every seven minutes. This means that whatever data we obtain, these 20% will always give false positives. We tried to diminish this problem by introducing the subnet analysis, but this analysis mainly shows which subnets were involved. If there are Bots installed on single hosts with no relationship to others, they will result as a false negative.

Another problem is the blurriness in the measurement of the number of logged in IRC clients. This number sometimes jumps with values of more than 100 clients. The calculation of the moving average solved this problem, but this wasn't possible using the online version. There this problem remains.

## 5.2 Outlook

### 5.2.1 Longtime Standing Connection Bots

As said in the Chapter 5.1.1, we are able to display some properties of an IRC connection, but it's nearly impossible to assign these values to a malicious bot. Therefore there should be further analysis about the behavior of this bots to trim the made point scale.

A home user, infected with a malicious bot, is normally offline during night. A lot of them are only online after work, i.e. five o'clock. Hence, a daytime analysis could be interesting.

Further, today a lot of Internet users have a broadband Internet Access. So many of them don't change the IP numbers, respectively get the same IP on the next login. So an analysis over more than one connection could also be interesting, to see regularities.

We found different values and made an additive rating. Perhaps it is better to make a multiplicative scale, or in general, put the values in touch with each other.

### 5.2.2 Fast Joining Bot Detection

It is well possible to imagine the use of these tools to observe other chatting systems, as for example IM or ICQ (in the ICQ system there already exists a Bot called "Phat Bot").

Another possible application of our methods might be to detect load problems for web servers or other public services. The breach detection would be a useful tool.

A further examination of the subnet analysis could give very interesting results. Since this algorithm became quite fast due to strict optimization, an application in a real time environment is possible.

## 5.3 Acknowledgment

We would like to thank our tutor Thomas Duebendorfer and our co-tutor Arno Wagner for their support during our work on this thesis and their patient answers to so many desperate questions. An additional "thank you" concerns their tolerance on our working method on this report

(Thomas Kaufmann has left Switzerland in the middle of July for an internship abroad and the documentation had to be written on two different continents).

Most of all we are grateful to them and our supervisor Prof. Dr. Bernhard Plattner to give us the opportunity to apply our first thesis in such a sophisticated and professional environment; it is incredible what insight we gained within 3 month of intensive work on this interesting subject.

# Appendix A

## Deployment

### A.1 Introduction

Here we will describe how our programs actually work. We describe the necessary commands to obtain the results we did so far. A big part is the description of the Perl framework we worked on, with all the necessary functions to perform the analysis.

### A.2 Framework in a Nutshell

#### A.2.1 longtime.pl

The `longtime.pl` is the offline longtime standing connection detection tool. The usage is as follows:

- b <Identification> Special Identification to identify the conducted analysis
- d <Path> Path to NetFlow files
- h Usage (Help)
- l <Minutes> Output length (gives output, every 'l' minutes. Default: only one file is reported)
- t <Seconds> Longtime threshold (every Connection, shorter than 't' won't be reported. Default: Every Flow will be reported)
- n <Path> Path to the 'netflow\_to\_text' program (only path!)
- o <Path> Outputpath
- s <Date/Time> Startpoint of the filter (format: "13.05.2004 00:00:00")
- e <Date/Time> Endpoint of the filter (format: "13.05.2004 00:00:00")

The input are NetFlow files and the output is a file in form of a tabular. It would look like this<sup>1</sup>:

```
2 #
  # Result of offline plugin 'longtime'
  # File written at: 05.07.2004 14:18:34
  # Using the following scheme:
  # CLIENTIP SERVERPORT BYTES_TO_SERVER BYTES_TO_CLIENT PACKETS_TO_SERVER PACKETS_TO_CLIENT ←
  # CONNECTIONS_TO_SERVER PINGPONGS POINTS[] STARTTIME ENDTIME
  6 # POINTS[CON_TIME          BYTES_TO_SERVER QUOTIENT          PINGPONG]
  #
  8 #
  x.x.x.x 6667      720      0          15         0          2          0          4          1          0          0          ←
     1087924924.3660 1087926620.6720
 10 x.x.x.x 6667      288      0           6          0          2          0          2          1          0          0          ←
     1087925170.4440 1087926256.3200
  x.x.x.x 6667      144.0    0           3.0       0          1          0          0          1          0          0          ←
     1087926431.2970 1087926440.1290
 12 x.x.x.x 6667      288      0           6          0          1          0          1          1          0          0          ←
     1087924567.4390 1087924917.2000
  x.x.x.x 6667      288      0           6          0          1          0          0          1          0          0          ←
     1087923577.8140 1087923713.4950
```

<sup>1</sup>IP addresses marked with "x.x.x.x" are faked for security reasons

14	x.x.x.x	6667	311961	390418	6013	6943	1	0	0	69112	13	0	↔
			1087927185.5900	1087927192.8230									
	x.x.x.x	6667	128.0	473	3.0	4	1	0	0	1	37	0	↔
			1087925722.7570	1087925733.9570									
16	x.x.x.x	6667	73888	83128	1373	840	1	0	8	1	11	0	↔
			1087923253.3350	1087926683.4570									
	x.x.x.x	6667	448760	550995	8507	9858	1	0	0	766352	12	0	↔
			1087927189.0480	1087927192.1210									
18	x.x.x.x	6667	3523	4277	59	48	1	0	6	1	12	0	↔
			1087924090.2640	1087926894.4630									
	x.x.x.x	7000	3896	4428	66	51	1	0	6	1	11	0	↔
			1087924087.1260	1087926891.6480									

To use the `longtime.pl` tool, the `NetflowTools.pm` have to be in the same directory!

## A.2.2 fastjoin.pl

The `fastjoin.pl` is the offline fast joining bot detection tool. The usage is as follows:

```
-b <Identification>  Special Identification to identify analysis. This name will be used for all further
                      files
-d <Path>             Path to NetFlow files
-h                   Usage (Help)
-n <Path>            Path to the 'netflow_to_text' program (only path!)
-o <Path>            Outputpath
-s <Date/Time>       Startpoint of the filter (format: "13.05.2004 00:00:00")
-e <Date/Time>       Endpoint of the filter (format: "13.05.2004 00:00:00")
```

## A.2.3 NetflowTools.pm

`NetflowTools.pm` is a library with useful functions, managing NetFlow files. The following functions are in it:

<b>ip2int</b>	<pre>int ip2int(string IP);</pre> <p>Convert the human readable IP (like: 132.129.57.115) and return the int value.</p>
<b>ip2bin</b>	<pre>array ip2bin(string IP);</pre> <p>Convert the decimal human readable IP (like: 132.129.57.115) and return the binary IP (10000100100000011110011110011);.</p>
<b>ipsplit</b>	<pre>int ipsplit(string IP);</pre> <p>Convert the human readable IP (like: 132.129.57.115) and return an array with four entries (array[0]=132 ...).</p>
<b>dec2bin</b>	<pre>int dec2bin(string IP);</pre> <p>Convert decimal value to a binary value.</p>
<b>checkipmatch</b>	<pre>int checkipmatch(string IP);</pre> <p>Return the value or the prefix matching.</p>
<b>getNextGridPoint</b>	<pre>int getNextGridPoint(float UNIX_timestamp);</pre> <p>Return next lower gridpoint. For this function the constant GridLength [s] has to be set in the main program first!</p>
<b>round</b>	<pre>int round(float number);</pre> <p>Round like in other languages.</p>
<b>getLokaleZeit</b>	<pre>string getLokaleZeit();</pre> <p>Returns the actual date and time as a string (Format: 13.05.2004 13:18:33)</p>
<b>stamp2date</b>	<pre>string stamp2date(float UNIX_timestamp);</pre> <p>Returns the actual date as a string (Format: 13.05.2004)</p>
<b>stamp2time</b>	<pre>string stamp2time(float UNIX_timestamp);</pre> <p>Returns the actual time as a string (Format: 13:18:33)</p>
<b>epochseconds2dmyhms</b>	<pre>string epochseconds2dmyhms(float UNIX_timestamp);</pre> <p>Returns the actual date and time as a string (Format: 13.05.2004 13:18:33)</p>
<b>dmyhms2epochseconds</b>	<pre>float dmyhms2epochseconds(sting datetime);</pre> <p>Convert human readable date/time (Format: 13.05.2004 13:18:33) and returns the UNIX timestamp</p>
<b>getLogTime</b>	<pre>float getLogTime(string NetFlowFileName);</pre> <p>Return the log time as a UNIX timestamp. Input is the name of the NetFlow file</p>
<b>getEintraegeZeile</b>	<pre>void getEintraegeZeile(string Zeile);</pre> <p>Input is a line from the netflow_to_text tools (netflow_to_text -d -f ...). The function return the values: Protokoll; SourceIP; DestinationIP; SourcePort; DestinationPort; AnzahlBytes; AnzahlPakete; Anfangszeit (seconds.milliseconds); AnfangszeitMSEK; AnfangszeitSEK; Endzeit (seconds.milliseconds); EndzeitMSEK; EndzeitSEK; Dauer (seconds.milliseconds); DauerMSEK. By calling this function, the values are accessible.</p>
<b>getFilesToProcessInFolder</b>	<pre>float getFilesToProcessInFolder(string Path);</pre> <p>Return a sorted list (sorted by date/time of the logtime) of NetFlow files in Path. Only NetFlow files are returned, even if there are others.</p>

### A.2.4 `longtime_plot.pl`

The `longtime_plot.pl` is the plotting script for the output files generated by `longtime.pl` of the UPFrame plugin '*longtime*'. The usage is as follows:

```
-b <Identification>   Special Identification to identify the conducted analysis
-d <File>             File to process
-h                   Usage (Help)
-o <Path>            Outputpath
```

The output is a `.dat` file and a `.gplot` file, which can be plotted by calling `gnuplot Identification.gplot`. The output of this command is a file called `Identification.eps`. The file looks like Figure 4.2.

## Appendix B

# Pseudo Code “Offline Longtime Standing Connection” Algorithm

```
#####
2 # Pseudo code of the "Long Standing IRC Bots Detection" algorithm #
#####
4
6 # l: length of flow (Bytes)
6 # p: number of packets in flow
8 # s: start time of flow
8 # e: end time of flow
10 # connection key always is [ClientIP:ServerPort] and is swaped if necessary
12 # OutputTimeOut: Time between Output is reported
14 # Step 1a: Generating the data structure of connections and
14 # Ping-Pong analysis using the "Sliding Window" technique
16 #####
18 for (each FlowFile) {
20     for (each Flow) {
22         if (Flow is an IRC Connection)
24             addToConnectionList(Flow);
26             addToPingPongList(Flow);
28
30             if ((window(next FlowFile) != window(FlowFile)) ||
32                 OutputTimeout is reached
34                 ) {
36                     doEvaluation();
38                 }
40             }
42         }
44     }
46 # Step 1b: addToConnectionList
46 #####
48 addToConnectionList(Flow) {
50     if (DstIP == IRCIP) {
52         if (Flow exists in Connections) {
54             update Flow #start or end time
56         }
58         else {
60             add Flow to Connections
62             set dataToClient 0 in this Flow
64         }
66     }
68     else if (SrcIP == IRCIP) {
70         #reverse means other direction
72         update reverse(Flow) #dataToClient only
74     }
76 }
78 # Step 1c: addToPingPongList
78 #####
80 addToPingPongList(Flow) {
82     if (
84         (DstIP == IRCDP) &&
86         (((l >= 46) && (l <= 121) && (p == 1)) ||
88         ((l >= 86) && (l <= 173) && (p == 2))
90     ) {
92         if (connection allready extist in this timeslot) {
94             mark this PongCandidate as deleted
96         }
98     }
100 }
```



```

62         }
63         else {
64             add Flow to PongCandidates in corresponding timeslot
65         }
66     }
67     else if ((SrcIP eq IRCIP) && (p >= 2) && (l >= 86)) {
68         if (connection already exist in this timeslot) {
69             mark this PingCandidate as deleted
70         }
71         else {
72             add Flow to PingCandidates in corresponding timeslot
73         }
74     }
75 }
76
77 # Step 2: Evaluating collected data
78 #####
79 doEvaluation () {
80     for (each closed Connection) {
81         #closed means totally inactive for 20 minutes
82         add "calculated points depending on connection time" to Points
83         add "calculated points depending on amount of sent data" to Points
84         add "calculated points depending on number of connections" to Points
85         add "calculated points depending on quotient of amount of sent/received data" to ←
86         Points
87
88         #evaluating ping-pong
89         for (each Ping) {
90             if (corresponding PongCandidate is in the same timeslot or in the next ←
91                 timeslot) {
92                 # s1: start time of ping
93                 # e1: end time of ping
94                 # s2: start time of pong
95                 # e2: end time of pong
96                 #if ((s2 <= s1) && (e2 >= e1)) {
97                 if ((s1 <= s2) && (e1 >= e2)) {
98                     --> found ping-pong
99                     add "calculated points dependet on number of ping-pongs" ←
100                     to Points
101                 }
102             }
103         }
104     }
105     write found connections with appropriate points to output file
106 }
107
108 # Appendix: Used Functions
109 #####
110 getNextGridPoint (timeStamp) {
111     #the time is splitted into timeslots of 180s, mesured from 01.01.1970
112     return next smaller timeStamp;
113 }
114
115 # Data Stucture
116 #####
117 # {...} means, that this is a hash-key
118
119 #Connections
120 {ClientIP:ServerPort} || startTime | endTime | #Packets | #BytesToServer | #Connection | # ←
121     BytesToClient
122
123 #PingCandidates
124 {ClientIP:ServerPort}{timeSlot} || startTime | endTime | #Packets | #BytesToClient | # ←
125     deletedFlag
126
127 #PingCandidates
128 {ClientIP:ServerPort}{timeSlot} || startTime | endTime | #Packets | #BytesToServer | # ←
129     deletedFlag

```

## Appendix C

# Pseudo Code “Offline Fast Joining Bot Detection” Algorithm

```
#####
2 #
# Pseudo Code for the Offline Analysis of Fastjoining Bots
4 #
# Thomas Kaufmann <thomaska@ee.ethz.ch>
6 #
#####
8
window:          1 hour
10 time slot:     420s (to be sure that there is at least one ping/pong sequence)
boundary_time:   4 time slots
12 boundary_top:  120 hosts

14 #####
#
16 # Step 1: Data Aquisition

18 for all windows within specified time {
   for each flow within a window {
20       if (within IRC port range and destination address is IRC server) {
           doProcessing                               //enter only IRC netflows
22       }
   }
24 }

26 #####
#
28 # Step 2: Data Processing

30 doProcessing {
   for incoming flow {
32       for every time slot {                       // calculate number of hosts
           // and reorganize CONNECTION list
34           increase number of clients within this time slot into NUMBER list
           enter address into IPLIST
36       }
   }
38 }

40 calculateAverage {
   for all entries in NUMBER list {                 // calculate average of NUMBER list
42       calculate moving average and enter into AVERAGE list
   }
44 }

46 #####
#
48 # Step 3: Data Analysis

50 findBoundaryBreach {                               // find moments where jump in the
   // number of connection occurs
52   for each time slot t_1 look into future until t_1+boundary_time
       and search for upper boundary breach
54   if boundary breach occured at t_n{
       searchNewHost()
56       enter every address of NEW list from t_1 to t_n into
           SUSPICIOUS list or increase number of breaches if already existent
58   }
   look until t_1+boundary_time and find out if further breaches detected
60   if yes {
       enter these addresses too into SUSPICIOUS list or increase number of breaches
   }
}
```

## APPENDIX C. PSEUDO CODE "OFFLINE FAST JOINING BOT DETECTION" ALGORITHM

```
62     }
64     delete old data
65 }
66
68 searchNewHost {
69     for all time slots in AVERAGE list {
70         for each address of IPLIST there { // find out new addresses for every time slot
71             if address unknown in last time slot {
72                 enter address into NEW list
73             }
74         }
75     }
76 }
77
78 #####
79 #
80 # Step 4: Data Output
81
82 printList {
83     for every address in SUSPICIOUS list {
84         print suspicious address
85         print number of breaches of this address
86         print the last time this address committed a breach
87     }
88 }
89
90 #####
91 #
92 # Step 5: Post Analysis
93
94 readEntries {
95     read all entries from the output file generated above
96     generate POINT list
97     generate TIME list
98 }
99
100 subnetAnalysis {
101     for all addresses in IPLIST {
102         for all addresses in IPLIST {
103             if addresses are different and if they were recorded at the same time ↔
104                 for the last time {
105                     do a prefix matching of the two binary addresses
106                 }
107             if result > 16 Bit {
108                 accumulate and enter into POINT list and TIME list
109             }
110         }
111     }
112 }
113
114 generatePointList {
115     for every entry in POINT list {
116         print address, number of connects, the subnet points and the time it occurred ↔
117         last time
118     }
119 }
120
121 generateTimeList { // prepare gnuplot data
122     for every entry in TIME list {
123         sort time
124         print time, number of suspicious users at this time, subnet points and subnet ↔
125         points per user
126     }
127 }
128
129 generateGnuplotFile {
130     prepare a GnuPlot configuration file with all the necessary settings, inclusive time, ↔
131     scale, ...
132 }
133
134 doPlot {
135     execute gnuplot with the produced gnuplot file
136 }
137
138 #####
139 #
140 # Data Structures:
141
142 IPLIST:
143 key is time stamp
144 element is a client ip address
145
146 NUMBER list:
```

```
146 key is a time slot
    element is the number of logged in clients at this time slot

148 NEW list:
    key is a time slot
    element is a list of new client ip addresses for this time slot

152 POINT list:
    key is a address
    elements are the number of breaches, the time the last breach occurred and the result of the ←
        subnet analysis

156 TIME list:
    key is a time stamp
    elements are the number of clients for this time stamp, accumulated number of breaches of all ←
        addresses during this time stamp and the result of the subnet analysis
```

## Appendix D

# Output of Fast Joining Bot Analysis

The actual data we were seeking for which assigns our results to the distinguished IP addresses (#basename\_data.pdat)<sup>1</sup>:

```
# Format:
2 # Time Number of Clients that breached Number of Breaches Points of Subnet Analysis ←
   Points of Subnet Analysis per User
x.x.x.x      5      30      1089021780
4 x.x.x.x      5      30      1089021780
x.x.x.x      10     0      1089027660
6 x.x.x.x      5      0      1089021360
x.x.x.x      10     0      1089027240
8 x.x.x.x      5      0      1089020520
x.x.x.x      10     0      1089021780
10 x.x.x.x      5      0      1089020520
x.x.x.x      15     0      1089027240
12 x.x.x.x      5      0      1089021360
x.x.x.x      5      61     1089021780
14 x.x.x.x      5      61     1089021780
x.x.x.x      5      60     1089021780
16 x.x.x.x      10     0      1089027240
x.x.x.x      10     0      1089027660
18 x.x.x.x      5      0      1089020520
x.x.x.x      5      0      1089021360
20 x.x.x.x      5      0      1089021360
x.x.x.x      5      0      1089020940
22 x.x.x.x      5      0      1089021360
24 ...
26 169.229.50.5  5      397     1089021780
169.229.50.6  5      397     1089021780
28 169.229.50.7  5      397     1089021780
169.229.50.8  5      399     1089021780
30 169.229.50.9  5      399     1089021780
```

The data put into a format that can be easily printed and show significant occurrences(#basename\_plot.pdat)<sup>2</sup>:

```
# Format:
2 # Time Number of Clients that breached Number of Breaches Points of Subnet ←
   Analysis Points of Subnet Analysis per User
11:07:00-2004-07-05 131 655 6286 47.984733
4 11:14:00-2004-07-05 121 605 2435 20.123967
11:21:00-2004-07-05 116 610 1714 14.775862
6 11:28:00-2004-07-05 120 630 2032 16.933333
11:42:00-2004-07-05 604 3175 24634 40.784768
8 11:49:00-2004-07-05 514 2675 18833 36.640078
11:56:00-2004-07-05 2397 12090 551045 229.889445
10 12:03:00-2004-07-05 768 3895 45050 58.658854
13:27:00-2004-07-05 807 5600 59293 73.473358
12 13:34:00-2004-07-05 911 6300 70251 77.114160
13:41:00-2004-07-05 921 7020 62436 67.791531
14 13:48:00-2004-07-05 615 4600 41550 67.560976
```

<sup>1</sup>IP addresses marked with "x.x.x.x" are faked for security reasons

<sup>2</sup>IP addresses marked with "x.x.x.x" are faked for security reasons

## Appendix E

# Pseudo Code “Online Longtime Standing Connection” Algorithm

```
#####
2 # Pseudo code of the "Long Standing IRC Bots Detection" algorithm #
# implemented as an plugin for the UPFrame #
4 #####

6 # l: length of flow (Bytes)
# p: number of packets in flow
8 # s: start time of flow
# e: end time of flow

10 # PINGPONGTIMEOUT = 180 seconds

12 # connection key always is [ClientIP:ServerPort] and is swaped if necessary
14

16 # this function is called by the UPFrame for each flow header
addConnection(Header) {
18 #each Header has a number of flows
    for (each Flow) {
20         if (is IRC Connection to IRC Server)
                key = ClientIP:ServerPort
22                 if (Flow exists in Connections) {
                        Connections{key}.lastActivity = Connections{key}.endTime;
24                         update Connections{key} #start or end time and the other values
                        #look, if this flow could be a Pong
26                         if (((l >= 46) && (l <= 121) && (p == 1)) ||
                                ((l >= 86) && (l <= 173) && (p == 2))
28                                 ) {
                                Connections{key}.pongStartTime = s
30                                Connections{key}.pongEndTime = e
                                if (Connections{key}.pingStartTime <= Connections{key}. ←
                                        pongStartTime &&
32                                Connections{key}.pingEndTime >= Connections{key}. ←
                                        pongEndTime &&
                                        Connections{key}.lastActivity+PINGPONGTIMEOUT < ←
34                                Connections{key}.pingStartTime
                                        ) {
                                                Connections{key}.numberOfPingPong++
36                                }
                                }
38
                                }
40         else {
                add Flow to Connections
42         }

44         calculatePoints(key)
    }
46     else if (is IRC Connection to IRC Client) {
            key = ClientIP:ServerPort
            update Connections{key}
            #look, if this could be a Ping
48             if ((p >= 2) && (l >= 86)) {
                    Connections{key}.pingStartTime = s
50                    Connections{key}.pingEndTime = e
                    if (Connections{key}.pingStartTime <= Connections{key}. ←
                            pongStartTime &&
52                    Connections{key}.pingEndTime >= Connections{key}. ←
                            pongEndTime &&
                            Connections{key}.lastActivity+PINGPONGTIMEOUT < ←
54                    Connections{key}.pingStartTime
                    ) {
                            Connections{key}.numberOfPingPong++
                    }
            }
    }
}
```

```

56         ) {
57             Connections{key}.numberOfPingPong++
58         }
59     }
60     calculatePoints(key)
61 }
62
63     if (is Time to report) {
64         writeFile()
65         give unused memory free
66     }
67 }
68
69
70
71
72 # Data Structure
73 #####
74
75 #[...] means, that this is a hash-key
76
77 #Connection is a Hash of FlowEntries, a struct
78
79 #typedef struct {
80 #     unsigned int startTime;
81 #     unsigned int endTime;
82 #     uint32_t packetsToServer;
83 #     uint32_t packetsToClient;
84 #     uint32_t bytesToServer;
85 #     uint32_t bytesToClient;
86 #     unsigned int connectionsToServer;
87 #     unsigned int connectionsToClient;
88 #     unsigned int dataSentPoints;
89 #     unsigned int dataReceivedPoints;
90 #     unsigned int connectionsToServerPoints;
91 #     unsigned int connectionsToClientPoints;
92 #     unsigned int dataQuotientPoints;
93 #     unsigned int connectionTimePoints;
94 #     unsigned int numberOfPingPong;
95 #     unsigned int pingPongPoints;
96 #     unsigned int lastActivity;
97 #     unsigned int pingStartTime;
98 #     unsigned int pingEndTime;
99 #     unsigned int pongStartTime;
100 #     unsigned int pongEndTime;
101 #} flowentry;
    
```

## Appendix F

# Starting Bots on a Huge Number of Hosts Using PSSH

This script called `startpssh #hostlist` starts our bot on all hosts given in the file passed as first argument. After a timeout of three seconds for each host, the start process will be canceled for this host (see the `-t` option).

```
#!/bin/sh
2 # Start a program defined in this script on a huge number of hosts
#
4 # USAGE: startpssh $hostlist
6 UNAME=ethz_ddosvax
  PROG=/home/$UNAME/eunuch
8
  pssh -h $1 -t 3 -l $UNAME $PROG
```



# Bibliography

- [1] *DDoSVax*,  
<http://www.tik.ee.ethz.ch/~ddosvax/>.
- [2] *LEO - Link Everything Online*,  
<http://dict.leo.org/>.
- [3] *SWITCH - The Swiss Education & Research Network*,  
<http://www.switch.org/>.
- [4] Lukas Ruf, *Latex Essentials – HowTo Create Your LaTeX-based Documentation*, TIK, ETH Zuerich, 2002.
- [5] Stéphane Racine, *Analysis of Internet Relay Chat Usage by DDoS Zombies*,  
<ftp://www.tik.ee.ethz.ch/pub/students/2003-2004-Wi/MA-2004-01.pdf>.
- [6] UPFrame, *An Extendible Framework for the Reception and Processing of UDP Data*, TIK, ETH Zuerich, 2004,  
<http://www.tik.ee.ethz.ch/~ddosvax/upframe/>.
- [7] Ove Ruben R Olsen *irciiman.txt*,  
<http://www.irchelp.org/irchelp/ircii/irciiman.txt>.
- [8] *The Undernet IRC network*,  
<http://www.undernet.org/>.
- [9] *tcpdump*,  
<http://www.tcpdump.org/>.
- [10] *Ethereal*,  
<http://www.ethereal.org/>.
- [11] *PSSH Parallel Secure Shell*,  
<http://www.theether.org/pssh/>
- [12] *ntop - a network traffic probe*,  
<http://www.ntop.org/>.
- [13] Larry Wall, Tom Christiansen & Jon Orwant *Programming Perl*, O'Reilly, 3rd Edition July 2000  
<http://www.oreilly.com/>.
- [14] Tobias Oetiker, *The Not So Short Introduction to L<sup>A</sup>T<sub>E</sub>X<sub>2</sub> $\epsilon$* , Version 4.14, 04 April, 2004  
<http://people.ee.ethz.ch/~oetiker/lshort/>.
- [15] Herbert Schuldt *C/C++ GE-PACKT*, MITP-Verlag, 1. Auflage 2001  
<http://www.mitp.de/>, <http://www.ge-packt.de/>
- [16] Martin Schader, Stefan Kuhlins *Programmieren in C++*, Springer-Verlag, 5. Auflage, 1998  
<http://www.springer.de/>.
- [17] Puri Ramneek *Bots&Botnet: An Overview*, SANS Institute, 2003
- [18] *Ferngesteuerte Spam-Armeen*, c't 2004, Heft 5, 2004