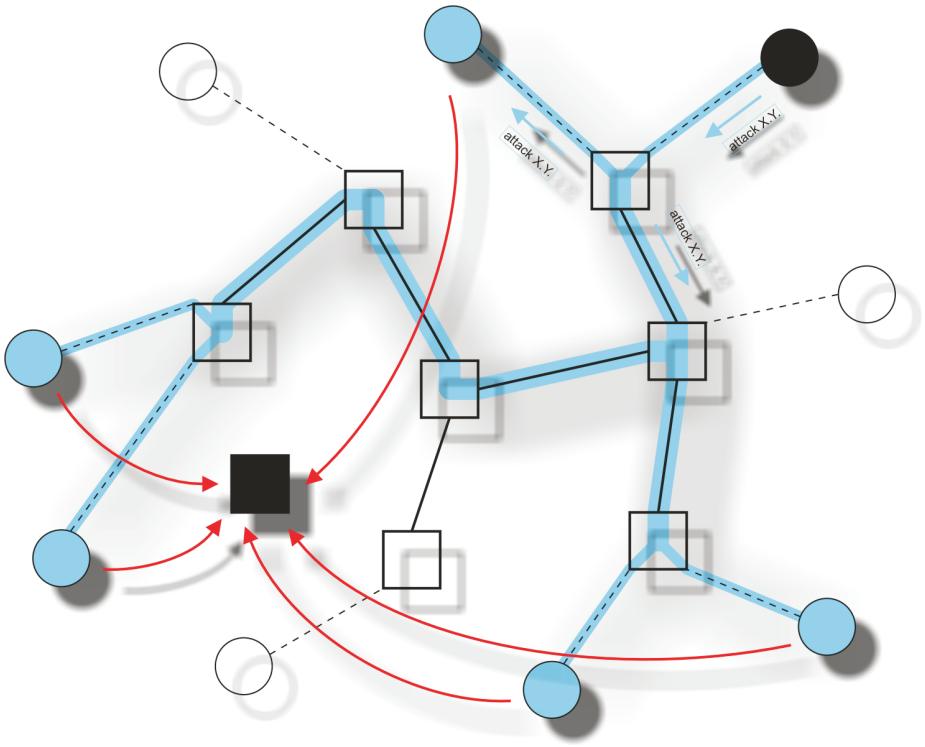
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Stéphane Racine

Analysis of Internet Relay Chat Usage by DDoS Zombies



Master's Thesis MA-2004-01
October 2003 - April 2004

Tutor:
Thomas Dübendorfer

Supervisor:
Prof. Dr. Bernhard Plattner

26.4.2004

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Abstract

This report gives an introduction to "Internet Relay Chat" (IRC), a popular chatting system, discusses the misuse of IRC to launch and control "Distributed Denial-of-Service" (DDoS) attacks and finally proposes methods for detecting the preparation of such IRC-based DDoS attacks. This detection is based on the analysis of Cisco NetFlow data, in other words, recorded network traffic with highly reduced information content.

Die hier vorliegende Dokumentation liefert eine Einführung in "Internet Relay Chat" (IRC), ein weit verbreitetes Chatsystem, bespricht einige der über IRC gesteuerten "Distributed Denial-of-Service"-Attacken (DDoS-Attacken) und schlägt schliesslich Methoden zur Früherkennung der Vorbereitung solcher IRC-basierten DDoS-Attacken vor. Wesentlich dabei ist, dass die Erkennung aufgrund von Cisco NetFlow Daten geschieht, also aufgezeichnetem Netzwerkverkehr mit sehr stark reduziertem Informationsgehalt.

Preface

During my studies I attended a lot of different and interesting courses, but the two ones which attracted my attention most were called "Communication Networks" and "Practical IT Security", both taught at the Computer Engineering and Networks Laboratory (TIK) of the ETH Zurich.

Accordingly to my interests and the various possibilities for student theses I decided to write my thesis at TIK. In the context of the DDoSVax project [1] I focused my work on the "Analysis of Internet Relay Chat Usage by DDoS Zombies".

The following chapter-by-chapter text organisation gives a short overview over this Master's Thesis documentation:

- *Chapter 1* introduces the topic, motivates the work and formulates the Master's Thesis task.
- *Chapter 2* gives an introduction to Internet Relay Chat (IRC).
- *Chapter 3* shortly explains the mode of operation of denial-of-service (DoS) and distributed denial-of-service (DDoS) attacks and illustrates how current IRC-based DDoS attacks work.
- *Chapter 4* shows the way several analyses of IRC traffic were done using Cisco NetFlow data.
- *Chapter 5* discusses the steps conducted to develop an algorithm for the detection of IRC bots.
- *Chapter 6* contains an evaluation of the developed algorithm and results on the investigation of the quality of the NetFlow data.
- *Chapter 7* finally concludes the thesis and states what could be done further.

Acknowledgements

Many thanks to my tutor Thomas Dübendorfer and my co-tutor Arno Wagner for having assisted me during the entire period of my Master's Thesis. Their always opened door allowed a very agreeable working atmosphere.

Furthermore I would like to thank Prof. Dr. Bernhard Plattner for giving me the opportunity to work on a highly interesting subject.

In addition, my thanks go to Pascal Gloor, the administrator of an Undernet IRC server used for this thesis, the people at SWITCH and all other persons, especially the "ETZ G69" student crew (Roman Plessl, Lukas Hämmerle, Samuel Nobs, Simon Steinegger), who helped and encouraged me in many ways.

Last but not least, also special thanks to Leo [2], that gave me a lot of inspiration for writing this thesis in English.

Author

Stéphane Racine
<sracine@ee.ethz.ch>

School

Institut für Technische Informatik und Kommunikationsnetze (TIK)
Departement Informationstechnologie und Elektrotechnik (D-ITET)
Eidgenössische Technische Hochschule Zürich (ETH)

Computer Engineering and Networks Laboratory (TIK)
Department of Information Technology and Electrical Engineering (D-ITET)
Swiss Federal Institute of Technology Zurich (ETH)

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Problem Description

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 where hit. ETH suffered repeated complete loss of Internet connectivity ranging from minutes to hours in duration. Massive distributed DDoS attacks have the potential to cause major disruption of Internet functionality up to severely decreasing backbone availability. [1]

1.1 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.

1.2 The DDoSVax project

In the joint ETH/SWITCH research project *DDoSVax* [1] aggregated Internet traffic data (Cisco NetFlow) is collected at all border gateway routers operated by SWITCH [3]. 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.

1.2.1 Motivation and objectives

The DDoSVax project is motivated by the fact that more and more hosts are connected to the Internet for longer times, often without competent system administration. One of the largest sources of weakly protected hosts are private users and small businesses that use cheap ADSL or television cable based Internet access. While the individual network bandwidth of these hosts is small, control of a larger, well distributed number of these hosts is enough to threaten not only individual servers or networks, but to conduct devastating attacks on the Internet infrastructure itself. Research into countermeasures to these threats is therefore essential.

The DDoSVax project has the following objectives:

- Detection of infection phases while infection takes place.
- Detection and analysis of massive DDoS attacks when they start in near real-time.
- Provision of methods and tools that support countermeasures during both phases.

The hypothesis of the DDoSVax project team is that both attack phases exhibit distinct traffic patterns that allow detection. They will test this hypothesis with measurements of real network traffic and with simulations. [1]

1.3 Master's Thesis task

The task of the student is split in four subtasks:

- IRC analysis,
- IRC-based DDoS attacks,
- NetFlow-based IRC abuse detection,
- and, as a task with less importance, DDoS attack countermeasures.

IRC analysis

The IRC service and protocol must be thoroughly understood and analysed. We recommend to read the IRC protocol specification RFC, to investigate various IRC clients[1], to connect to various IRC servers and to listen as well as to participate in many IRC channel discussions.

A rough estimation on the IRC user community size and a classification should be made to judge the importance of this service.

IRC-based DDoS attacks

Related work about IRC abuse for DDoS attacks will be considered and a survey written. By analyzing the logs and parameters of a specific IRC system, channels that are abused by bots will have to be identified and their behaviour will be observed and various methods for detection of such channels will be proposed.

NetFlow-based IRC abuse detection

The IRC traffic between IRC clients outside the SWITCH network and the IRC server within the SWITCH network, which is investigated in this thesis, crosses one of SWITCH's various border gateway routers. As the DDoSVax project collects NetFlow traffic data at those border gateway routers, such IRC traffic data is available for analysis.

Before any analysis can be done, the student must understand the structure and limitations of NetFlow data as well as the data capturing process used in the DDoSVax project. Already existing tools for data analysis should be considered.

With the results of the previous steps, various algorithms to extract and analyse IRC traffic data and especially such data that belongs to possibly abused IRC channels will have to be developed and thoroughly tested for effectiveness. The result will be one or more implemented and tested "IRC attack detection signature" that can detect IRC-based DDoS attack preparation traffic in NetFlow traffic data.

IRC-based DDoS attack countermeasures

Assuming that we can detect suspicious IRC traffic in near-real-time, countermeasures that could be applied to routers and/or IRC systems will be proposed (on a rather conceptual basis) and their effectiveness evaluated.

[1] mIRC is a very commonly used IRC client

Validation of the results

Thanks to the fact, that we have access to

- an IRC server (being part of a one of the largest IRC networks) located inside AS-559 (SWITCH)

and

- the NetFlow data captured at the border gateway routers of the SWITCH network,

there is the possibility to also validate the proposed solutions and algorithms.

Chapter 2

Internet Relay Chat Survey

This chapter explains current *Internet Relay Chat* (IRC) services. A large part of this text was adapted from the different *Requests for Comments*[1] (RFC's)

- RFC 1459 Internet Relay Chat Protocol [4],
- RFC 2810 Internet Relay Chat: Architecture [5],
- RFC 2811 Internet Relay Chat: Channel Management [6],
- RFC 2812 Internet Relay Chat: Client Protocol [7] and
- RFC 2813 Internet Relay Chat: Server Protocol [8].

The goal of writing this survey was not to create a user's manual, but to give an overview on IRC characteristics that might be of interest for an analysis of Internet Relay Chat usage by DDoS zombies.

There are many user guides on the Internet. A good starting point for new IRC users is *IRChelp.org* [9] on which one can find "The IRC Prelude" and "An IRC Tutorial". Further on there is a German introduction from Kai Seidler [10]. Also the "History of IRC" [11] was a source of information for this survey.

The fact, that we (my tutor and me) had privileged access to an IRC server (geneva.ch.eu.undernet.org) connected to the *Undernet IRC network* [12] explains why this text sometimes focuses on this specific IRC network.

[1] RFC's can be found at the web pages of the *Internet Engineering Task Force* (IETF): `http://www.ietf.org/rfc.html`

2.1 Introduction

As explained on *IRChelp.org* [9], IRC provides a way to use the Internet to communicate in real-time with people from all over the world. In other words: IRC is a multi-user, multi-channel chatting system. This communication takes place on a computer screen in the form of text lines. When "talking" on IRC, everything one types will instantly be transmitted around the world to other users that might be watching their terminals at that time. They can then type something and respond to messages, and vice versa. An IRC system (*IRC network*) consists of *IRC servers*, machines to which users connect. In most cases a user runs a program (*IRC client*) which connects to a server on one of the IRC networks (e.g. the *Undernet IRC network*). The server is responsible for relaying information to and from other servers on the same net.

Once connected to an IRC server on an IRC network, a user will usually join one or more *channels* (comparable to a "chat room" on other chatting systems) and converse with others there. Conversations may be public (where everyone in a channel can see what is written) or private (messages between only two people, who may or may not be on the same channel).

On an IRC network each user is known by its *nickname*.

2.1.1 History of the IRC protocol

IRC was born during summer 1988 when Jarkko Oikarinen wrote the first IRC client and server at the University of Oulu (Finland), where he was working at the Department of Information Processing Science.

Jarkko intended to extend a Bulletin Board System (BBS) software he administrated, to allow Usenet news-kind of discussion, real-time discussions and similar BBS features. The first part he implemented was the chat part, which he did with adapted program parts written by some friends. It was initially tested on a single machine, and according to the words from Jarkko himself "The birthday of IRC was in August 1988" [13]. The first IRC server was named "tolsun.oulu.fi".

Jarkko got some people at the Helsinki and Tampere Universities to start running IRC servers when the number of users increased. Markku Järvinen helped improving the client. At this time Jarkko realized that the rest of the BBS features probably wouldn't fit in his program.

In November 1988 some guys at the University of Denver and Oregon State University had got an IRC network running (they had got the program from one of Jarkko's friends) and wanted to connect to the Finnish network. IRC then grew larger and got used on the entire Finnish national network

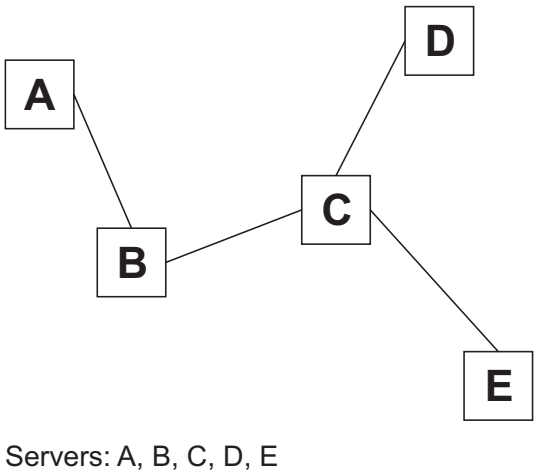


Figure 2.1: Format of a small IRC server network (spanning tree)

(Funet) and then connected to Nordunet, the Scandinavian branch of the Internet. In November 1988, IRC had spread across the Internet.

In the 1989, there were some 40 servers worldwide and *ircII* (an IRC client) was released by Michael Sandrof. In July 1990, IRC averaged at 12 users on 38 servers. [11]

Some more up-to-date statistics will be discussed later in Section 2.6.

2.2 Components and architecture

IRC is based on a client-server model. A user runs a client program on his computer and connects to a server in the Internet. These servers for their part link to many other servers to make up an IRC network, which transports messages from one user (resp. client) to another.

An IRC network is defined by a group of servers connected to each other. A single server forms the simplest IRC network. Figure 2.1 shows a possible format of a small IRC server network. The IRC protocol defined by the RFC's provides no means for two clients to directly communicate (without the need of servers in between)[2]. All communication between clients (e.g. messages) is relayed by the server(s).

[2]Nevertheless there are ways to do so. For further information see Section 2.4.2 on the Client-To-Client-Protocol (CTCP) and the Direct-Client-Connection (DCC).

2.2.1 Servers

The server forms the backbone of IRC, providing a service (typically by default on TCP port 6667) to which clients may connect to in order to talk to each other, and a service for other servers to connect to (also through TCP connections), forming an IRC network.

The server is also responsible for providing the basic services required for real-time conferencing defined by the IRC protocol (client locator, message relaying, channel hosting and management).

Although the IRC protocol defines a fairly distributed model, each server maintains a "global state database" about the whole IRC network. This database is, in theory, identical on all servers.

Servers are uniquely identified by their name which has a maximum length of 63 ASCII characters.

The only network configuration allowed for IRC servers is that of a spanning tree (see Figure 2.1) where each server acts as a node.

2.2.2 Clients

A client[3] is (a computer program on) a host connecting to a server. Each user (of a client) is distinguished from other users by a network-wide unique nickname having a maximum length of 9 ASCII characters. In addition to the nickname of a user, all servers must have the following information about all clients:

- the real name of the host that the client is running on,
- the user name of the client on that host,
- and the server to which the client directly is connected to.

Some clients connected to a small sample IRC network are shown in Figure 2.2.

IRC operators

To manage an IRC network, a special class of users (*IRC operators*) is allowed to perform general maintenance functions on the network. Operators should be able to perform basic network tasks such as disconnecting and reconnecting servers as needed to prevent long-term use of slow links in the spanning tree.

[3]There are two types of clients: user-clients and service-clients. The difference won't be explained in more details here, but can be found in [5, Section 2.2].

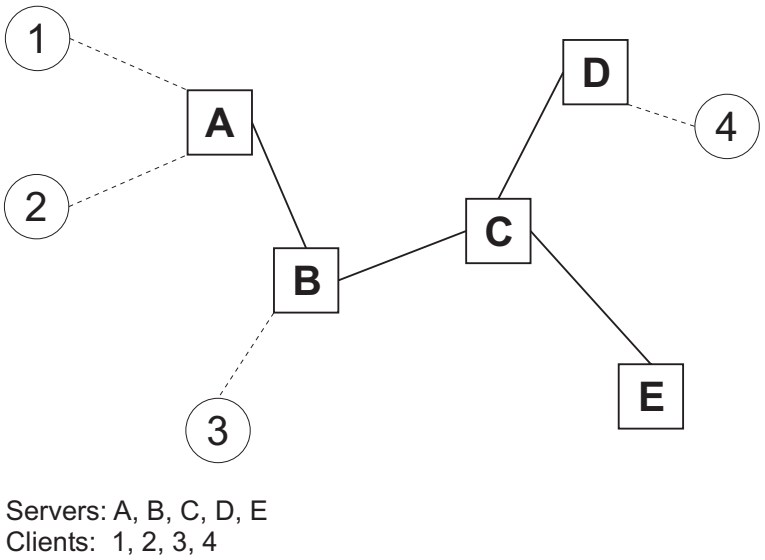


Figure 2.2: Sample small IRC network with clients connected to several servers

IRC operators also have the ability to remove a user from the connected network by "force", i.e. IRC operators are able to close the connection between any client and server.

2.2.3 Channels

A channel is a named group of one or more clients (resp. users) which will all receive messages addressed to that channel. A channel is characterised by its name and current members, it also has a set of properties (*channel modes*) which can be manipulated by (some of) its members. A user can be concurrently connected to more than one channel.

Channels provide a means for a message to be sent to several clients. Servers host channels and provide the necessary message broadcasting to the clients (depending on which channels the user is in). Servers are also responsible for managing channels by keeping track of the channel members.

To create a new channel or become part of an existing channel, a user is required to join the channel. If the channel doesn't exist prior to joining, the channel is created and the creating user becomes a *channel operator* (similar to a moderator). The channel ceases to exist when the last client (resp. user) leaves it. While a channel exists, any client can reference the channel using the name of the channel.

A channel entity is known by one or more servers on the IRC network. A user can only become member of a channel known by the server to which his

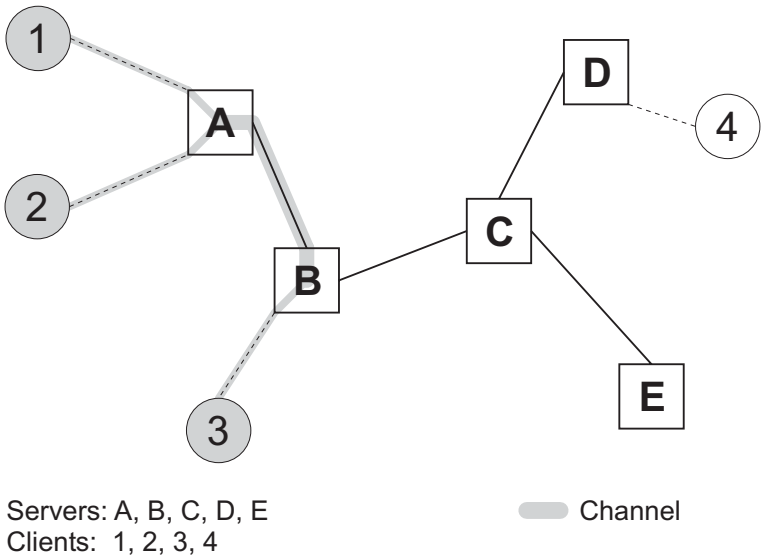


Figure 2.3: Some clients having joined the same channel

client is directly connected. The list of servers which know of the existence of a particular channel must be in a contiguous part of the IRC network, in order for the messages addressed to the channel to be sent to all the channel members. Different clients being members of the same channel are shown in Figure 2.3.

Channel names are strings of length up to 50 ASCII characters with the requirement that the first character (channel prefix) has to be either "#", "&", "+" or "!".

If the IRC network becomes disjoint because of a split[4] (*network split* resp. *net split*) between two servers, the channel on each side is only composed of those clients which are connected to servers on the respective sides of the split, possibly ceasing to exist on one side of the split. When the split is healed, the connecting servers announce to each other who they think is in each channel.

Channel modes

Channel modes define the characteristics and attributes of channels. Channel operators are able to change these modes and can thus permit or forbid certain actions of (normal) users. For instance a channel can be key (password) protected or visible to only some users. More on channel modes can be read in [6, Chapter 4].

[4] A split can occur when a router fails or the network is congested.

Channel operators

The channel operator (which is not the same as an IRC operator) on a given channel is considered to "own" that channel. In recognition of this status, channel operators are endowed with certain powers which enable them to keep control and some sort of sanity in their channel. They have the power to:

- Eject a client from the channel
- Change the channel's mode
- Invite a client to an invite-only channel
- Change the channel topic

A channel operator is identified by the "@" symbol next to his nickname.

2.3 Concepts and communication paths

This section is devoted to describe the concepts behind the organisation of the IRC protocol and the different classes of *messages*. Figure 2.2 shows a small sample IRC network.

There are three different types of communication:

- one-to-one communication
- one-to-many communication (e.g. to a channel, which corresponds to a multi-cast message)
- one-to-all communication (e.g. a broadcast message to all clients or servers or both)

Communication on a one-to-one basis is usually performed by clients. To provide a means for clients to talk to each other, it is required that all servers are able to send a message along the spanning tree in order to reach any client. Thus the path of a message being delivered is the shortest path between any two points of the spanning tree. The following example refers to Figure 2.2: A private message between clients 1 and 3 is seen by servers A and B, and clients 1 and 3. No other clients or servers are allowed to see the message.

The main goal of IRC is to provide a platform which allows easy and efficient conferencing (one-to-many conversations). In IRC the channels have a role equivalent to that of the multi-cast group; their existence is dynamic and the actual conversation carried out on a channel must only be sent to

servers which are supporting users on a given channel. Moreover, the message shall only be sent once to every local link as each server is responsible to fan the original message to ensure that it will reach all the recipients. The following example refers to Figure 2.3: Clients 1, 2 and 3 are members of the same channel. All messages to this channel are sent to only those clients and servers which must be traversed by the messages as if they were private messages to a single client. If client 1 sends a message, it goes via server A to client 2 and via server A and B to client 3.

The one-to-all type of message is better described as a broadcast message, sent to all clients or servers or both. On a large network of users and servers, a single message can result in a lot of traffic being sent over the network in an effort to reach all of the desired destinations. For some sort of messages, there is no option but to broadcast it to all servers so that the state information held by each server is consistent between servers. There is no class of message which, from a single client-message, results in a message being sent to every other client in the IRC network. Most of the commands which result in a change of state information (such as channel membership, channel mode, user status, etc.) must be sent to all servers by default. While most messages between servers are distributed to all other servers, this is only required for any message that affects a user, channel or server. Since these are the basic items found in IRC, nearly all messages originating from a server are broadcast to all other connected servers.

2.4 Protocol

IRC has been implemented as a text-based (8-bit characters) protocol on top of TCP (usually using ports in the range of 6000 to 7000[5], whereas the non-registered port 6667 is often used as default) since TCP supplies a reliable network protocol.

Even if the IRC protocol has been developed on systems using the TCP/IP network protocol, there is no requirement that this remains the only protocol suite in which it operates.

2.4.1 Messages

Servers and clients send each other IRC messages which may or may not generate a reply. If the message contains a valid command the client or server

[5]The server to which we had access to (geneva.ch.eu.undernet.org) has enabled the following ports for IRC clients: 6660-6669, 7000, 7777, 8000. Undernet servers communicate on port 4400 with each other.

should expect a reply but it is not advised to wait forever for the reply; client-to-server and server-to-server communication is essentially asynchronous in nature.

Each IRC message may consist of up to three main parts:

- the prefix (optional),
- the command,
- and the command parameters.

The prefix, command, and all parameters are separated by one ASCII space character each.

The presence of a prefix is indicated with a leading colon character (":"), which, if present, must be the first character of the message itself. There must be no gap (whitespace) between the colon and the prefix. The prefix is used by servers to indicate the true origin of the message. If the prefix is missing from the message, it is assumed to have originated from the connection it was received from. As an example two messages from Appendix A were taken:

- `:ddv_argan!ddosvax@pc-abcd.ethz.ch JOIN #ddv_ddvax`
- `PRIVMSG #ddv_ddvax :Hello everybody!`

The first message contains a prefix indicating, that a user "ddosvax" on "pc-abcd.ethz.ch" having "ddv_argan" as his nickname has joined channel "#ddv_ddvax". Such a message would be sent through the IRC network to all users already being part of this channel. The second message without a prefix would be seen between a client and the server it is directly connected to, meaning that the user of the client sends the text (indicated by the command "PRIVMSG") "Hello everybody!" to channel "#ddv_ddvax".

IRC messages are always lines of characters terminated with a CR-LF (Carriage Return - Line Feed) pair, and these messages must not exceed 512 characters in length, counting all characters including the trailing CR-LF. Thus, there are 510 characters maximum allowed for the prefix, the command and its parameters.

Numeric replies

Most of the messages sent to the server generate a reply of some sort. The most common reply is the numeric reply, used for both errors and normal replies.

2.4.2 Client-specific protocols

As already mentioned in Section 2.2, the IRC protocol specified by the RFC's has no possibility for two clients to communicate directly. This drawback has been eliminated with the implementations of the *Client-To-Client-Protocol (CTCP)* and the *Direct-Client-Connection (DCC)*. In both cases the servers in the IRC network have nothing to do with these extended features. The whole protocols are sent and interpreted by the clients. The specification and more detailed information can be found in [14], [15] and [16].

Client-To-Client-Protocol

If one would use the layered view of network protocols to explain CTCP, then CTCP is best seen as being just over the "real" IRC protocol. Every CTCP message/command is "packed" into a normal IRC message on the client side, sent over the IRC network to the desired client (resp. user) like a private message and "unpacked" and interpreted by the receiving client.

The Client-To-Client-Protocol is meant to be used as a way to

- in general send structured data between users clients,

and in a more specific case:

- place a query to a user's client and getting an answer.

Direct-Client-Connection

DCC uses direct TCP connections[6] between the clients taking part to carry data. There is no flood control (on application level), so packets can be sent at full speed, and there is no dependence on server links (or load imposed on them). In addition, since only the initial handshake for DCC connections is carried by CTCP messages through the IRC network, two clients using DCC have a somewhat more secure chat connection while still in an IRC-oriented protocol.

CTCP DCC extended data messages are used to negotiate file transfers between clients and to negotiate chat connections over TCP connections between two clients, with no IRC server involved.

[6] As an example the *mIRC* client sets the port to a random number between 1024 and 5000.

2.5 IRC software

2.5.1 IRC clients

ircII

In the early days of IRC, the *ircII* [17] program was the premiere client. Up to now it is still enhanced with new features. A lot of other clients are based on this one. The *ircII* client is designed to run in text-mode. If, for example, you type

```
ircII ddv_argan geneva.ch.eu.undernet.org
```

in a terminal of a UNIX environment with *ircII* installed, this will connect you as user "ddv_argan" to the default IRC port (6667) of the IRC server "geneva.ch.eu.undernet.org". Once you are connected you can type something like

```
/JOIN #ddv_ddvax
```

which will make user "ddv_argan" enter the channel "#ddv_ddvax".

To leave the channel type

```
/PART
```

and then type

```
/QUIT
```

to disconnect from the server.

mIRC

mIRC [18] is the most popular and probably most powerful IRC client for Windows (shareware).

ChatZilla

IRC client being part of the Mozilla [19] web browser.

2.5.2 IRC servers

Undernet IRC server

The *Undernet IRC server (ircu)* [20] uses an adapted version of the server protocol described in [8]. Servers connected to Undernet communicate through the *Undernet P10 Protocol* [21].

The P10 protocol uses a scheme of "numerics" to uniquely identify a client or server within the network. Each server has its own unique number (0 to 4095) and each client has its own number within that server (0 to 262,143).

The numbers are encoded into a Base64 stream to maintain human readable data flow and reduce the size of the messages. Also the possible commands contained in a message of the server-to-server-protocol are different. As an example the command "PRIVMSG" becomes "P". In this context "P" is called a *token*.

The aim of tokenisation is to reduce the bandwidth used during network communication by reducing the length of common message identifiers.

2.6 Statistics of IRC networks

On the 12th of November 2003 the IRC search engine of *SearchIRC* [22] monitored

- 1,265 IRC networks with a total of
- 1,026,097 people in
- 628,346 channels,

while the pages of *netsplit.de* [23] indexed

- 662 networks, with
- 1,082,747 people,
- 627,267 channels and
- 5,316 servers.

These numbers from two different sources, which in some points diverge a lot, show the difficulty to give an exact statement for statistics of IRC usage. Exact numbers of the worldwide amount of IRC networks can't be given.

Since these two mentioned sources don't publish how they exactly do the measurements and the counting, it is comprehensible that the respective numbers differ from each others.

	servers	channels	users
1. QuakeNet	n/a	138,754	155,443
2. EFnet	n/a	38,354	126,558
3. Undernet	n/a	35,345	112,622
4. IRCnet	n/a	46,331	101,847

Table 2.1: Statistics for the four largest IRC networks by *SearchIRC* [22] (average over the last week), November 2003

	servers	channels	users
1. QuakeNet	44	173,851	157,123
2. EFnet	49	46,028	128,032
3. Undernet	35	47,165	113,610
4. IRCnet	44	53,827	105,600

Table 2.2: Statistics for the four largest IRC networks by *netsplit.de* [23] (average over the last day), November 2003

Tables 2.1 and 2.2 show the statistics for the four largest IRC networks.

An IRC network does not always consist of the same number of servers. This is illustrated by Figure 2.4.

Figure 2.5 shows that on certain networks the user community is still growing.

When comparing Figure 2.6 to Figure 2.5 one can see that the number of channels is almost proportional to the number of users on a specific network.

2.7 Problems of the IRC protocol

This section will introduce the most important problems of the IRC protocol.

2.7.1 Problems due to the architecture of the protocol

Scalability

It is widely recognised that this protocol does not scale sufficiently well when used in large IRC networks. The main problem comes from the requirement that all servers know about all other servers, clients and channels and this information has to be updated as soon as it changes.

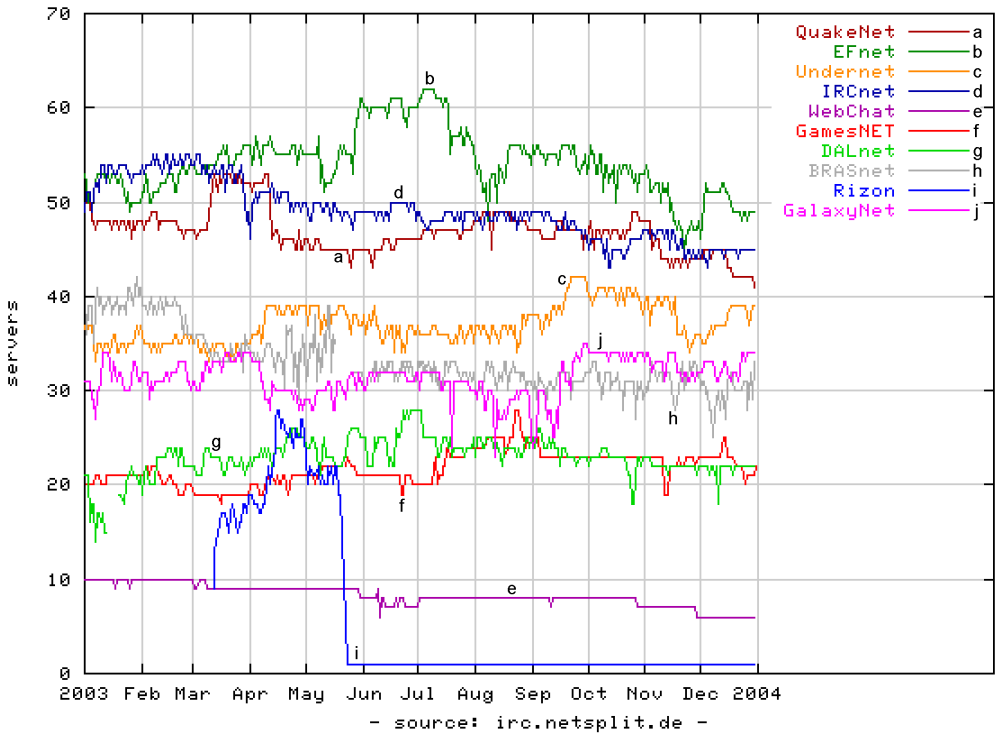


Figure 2.4: Top 10 IRC networks 2003 – Server statistics by *netsplit.de* [23]

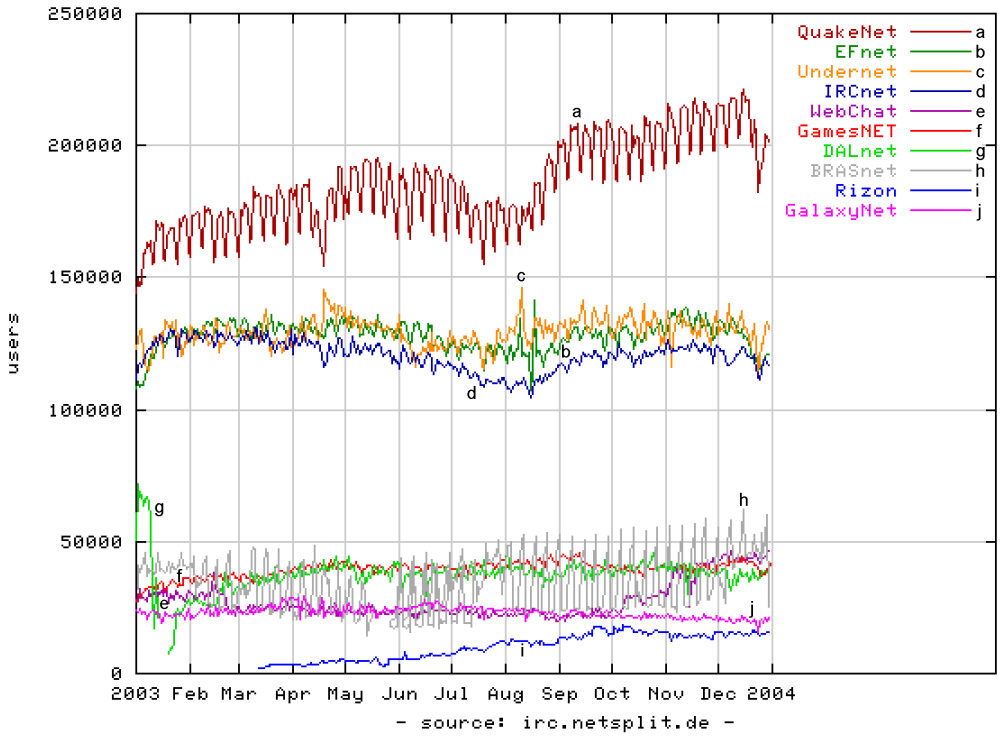


Figure 2.5: Top 10 IRC networks 2003 – User statistics by *netsplit.de* [23]

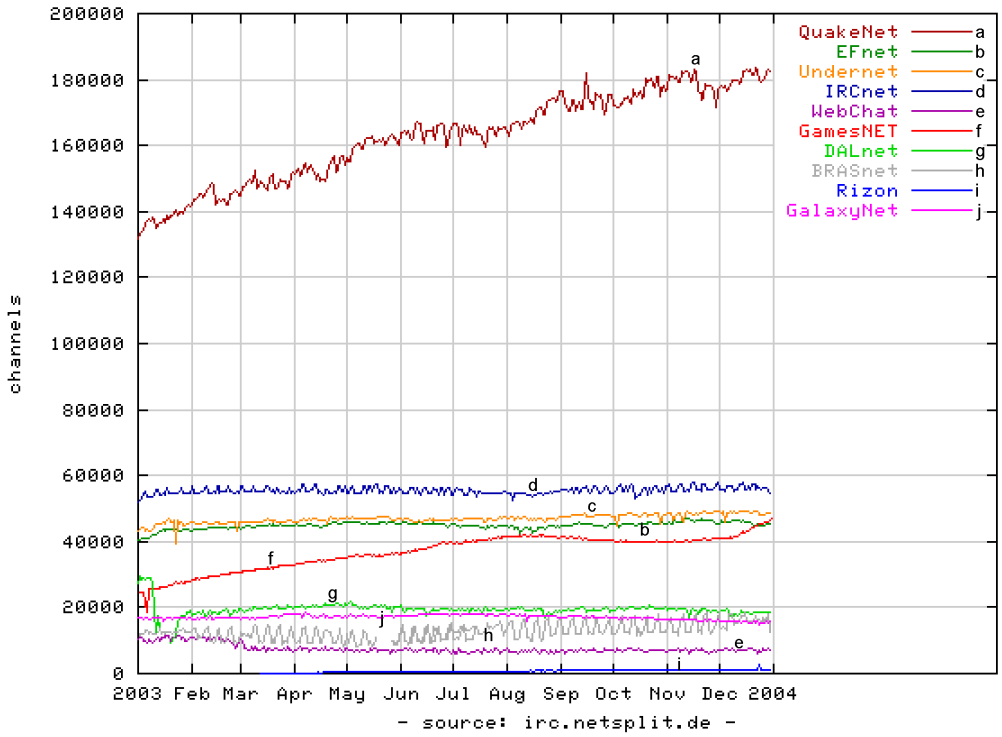


Figure 2.6: Top 10 IRC networks 2003 – Channel statistics by *netsplit.de* [23]

Reliability

As the only network configuration allowed for IRC servers is that of a spanning tree, each link between two servers is an obvious and quite serious single point of failure. This particular issue is addressed in more detail in [8].

Network congestion

Another problem related to the scalability and reliability issues, as well as the spanning tree architecture, is that the protocol and architecture for IRC are extremely vulnerable to network congestions. If congestion and high traffic volume cause a link between two servers to fail, not only this failure generates more network traffic, but the reconnection (eventually to another node in the spanning tree) of two servers also generates more traffic.

Privacy and anonymity

Besides not scaling well, the fact that servers need to know most information (e.g. nickname, host name) about other entities, the issue of privacy of IRC users is also a concern. This is in particular true for channels, as the user-related information on channels is revealing a lot more than whether a user is online or not.

2.7.2 Security considerations

Access control

One of the main ways to control access to a channel is to use checks which are based on the user name and host name of the user connections. This mechanism can only be efficient and safe if the IRC servers have an accurate way of authenticating user connections, and if users cannot easily get around it. While it is in theory possible to implement such a strict authentication mechanism, most IRC networks (especially public networks) do not have anything like this in place and provide little guarantee about the authenticity of the user name and host name for a particular client connection.

Another way to control access is to use a *channel key* (password), but since this key is sent in plain-text, it is vulnerable to traditional packet sniffing attacks.

Channel privacy

Because channel collisions (which can happen when two servers (re-)connect to each other) are treated as inclusive events[7] (see [6, Section 6.3]), it is possible for users to join a channel overriding its access control settings. This method has long been used by individuals to "take over" channels by "illegitimately" gaining channel operator status on the channel. The same method can be used to find out the exact list of members of a channel, as well as to eventually receive some of the messages sent to the channel.

Authentication

Servers only have two means of authenticating incoming connections: plain text password, and DNS lookups. While these methods are weak and widely recognised as unsafe, their combination has proven to be sufficient in the past.

The same comments apply to the authentication of IRC operators. It should also be noted that while there has been no real demand over the years for stronger authentication, and no real effort to provide better means to authenticate users, the current protocol offers enough to be able to easily plug-in external authentication methods based on the information that a client can submit to the server upon connection: nickname, user name, password.

Integrity

Since the messages of the IRC protocol are sent in clear text, a stream layer encryption mechanism (like "The TLS Protocol" [24]) could be used to protect password transmissions.

[7] This means, that the resulting channel has for members all the users who are members on either server prior to the (re-)connection.

Chapter 3

IRC-based DDoS Attack Survey

This chapter explains current IRC-based distributed denial-of-service (distributed DoS or DDoS) attack methods, but will first give a short introduction to the problem of denial-of-service (DoS) attacks in general.

Like in Chapter 2 this survey will give a summary of related work and literature covering this topic. Most, if not all, of the references were found in the Internet.

If you are not very familiar with what this chapter is all about and like to read "good" (depending on your own opinion) thrillers, I suggest to at least have a look at [25].

3.1 Introduction

The traditional purpose and impact of DoS or DDoS attacks is to prevent or deteriorate the legitimate use of computer or network resources. These attacks illegitimately consume the resources of hosts or networks.

As already mentioned in Chapter 1, distributed DoS attacks are a threat to Internet services ever since the widely published attacks on Yahoo, Ebay and Amazon in February 2000. Massively distributed DoS attacks have the potential to cause major disruption of Internet functionality up to severely decreasing backbone availability. They are a significant problem because they can shut an organization off the Internet and because there is no comprehensive solution for protecting a site from a denial-of-service attack.

In the years 2000 and 2001 there has been seen an increase in the use of IRC protocols and networks as the communications backbone for DDoS networks ([26]). The use of IRC essentially replaces the function of the *handlers*

in older DDoS network models (compare to Section 3.2 and Figure 3.1). IRC-based DDoS networks are sometimes referred to as *botnets*, referring to the concept of *bots* (robots) on IRC networks being software-driven participants rather than human participants. [26]

3.2 DoS and DDoS attacks

A denial-of-service attack's primary goal is to deny a victim (host, router or entire network) providing or receiving normal services in the Internet. It is an explicit attempt by attackers to prevent users or providers of a computer-related service from using, respectively providing, that particular resource.

Today, the most common DoS attack type reported involves sending a large number of packets to a destination which causes the endpoint (and possibly transit) network bandwidth to be used up.

There are two principal classes of attacks: *logic attacks* and *(packet) flooding attacks*. Attacks in the first class, exploit existing software vulnerabilities to cause remote servers to crash or substantially degrade in performance. The second class, *flooding-based DoS attacks*, floods the victim's CPU (e.g. by imposing computationally intensive tasks on a victim, such as encryption and decryption computation), memory or network resources by sending large numbers of faked requests or packets. Because there is typically no simple way to distinguish the "good" packets from the "bad" ones, it can be extremely difficult to defend against flooding attacks.

Early DoS attack technology involved simple tools that generated and sent packets from a single source aimed at a single destination. Over time the model for denial-of-service attacks has evolved from

- "single attacker machine against single target machine" (DoS) to
- "multiple attacker machines flooding requests to single (or multiple) target machine(s)" (DDoS)

The DDoS model was improved by attackers by using multiple *handlers* (see Figure 3.1) for directing and managing a large number of hosts against a single target.

DDoS attacks do not rely on particular network protocols or system weaknesses. Instead, they simply exploit the huge resource asymmetry between the many attacking hosts and the victim in that a sufficient number of hosts is amassed to send useless packets toward a victim around the same time. Typically an attacker compromises a set of Internet hosts (using manual or automated methods) and installs a small attack daemon on each, producing

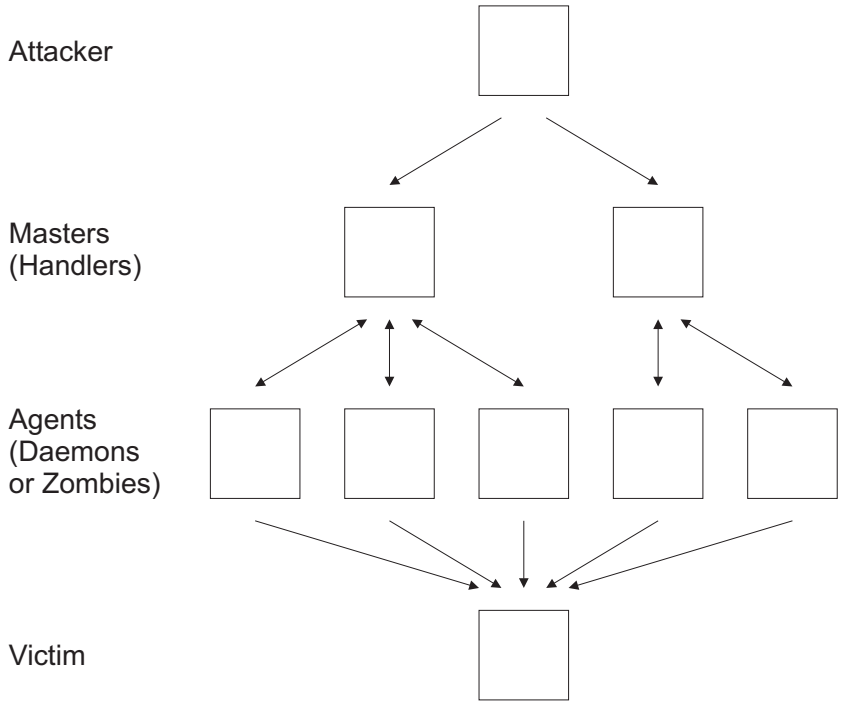


Figure 3.1: Direct DDoS attack architecture

a group of *zombie hosts*. This daemon typically contains both the code for sourcing a variety of attacks and some basic communications infrastructure to allow remote control of the *zombies*. This allows an attacker to focus a coordinated attack from thousands of *zombies* onto a single site. The magnitude of the combined traffic is significant enough to congest, or even crash, the victim's system (system resource exhaustion), or its Internet connection (network bandwidth exhaustion), or both, therefore effectively taking the victim off the Internet. The widely publicized DDoS attacks against popular Web sites in the year 2000 revealed the vulnerability of even very well equipped networks.

Before launching a *direct DDoS attack* (Figure 3.1), an attacker first sets up a DDoS attack network, consisting of one or more attacking hosts, a number of *masters* or *handlers*, and a large number of *agents* (also referred to as *daemons* or *zombies*). The attacking host is a compromised machine used by the actual attacker to scan for vulnerable hosts and to implant specific DDoS master and agent programs[1]. Each attacking host controls one or more *masters*, and each *master* in turn is connected to a group of *agents*. With an attack network ready, the attacker may launch a DDoS attack by issuing an

[1]e.g. Trinoo, Tribe Flood Network 2000, Stacheldraht, etc.

attack command with the victim's address, attack duration, attack methods and other instructions to the masters. This communication is often based on TCP and the messages are sometimes even encrypted. Each *master*, when having received the instructions, passes them to its *agents* for execution. Today's DDoS attack tools can launch attacks against multiple victims at the same time and use various types of attack packets[2].

To hide their location, attackers can forge, or "spoof", the IP source address of each packet that a zombie sends. Consequently, the packets appear to the victim to be arriving from one or more third parties. Spoofing can also be used to "reflect" an attack through an innocent third party (see Figure 3.2). In such *reflector DDoS attacks* the agents send packets that require responses (e.g. ICMP echo requests) to the *reflectors* with the packet's source address set to the victim's address. Without realizing that the source address was spoofed, the *reflectors* send the response packets to the victim.

3.3 IRC-based DDoS attacks

Interestingly, the use of *handlers* (see Figures 3.1 and 3.2) to manage and direct large number of *zombie hosts* (infected systems under attacker control) has in recent years largely been replaced by IRC networks, acting as the attacker's virtual command and control centers. Such an IRC-based DDoS attack architecture is shown in Figure 3.3. The term *bots* is used in analogy to the term *agents* which, in traditional DDoS models (compare to Section 3.2), infect host machines and maintain access for attackers to control them via *handlers*. Analogous one refers to *IRC botnets* when talking about the control infrastructure.

The use of IRC makes it quite difficult to identify DDoS networks. IRC networks and protocols allow DDoS agents being placed on compromised systems to establish out-bound connections to a standard service port (e.g. 6667) used by a legitimate network service (e.g. IRC). Agent communication to the control point may not be easily discernible from other legitimate network traffic. Also, the *agents* do not incorporate a listening port that is easily detectable with network scanners. An *attacker* or *master* can establish a connection to an IRC server using legitimate communication channels to control the DDoS *agents*. Security policies that control out-bound access to standard IRC-related ports (e.g. 6660-6669) may be able to detect and prevent unauthorized connections, but the popularity of IRC services means that such access controls are not widely implemented in security policies.

[2]Attack packet types can be TCP, ICMP, UDP, or a mixture of them. [27]

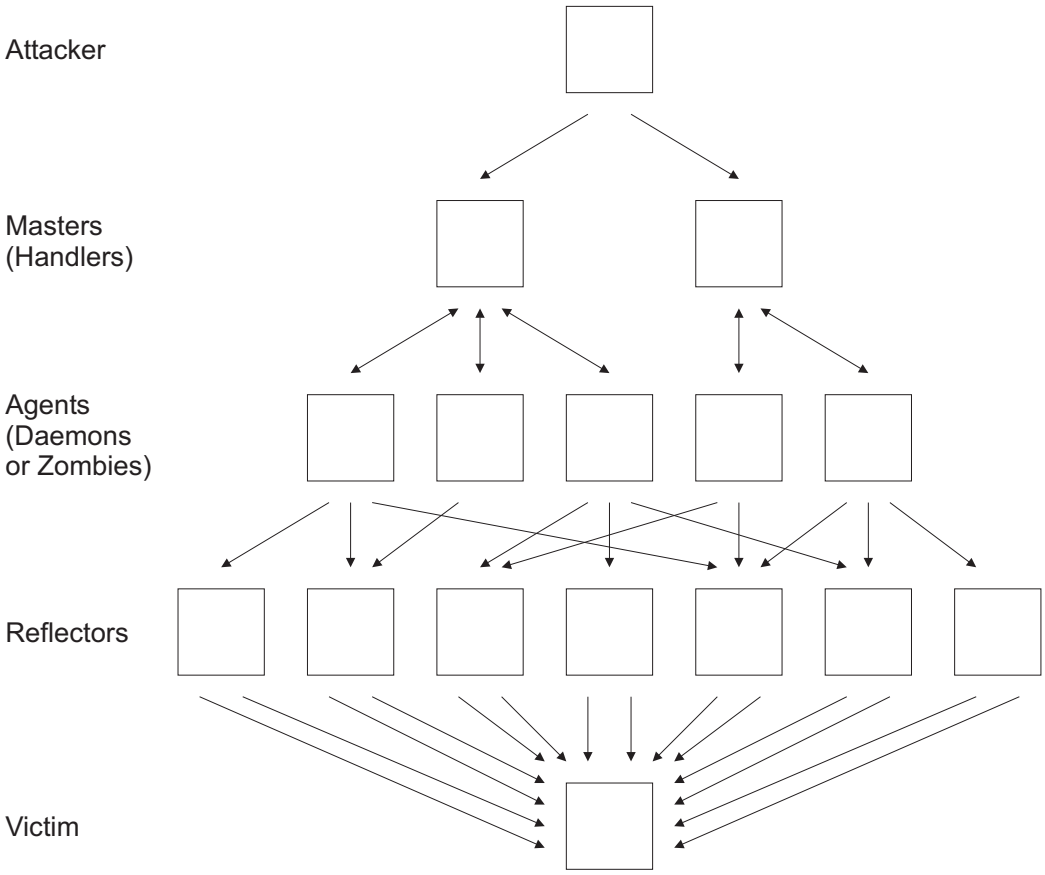


Figure 3.2: Reflector DDoS attack architecture

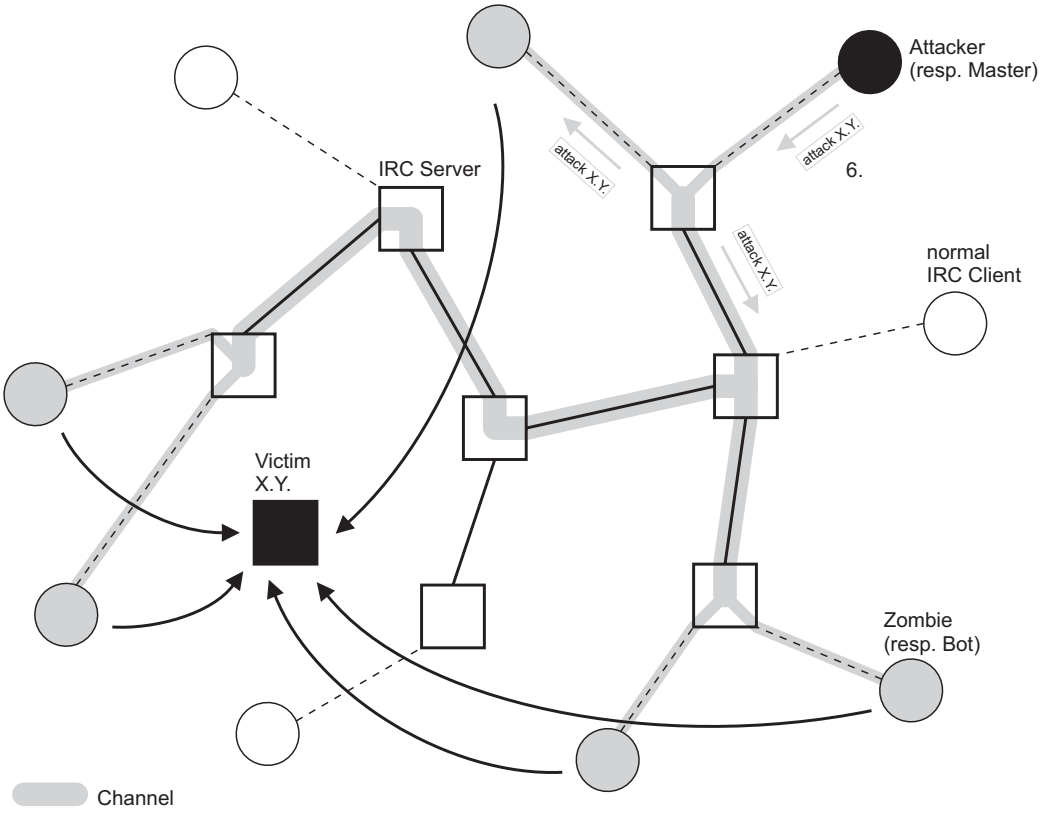


Figure 3.3: IRC-based DDoS attack architecture

IRC networks and protocols also offer good survivability of DDoS networks. The IRC server tracks the addresses for connected *agents* and allows communication between the *master* and the *agents*. The need for custom protocols and tracking of agents is therefore eliminated. Thus, the discovery of a single *agent* may lead to the identification of one or more IRC servers and channel names used by the DDoS network. From there, identification of the DDoS network depends on the ability to track *agents* currently connected to the IRC server.

For public IRC networks such as Undernet removing an IRC server to disable a DDoS network is not a realistic option. Thus, the use of public IRC networks has the advantage of providing a stable communication infrastructure for DDoS networks. On the other hand, public IRC networks expose DDoS networks and agent locations to external identification by security teams who are able to respond in some capacity. This is the reason why intruders are also using private IRC servers as the communication backbone for DDoS networks.

Some IRC-based DDoS *agents* (*bots*) also include the capability for an *attacker* to move the agent connection point by issuing a command to the *agents*. In other words, remote reconfiguration is being built into DDoS *agents* to make it possible for intruders to manage and control their DDoS networks. Regardless of that ability, it is trivial for intruders to alter the connection point in agent code and quickly redeploy DDoS *agents* that connect to a different IRC control point. As a consequence, the discovery of (expendable) *agents* being part of an IRC-based DDoS network (*botnet*) does not compromise and has only little impact to the effectiveness of the DDoS network. [26]

3.3.1 IRC bots and botnets

IRC bots (short for robots) are special programs or IRC scripts designed to perform predefined functions in an automated fashion. IRC scripts are programs used by IRC clients such as *mIRC* to extend their set of features in ways that either provide new functions for channel/user management, or provide malicious features to disrupt other user's IRC sessions. The original intent of *bots* was to enable a twenty-four-hour presence and a remote method for maintaining control of IRC channels. Its prevalent purpose is to remain on the IRC channel at all times and provide services (e.g. file sharing) to members of that channel. Also, *bots* may provide services to the clients of a certain server or to the users of an entire network.

Bots in their malicious mutation are used by *attackers* to infect victim machines after they have been compromised or the victim machine's user

is tricked into performing the installation. Typically, a *bot* establishes outbound connections to a standard IRC network service port, joins the (configured) attacker's private IRC channel and waits for the attacker's commands. Public IRC networks such as Undernet provide a stable, scalable and free of charge communication infrastructure which can be misused by attackers to maintain, expand, manage and control their bots army.

A malicious *bot* is typically an executable file, capable of performing a set of functions, each of which could be triggered by a specific command. A *bot* when installed on a victim's machine changes the system configuration to start each time the system boots.

The typical size of a compressed *bot* is less than 15 Kilobytes in size. A "standard" *bot* generally used by less sophisticated attackers can be downloaded from warez sites on the Internet and edited to include the desired remote IRC server to connect to, the remote TCP port to use for the connection, the channel to join on that server and the authentication password (resp. "key") to gain access to the attacker's private channel. A more sophisticated attacker can even manipulate the bot characteristics like the files created after installation and the install directory where the bot files reside after installation.

One important point to note is that *bots* are not the exploits of an operating system or an application, they are the payload carried by worms, or means used to install a backdoor once a machine has been compromised.

Till this day there have been reported botnets with the impressive dimension of 11,000 bots [28] and up to 25,000 bots [29].

In summary, *IRC bots* (resp. *zombies*) are automated and controlled by events which could be commands given in an IRC channel by another *IRC bot* or a client with necessary privileges. The *attacker* or *master* is the one that installs, configures, controls and directs the bots once they joined the predefined IRC channel. Finally the "control center" or "control channel" used is usually a private IRC channel created by the *attacker* as meeting point for the bots to be joined once they are installed on infected machines and are online. All the bots once connected to control the channel form a *botnet* (i.e. network of bots), awaiting the attacker's commands. [30]

3.3.2 Host infection and bot control process

The process of host infection and the following bot control process are explained in the enumerated list below. The numbers refer to the Figures 3.3 and 3.4.

1. The attacker attempts to infect the victim machines with bots through

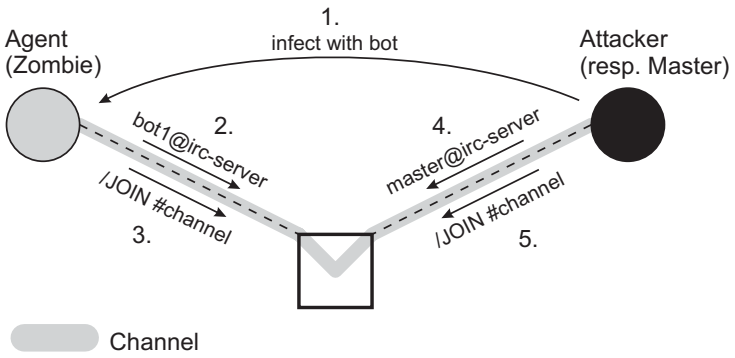


Figure 3.4: Host infection process

either exploiting some operating system or application vulnerabilities or trick the user into executing a malicious program leading to a bot installation. A typical way for attackers to infect large groups of hosts is to use exploit code of recently published vulnerabilities, use them to gain access to the victim's machines and install bots as backdoor to maintain an access. The described process could be automated by using a directed worm that will scan a target subnet for known vulnerabilities, exploit the largely unpatched systems and infect them with a malicious bot. Another way is to exploit unpatched web applications and trick the user into executing some malicious program or virus leading to bot infection. A user could install an IRC client with a Trojan inside which, while doing all legitimate tasks of an IRC client, also installs a bot on the user's machine.

After the bot has been installed on the victim's machine it copies itself to the install directory and usually updates the registry keys in case of windows platform.

Once infected these hosts are also referred to as *zombies*.

2. In the next stage, the bot attempts to connect to an IRC server with a randomly generated nickname, i.e. the unique name representing that bot in the attacker's private channel.[3]

3. Once the bot has connected to the IRC server, it joins, with a predefined "key" (authentication password), the attacker's channel as part of the attacker's botnet army and waits for instructions.

[3]Frequently, the attackers use public IRC servers for these activities and could be banned by IRC administrators, thus loosing their botnet army. To avoid this, attacker sometimes use service providers like dyndns.org or no-ip.com to dynamically map their bots with multiple IRC servers.

4. From time to time also the attacker will connect to the IRC server and

5. join the channel using IRC to log into the zombies with a possibly complex and sometimes encrypted access password, ensuring that the bots cannot be controlled by others and making it harder for someone to hijack the botnet.

6. After the access has been accepted the attacker may direct and remotely control the action of a large number of infected zombies via the botnet, launch attacks or use it for other malicious activities. When there are enough bots listening and waiting, the master sends an attack command to the channel which the bots will execute immediately (see Figure 3.3).

3.3.3 Some known DDoS bots

The company *Simovits Consulting*[4] maintains a huge list [31] with the descriptions of Trojans and bots. In the next two sections you will find detailed descriptions of some Windows- and Unix-based DDoS bots.

Evilbot, Slackbot (Windows)

There are several IRC DDoS bots targeted to run on Windows hosts. Two, rather similar ones, of them (Evilbot and Slackbot) will be discussed in this section.

Evilbot is one of those publicly available bots (Slackbot 1.0 is the other one), which happens to be the bot that was used to flood "grc.com" (see [25]). A closer look at this 16 Kilobytes long executable will show the following behavior: When a victim executes the bot file it copies itself to a Windows directory with a specified name (e.g. \Windows\WinRun2.exe). As the "WinRun2.exe" file must be run every time the computer is booted, it adds itself to registry's autostart as follows:

> `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\`
> `CurrentVersion\Run\<a specified reg key here>`
> (e.g. `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\`
> `CurrentVersion\Run\WinRun`)

That WinRun key points to "WinRun2.exe" which was copied to the Windows directory. If there is already a "WinRun2.exe" named file on Windows directory the Evilbot won't overwrite that file, instead it will add a similar

[4] `http://www.simovits.com/`

registry key as above but point it to load the file from the current directory where the bot is saved and where it was run.

Next the bot tries to connect to a specified IRC server. When it has joined the specified channel it sits and waits for commands. At the same time the bot connects to a Web or FTP server where it downloads a program file (e.g. a Trojan server or whatever there is behind the URL "update.ur.address./thepath.exe") and then executes it. (Note: The dots surrounding "ur" are dots but the dot sign between "address" and "/" stands for a null). If the Evilbot is the first who joins a non-registered channel it sets by default the following channel modes: `+nstk`.[5]

Evilbot accepts commands from anyone who is on the same channel with it (whereas Slackbot 1.0 requires a password before listening to commands from anyone). The following command on the same IRC channel where Evilbot is, would cause a UDP packet flood attack against a certain host:

```
!udp A.B.C.D 1000 0
```

1000 stands for the amount of packets that will be sent and 0 at the end of the line stands for the delay between each packet. Evilbot can attack by "pinging" a target host too. It supports four different kind of ping attacks:

- `!p4 <victim ip>`
  Sends 10,000 64 Kilobytes ping packets to specified IP

- `!p3 <victim ip>`
  Sends 1,000 64 Kilobytes ping packets to specified IP

- `!p2 <victim ip>`
  Sends 100 64 Kilobytes ping packets to specified IP

- `!p1 <victim ip>`
  Sends 10 64 Kilobytes ping packets to specified IP

The amount of Pings and the ping size can be configured (e.g. the `!p4` command could be set to send 15000 32 Bytes ICMP packets to a specified host, but by default it uses the above values).

Evilbot also understands other commands such as part/join a channel. [32]

[5] As specified in [6]: n: toggle the no messages to channel from clients on the outside; s: toggle the secret channel flag; t: toggle the topic settable by channel operator only flag; k: set/remove the channel key (password);

Trinity (Unix/Linux)

Trinity is a Linux-based DDoS attack tool used to launch coordinated denial of service attacks against one or more target systems simultaneously. As reported by [26] it was first seen in August 2000. Much the same as its predecessors[6], the Trinity tool consists of master and daemons. However, Trinity is a much more convenient tool for the attack since the daemons can be remotely controlled through a standard Internet Relay Chat, AOL's chat or ICQ. Another feature which makes Trinity difficult to detect is the fact that the daemons do not listen to specific ports to receive commands from the master, but receive them over an IRC or ICQ channel. There are at least eight variations of Trinity discovered on the Undernet Internet Relay Chat network (which requires a special password to access).

To launch a DDoS attack, the daemons must first be secretly installed on a number of compromised Linux machines. Then, each compromised machine or daemon will join a specific IRC channel and waits for attack instructions from the master. The Trinity master, under the control of an attacker, then orders the daemons to generate a specified type of denial-of-service attack against one or more target servers. Trinity is capable of generating a variety of DDoS attacks[7]. The flooding commands have this format:

```
<flood> <password> <victim> <time>,
```

where `<flood>` is the type of flood, `<password>` is the daemon's password, `<victim>` is the victim's IP address, and `<time>` is the length of time to flood (in seconds).

In the example from the alert issued by Internet Security Systems (ISS)[8], the daemon binary is installed at "/usr/lib/idle.so". When "idle.so" is started, it connects to an Undernet IRC server on port 6667 (TCP). The daemon binary also contains a list of servers.

When a Trinity daemon enters the channel "#b3eblebr0x", it sets its nickname as the first six characters of its host name plus three random letters or numbers (e.g. compromised.machine.com → comproxyz). The daemons always respond to commands on lines beginning with "(trinity)", hence ISS named this DDoS attack tool as Trinity.

With the earlier distributed denial-of-service attack tools, the attackers had to keep a list of all the machines they broke into, while with Trinity, all compromised machines simply show up in the specified channel. The chat

[6] e.g. Trinoo, Tribe Flood Network 2000, Stacheldraht

[7] e.g. UDP Flood, TCP SYN Flood, TCP ACK Flood, TCP RST Flood, TCP NULL Flood, TCP Fragment Flood, TCP Random Flood attacks

[8] `http://www.iss.net/`

feature in the Trinity tool also makes it easier for an attacker to launch the attack and prevents the attacker's real identity from being discovered, since attackers usually change their IP addresses for use in a channel.

In order to determine whether a system has been compromised by Trinity, just scan port 33270 (TCP) for any connection since it is reported that the Trinity port-shell may be installed there. [33]

Chapter 4

Monitoring IRC Traffic

To better understand the way IRC works, what characteristics it has and how network traffic, especially IRC traffic, is represented in Cisco NetFlow data, several analyses were made. Some of them are shortly presented in this chapter. A brief introduction to Cisco NetFlow will be given too.

The network measurements of IRC server traffic were made on "the IRC server" geneva.ch.eu.undernet.org.

4.1 Flow-level Internet traffic data (Cisco NetFlow)

Cisco NetFlow [43] was developed and patented at Cisco Systems in 1996 and is now the primary network accounting technology in the industry. It regards network traffic not as a heap of single packets but rather as a collection of flows, each flow describing one half of a packet stream between two hosts. There is no information about the data being sent in the IP packets like, for instance, HTTP headers. Not even the size of the single packets is known, only the total number of packets in a flow and their cumulated sizes.

NetFlow starts a new log entry for existing connections every fifteen minutes. That means one flow lasts at most fifteen minutes (or less if the connection is idle for longer than a predefined idle time), after that the current entry will be closed and a new flow entry will be started. [44]

An example of NetFlow entries is shown later in Table 5.4.

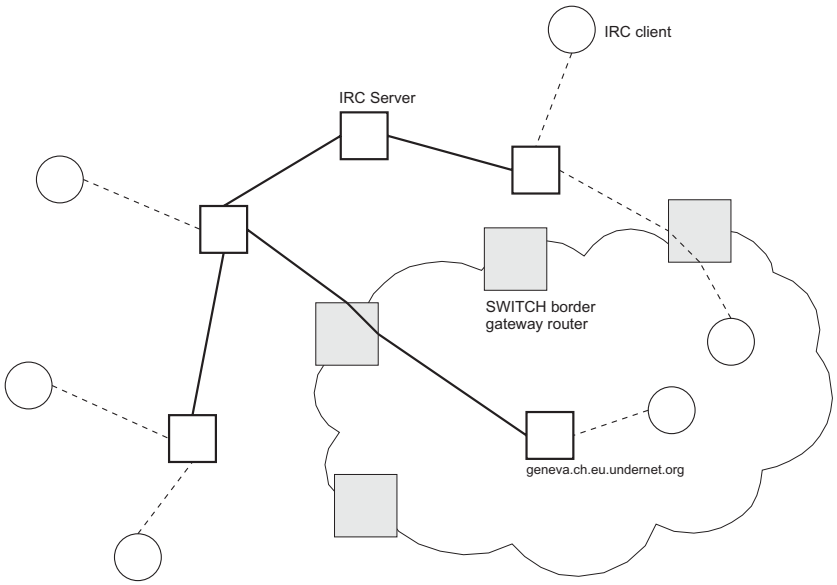


Figure 4.1: DDoSVax network topology

4.2 Network configuration

As already explained, we have access to NetFlow data collected on the four border gateway routers of SWITCH (see Figure 4.1). In principle any network traffic from or to the outside of SWITCH can be routed through one or more of these four routers. Therefore it is absolutely possible, that some traffic belonging to the same connection will appear in the NetFlow data of more than one border gateway router.

Moreover it is important to know, that the IRC server we considered is directly connected to one of these border gateway routers. If one wants to analyse the traffic from and to the IRC server found in the NetFlow data, then it suffice to look only at the flows captured by the router to which the IRC server is directly connected.

4.3 Analysis of NetFlow data over time

Thanks to some already available software tools (e.g. *netflow_to_text*) of the DDoSVax project, it is possible to read binary NetFlow data files and get the needed information (source/destination IP, source/destination port, number of packets, size of flow, start/end time, etc.) back from the software in a human readable way.

To analyse (e.g. the number of packets sent from a server per minute)

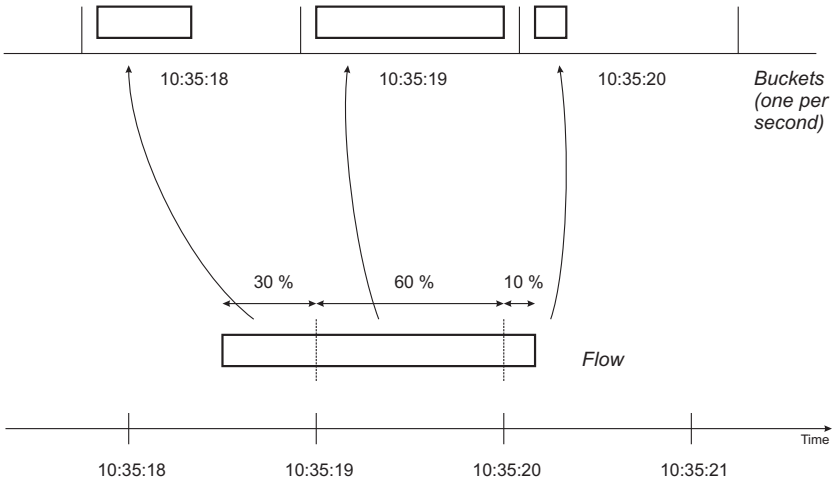


Figure 4.2: Bucket analysis

captured flows over time, the bucket analysis shown in Figure 4.2 was used.

If for example a flow starts some milliseconds after the time 10:35:18 and lasts some milliseconds longer than the time 10:35:20, then the flow (e.g. the number of packets) will be divided in a linear way (respecting the amount of time overlapping each second) into three parts and added to the buckets of every time interval (for this example one second). Having, as in Figure 4.2, a flow containing 100 packets, will result in the three buckets having 30, 60 and 10 "packets" added.

4.3.1 Two-day analysis of IRC traffic received and sent by an IRC server

Figure 4.3 and Figure 4.4 show such a bucket analysis. They show a two-day analysis of the cumulated traffic (number of Bytes) reaching and leaving an IRC server on the known IRC ports. The time interval of every bucket was set to 60 seconds.

As one certainly expected there is a cyclical variation of the traffic over day time, indicating that probably most of the chatters are active between eight and twelve o'clock in the evening. The omnipresent jitter is due to the small size of the buckets (60 seconds).

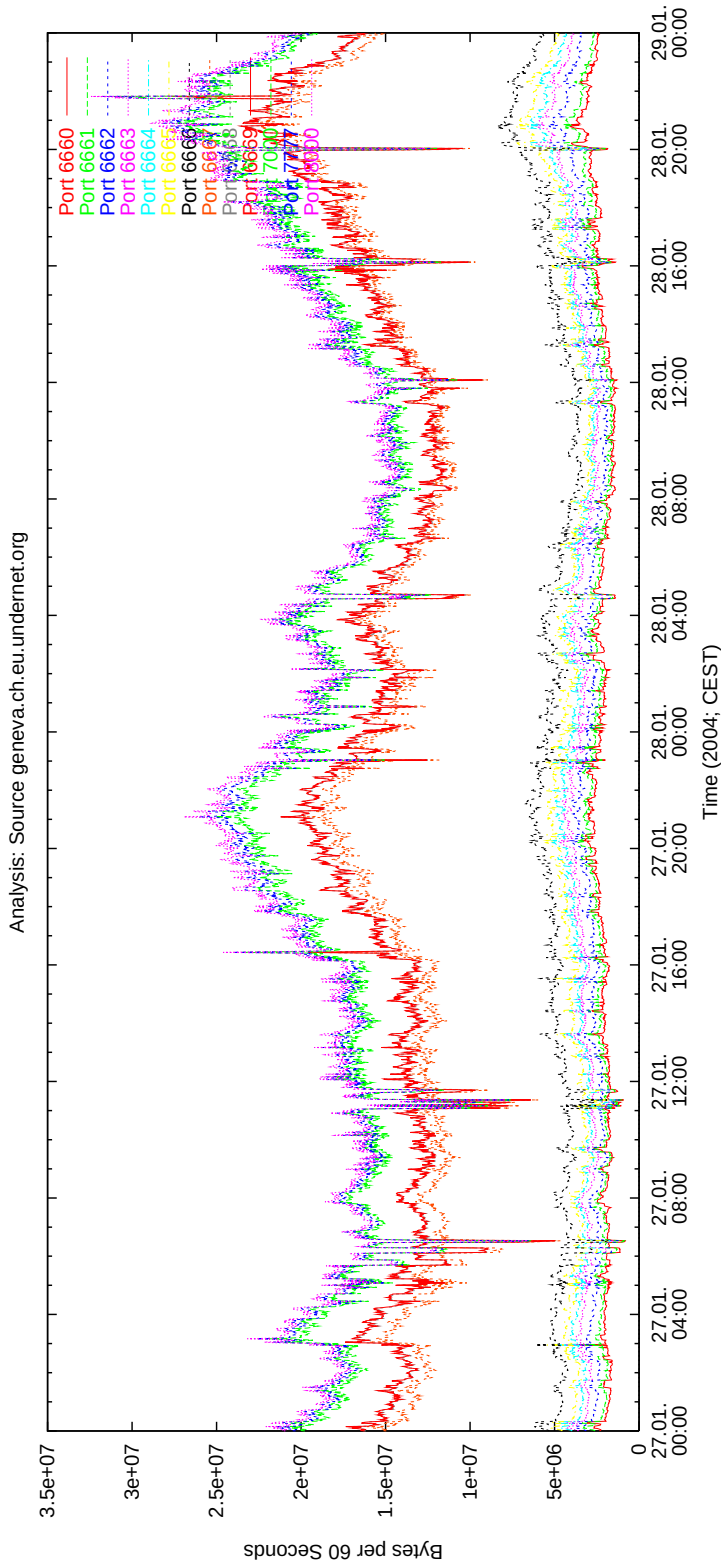


Figure 4.3: Two-day analysis of cumulated (by port) IRC traffic coming from geneva.ch.eu.undernet.org

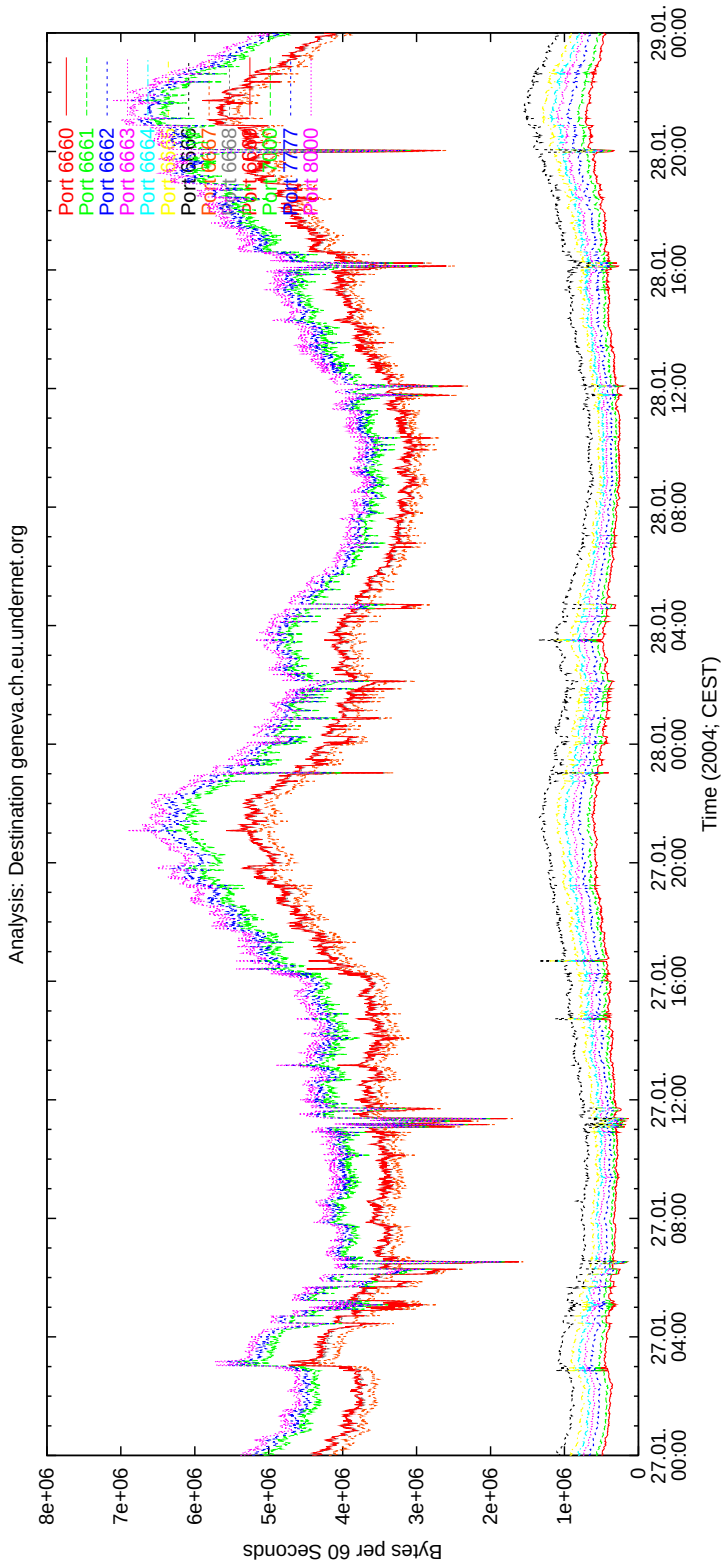


Figure 4.4: Two-day analysis of cumulated (by port) IRC traffic going to geneva.ch.eu.undernet.org

4.4 Analysis of full IRC network traffic

For the purposes of this thesis it turned out, that *tcpdump* [45] was the best and handiest tool to capture full network traffic (including payload).

A dump file created with *tcpdump* could then be analysed with *Ethereal* [46] or with the Perl script "ircsniffer.pl" (see Appendix B.3) originally written by Pascal Gloor and slightly adapted by the author of this thesis. The latter script can either directly sniff on a network interface or read from a dump file, but needs to get IRC traffic on the well-known IRC ports. Its output are human readable IRC messages, the connections the messages come from and the time the messages were sent.

With the tools described above it is now possible to either do a kind of bucket analysis to compare full network traffic to NetFlow data or to search for special IRC messages and try to find out how they look like in NetFlow data.

4.5 Scenarios

One idea to detect IRC bots is to find obvious patterns in the time behaviour of the network traffic IRC bots generate. For this purpose two scenarios were developed. Scenario I simulates a normal chat session which could occur between human users, whereas Scenario II, based on the findings in Chapter 3, simulates the behavior malicious bots could have. All hosts participating from the outside of SWITCH belonged to PlanetLab [47].

Since this kind of investigation did not result in simple and manageable time patterns for a possibly reliable detection of IRC bots, not more time was spent into further analysis of these graphs. Another, later explained, approach seemed more promising.

4.5.1 Scenario I

Figure 4.5 shows the network topology used for Scenario I. Clients 1 to 5 (nicknames: ddv_argan, ddv_clean, ddv_beral, ddv_tione, ddv_angel) represent the five different participants of the chat. Client 7 was used as channel operator. The "spoken" text (see Appendix B.1) was issued from a play written by Molière.

The resulting bucket analysis of the simulated human chat can be found in Figures 4.6 and 4.7. One part of each graph is the analysis of the flow data, the other part is the analysis of the also done full capturing (every single packet with payload).

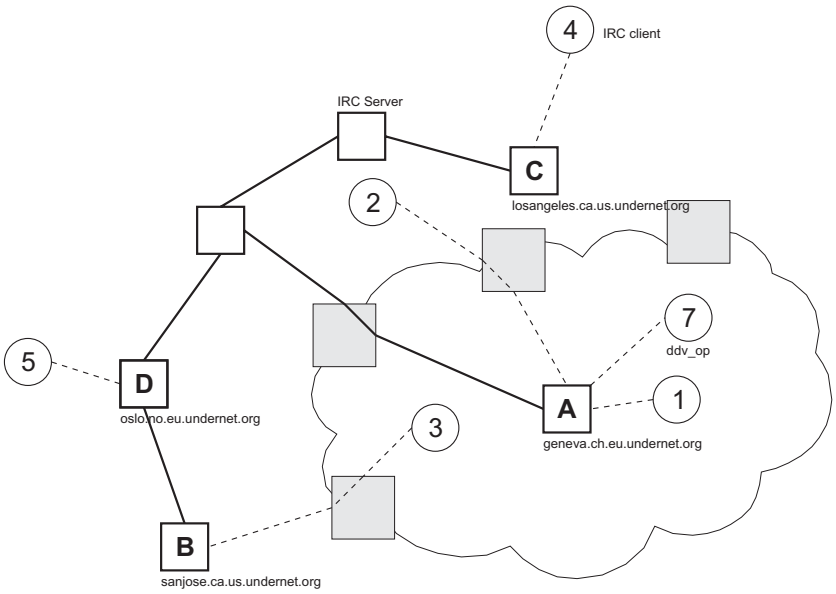


Figure 4.5: Network topology: Scenario I

Figure 4.6 clearly shows that all messages (including simple TCP Acks without IRC payload) coming from client 2 are aggregated into three flows (the arrows in the figure point at the start times of the flows) for the duration of the chat. The many small peaks in Figure 4.6 (about 50 Bytes in size) are TCP Acks sent back to the server. In Figure 4.7 one can see, that the largest part of the conversation (10:36 to 10:39) is aggregated into only one flow.

As a matter of fact, this IRC chat (with a total of 24 "spoken" messages) lasting more than six minutes is in one direction represented by three flows and in the other direction by four flows.

4.5.2 Scenario II

Scenario II, shown in Figure 4.8, tried to reproduce the behaviour known from DDoS attacks based on IRC: Several clients join the same channel and wait for instructions from the master. As soon as the master (client 6) sent the command "`!ready`", every single client answered with a message "`**ddv_slvXY** ready for test`".

The traffic between 10:47 and 10:49 seen in Figures 4.9 and 4.10 contains the login to the server, followed by the joining of channel "#ddv_ddvax".

At about 10:50:40 (exactly three minutes after the beginning of the login procedure) one can see an exchange of Ping and Pong messages.

Around 10:53:10 the "`!ready`" command occurred. The latest traffic peak

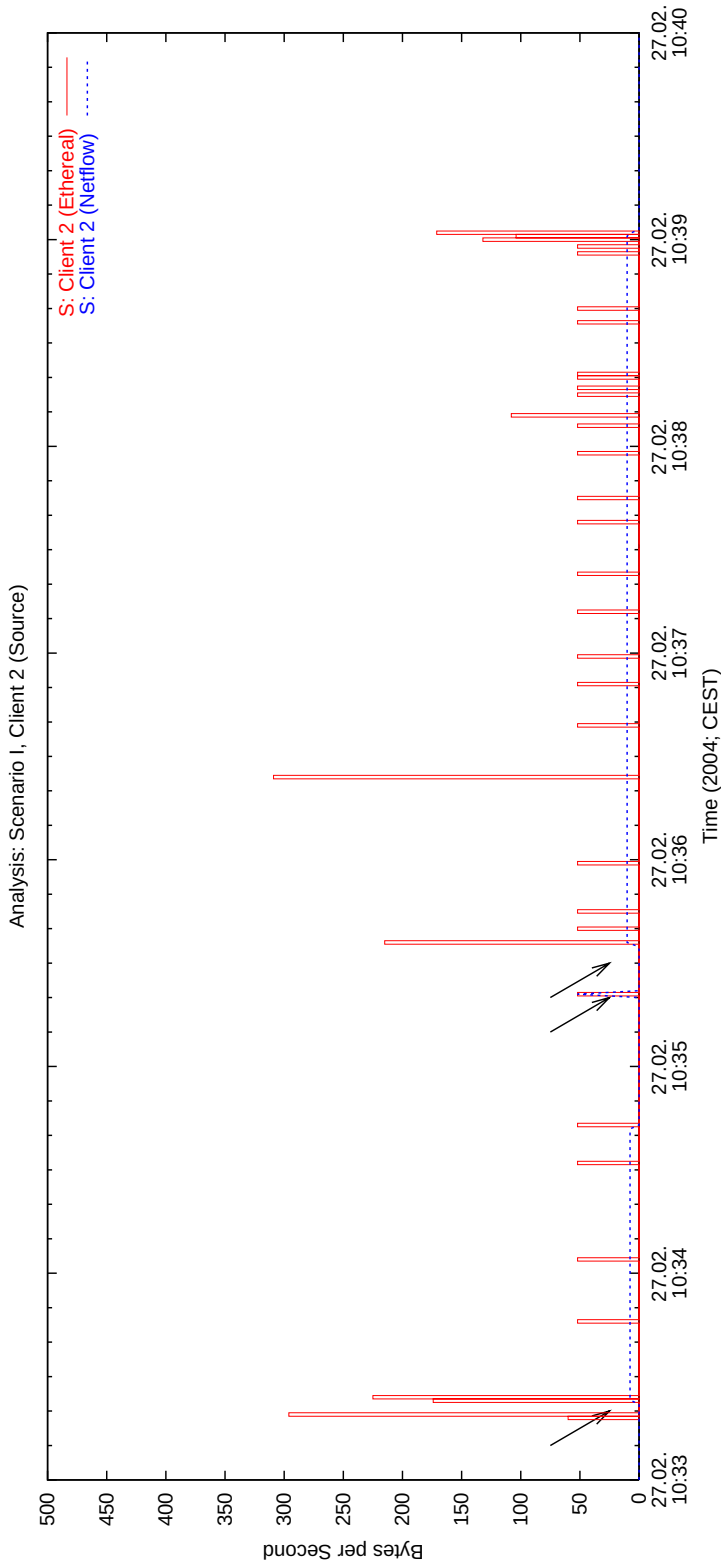


Figure 4.6: Analyzed Scenario I: Traffic with Client 2 as source

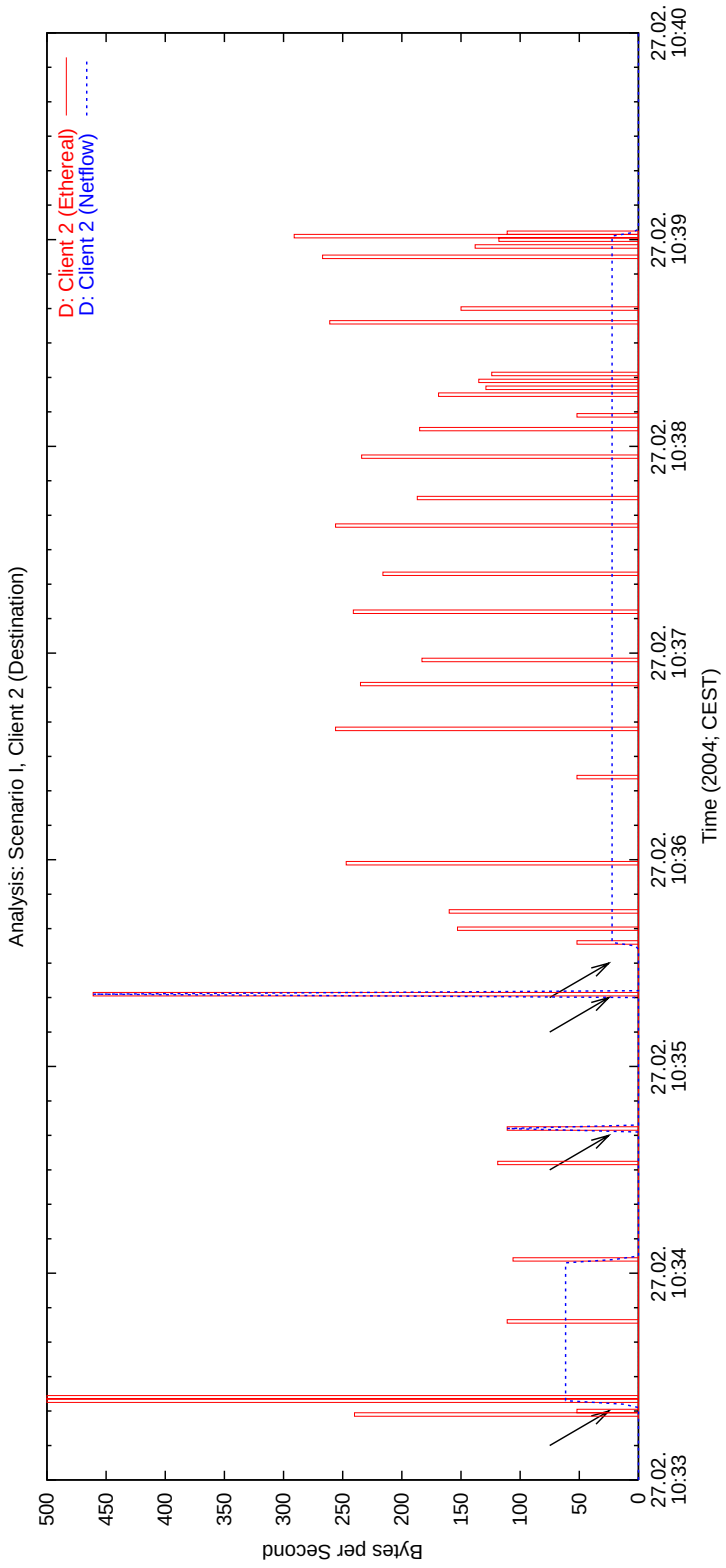


Figure 4.7: Analyzed Scenario I: Traffic with Client 2 as destination

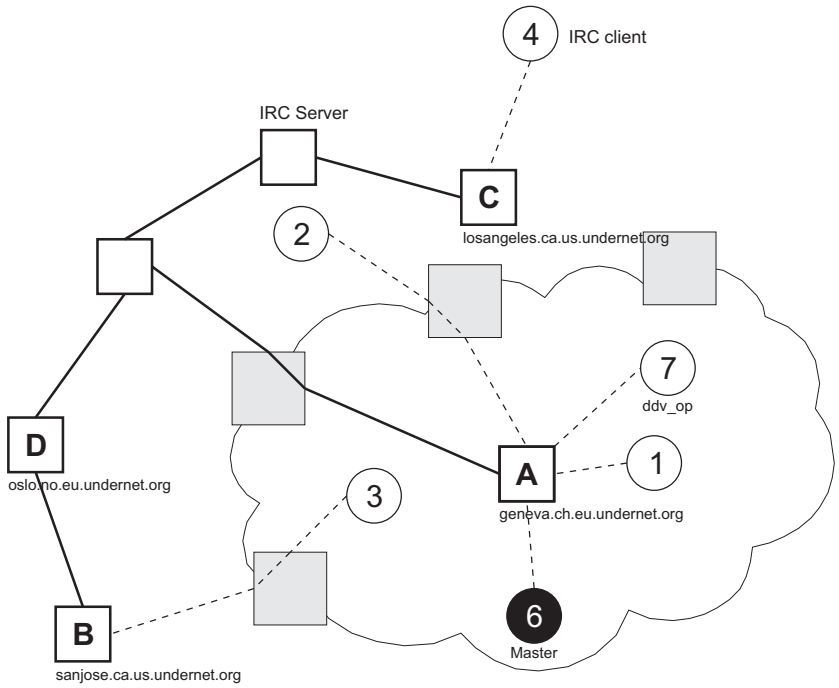


Figure 4.8: Network topology: Scenario II

at 10:55 shows the "`!end`" command, which instructs the slaves to send a message "`**ddv_slvXY** leaves`" and quit the channel.

One should also notice, that in Figure 4.9 one packet (Ack) was not found in the NetFlow data (10:48:20).

Since the flow idle time was largely exceeded between the launch of each command, the messages resulting from "`!ready`" and "`!end`" are, in each case, all found in one flow.

When only regarding the flows in Figures 4.9 and 4.10, it is interesting to see, that a flow with a high peak found in one direction occurs at the same time a flow with a small peak is found in the other direction.

4.5.3 Bot software

The Perl code of the software used for Scenario I (chatter.pl) and Scenario II (slave.pl) can be found in Appendix B.1 and B.2. In principle both Perl scripts have the ability to log on an IRC server, join a channel and then read messages sent to this channel. Depending on the messages the bots get, they can send responses back to the channel.

Both bots are based on the "HelloBot" found on [48]. Be aware that the used Perl module Net::IRC, which must be installed to be able to run the

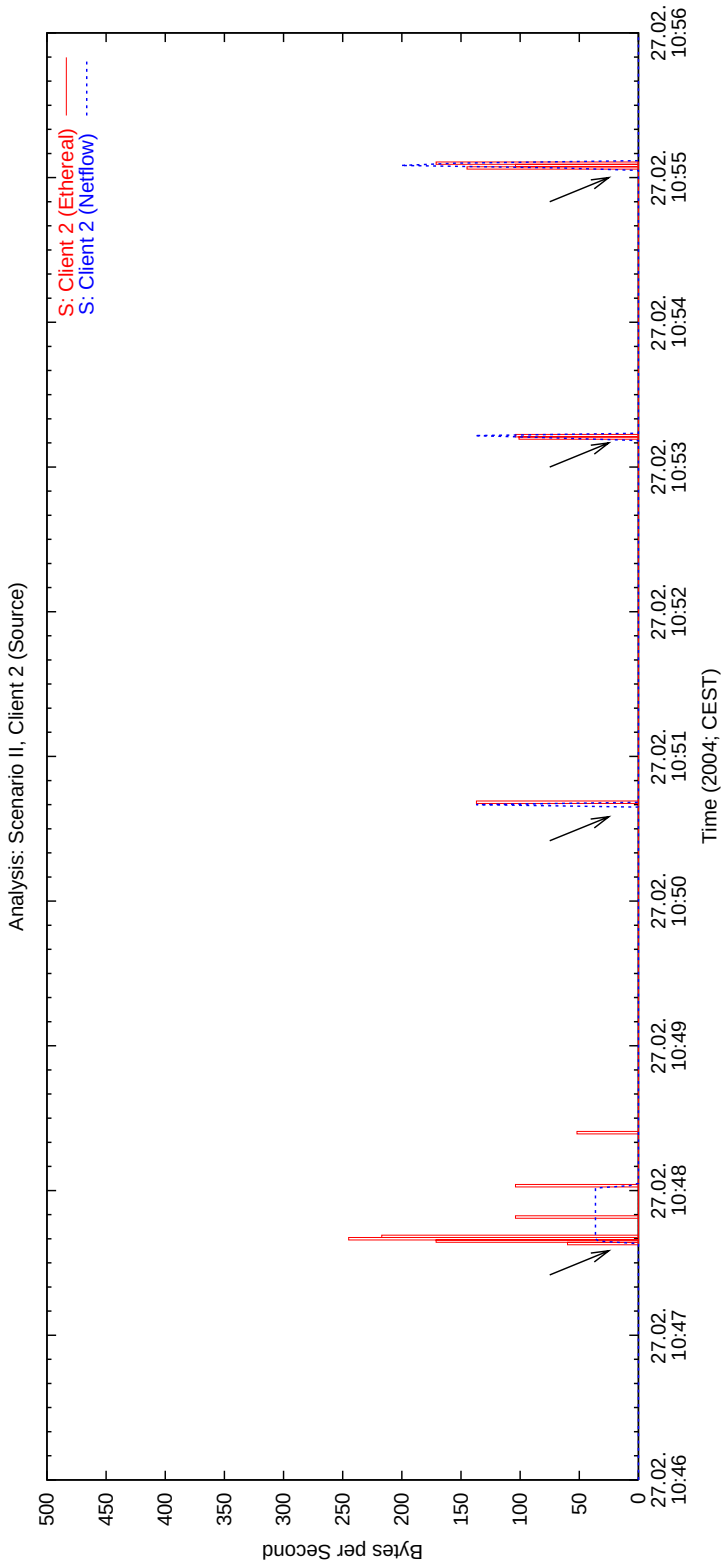


Figure 4.9: Analysed Scenario II: Traffic with Client 2 as source

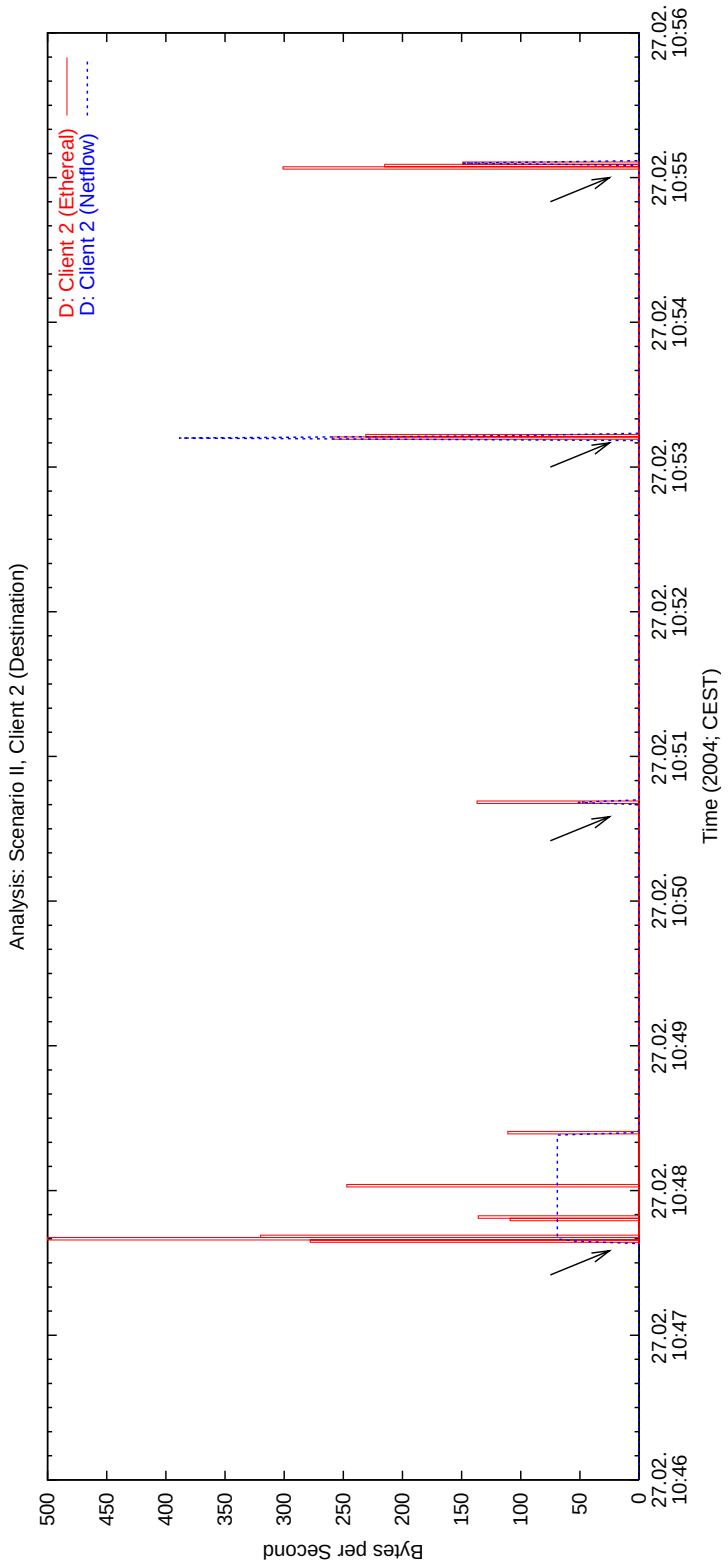


Figure 4.10: Analysed Scenario II: Traffic with Client 2 as destination

bot, is outdated and thus the bot may not run correctly on some computers. The author of the “HelloBot” recommends to use the POE::Component::IRC module.

Chapter 5

IRC Attack Preparation Detection Signatures

Having studied IRC and its protocol and illustrated the way IRC-based DDoS attacks work in previous chapters, the purpose of this chapter is to explain the steps made to develop an algorithm to detect possible botnets.

5.1 Ideas for detecting bots

The question we asked ourselves was:

- How can one find out, whether an IRC connection belongs to a normal IRC client used by a human user or to a bot respectively to a "machine"?

If one can answer this question and provide an algorithm which does the distinction, then this would certainly be a big step in the right direction.

It would also be of interest for an attack (preparation) detection to be able to find out which channels a user (resp. "connection") has joined, and this only by analysing network traffic (in this case NetFlow data).

We know from Chapter 3, that bots having joined the specified attacker channel, will in most cases just wait for commands. Therefore they do not send any messages (except Pong messages to the server) to the channel and can thus be considered as inactive connections.

Besides the already given ideas above, there are some more proposals to detect bots or find characteristics of bots. The following list can be regarded as a kind of brainstorming:

- Analysis of the IRC server-to-server protocol (see Section 2.5.2)

- Install a malicious bot in a secured environment, then capture the emerging network traffic and search for typical patterns. Also observe the behaviour in the control channel. → How does typical bot traffic look like?

- Are there messages, that, due to their size and frequency, can not be originating from a human user? There is a threshold of the typing rate of humans.

- Like in Section 4.5 one could think of scenarios with much more participants (bots). Are there patterns based on the chronological order or the quantity or size of messages/flows?

- A botnet consists of bots joining and leaving a channel. It might be, that a bot will not send the `/PART` and `/QUIT` commands when leaving the channel and disconnecting from the server (because an infected host is shut down without warnings to the bot). This may result in unanswered Ping messages.

- A suddenly large amount of messages to a channel might indicate answers to a command concerning all bots.

5.1.1 Outline of a possible botnet detection algorithm using analysing NetFlow data

First of all, we know that bots remain inactive most of the time. Because every normal user may have the same behavior, this can't be the only characteristic describing a bot. Consider the step of finding inactive clients as a one of multiple possible "filters" to achieve the goal of finding hosts respectively zombies being part of a botnet.

As a second step one would be interested to know which (inactive) client or user is belonging to which channel, respectively which clients/users are belonging to the *same* channel. If this could be done, then large groups of inactive clients belonging to the same channel would be suspicious.

Finally, if one was able to group inactive clients by channel membership, one could imagine a last filter which would analyse the traffic a channel generates and search for typical bot/botnet patterns (e.g. a sudden and simultaneous packet flood from the clients to the channel, which could signalise a response to a command the master sent).

In summary an outline of a possible botnet detection algorithm might be:

1. Find inactive "clients" (see Section 5.2).

2. Classify and group inactive "clients" by channel membership.

3. Analyse IRC traffic by channel and search for characteristic botnet traffic.

For the steps two and three no algorithm was developed. Therefore they will not be discussed further in this documentation. Subsection 5.1 gives some ideas on how to proceed after step one.

5.2 Detection of inactive connections

This section discusses the first step (find inactive clients) of the possible detection algorithm presented in Section 5.1.1. When referring to this first part of the possible detection algorithm, we will talk of the "Ping-Pong algorithm".

The reason for this name is rather simple. Every client which is connected to an IRC server and is idle for longer than a predefined period (three minutes for "our" IRC server) will be ping-ed by the server (IRC Ping message). The client's response to the Ping message is an IRC Pong message. If the client does not respond, the server will assume that the connection does not need to exist anymore and thus terminates it.

So, finding IRC Pong messages in the NetFlow data and being able to assign them to a connection will allow us to detect inactive clients, respectively possible bots.

The input of the "Ping-Pong algorithm" is a set of NetFlow data, the output is a set of inactive connections.

5.2.1 Ping and Pong signatures

To be able to detect inactive connections from NetFlow data we need to exactly know how these Ping and Pong messages look like. For this purpose let us first remember the definitions of those messages.

In [7] the Ping message is defined as follows:

> Command: `PING`
> Parameters: `<server1> [ <server2> ]`
>
> The `PING` command is used to test the presence of an active client or server at the other end of the connection. Servers send a `PING` message at regular intervals if no other activity detected coming from a connection. If a connection fails to respond to a `PING` message within a set amount of time, that connection is closed. A `PING` message MAY be sent even if the connection is active[1].

[1] Because this would generate unneeded traffic on the network, this is normally not done.

> When a PING message is received, the appropriate PONG message must be sent as reply to <server1> (server which sent the PING message out) as soon as possible. If the <server2> parameter is specified, it represents the target of the ping, and the message gets forwarded there.
>
> Examples:
>
> – PING tolsun.oulu.fi ; Command to send a PING message to server
> – PING WiZ tolsun.oulu.fi ; Command from WiZ to send a PING message to server "tolsun.oulu.fi"
> – PING :irc.funet.fi ; PING message sent by server "irc.funet.fi"

On the other hand, a Pong message has the following definition [7]:

> Command: PONG
> Parameters: <server> [<server2>]
>
> PONG message is a reply to a ping message. If parameter <server2> is given, this message MUST be forwarded to the given target. The <server> parameter is the name of the entity who has responded to the PING message and generated this message.
>
> Example:
>
> – PONG csd.bu.edu tolsun.oulu.fi ; PONG message from csd.bu.edu to tolsun.oulu.fi

As an example, if the IRC server (geneva.ch.eu.undernet.org) sends the message

```
PING :Geneva.CH.EU.Undernet.org
```

to a client connected to it, the message

```
PONG :Geneva.CH.EU.Undernet.org
```

will be sent as a response from that client.

Measurements[2] with different clients (*mIRC*, *ircII*, *ChatZilla*, self-written and non-malicious bot) revealed, that even if the IRC messages sent were always exactly the same (namely `PING :Geneva.CH.EU.Undernet.org` and `PONG :Geneva.CH.EU.Undernet.org`), the captured IP respectively TCP packets did not always have the same size. This is due to the fact, that some hosts and servers enable the "TCP Extensions for High Performance" [49], which may add 12 Bytes to the size of a TCP packet (resp. TCP header). This leads to two different possible sizes (20 Bytes without and 32 Bytes with this option enabled) of an empty TCP packet (e.g. an "Ack" without payload) as shown in Table 5.1.

[2]Packet sniffing occurred with the tools *Ethereal* [46] and *tcpdump* [45].

message / packet	min. size [Bytes]	max. size[Bytes]
<IRC server name>	not defined (0)	63
IRC[Ping] = PING: <IRC server name>	6	69
TCP[Ack / empty]	20	32 (TCP time stamp option)
TCP[IRC[Ping]]	26	101
TCP[Ack, IRC[Ping]]	26	101
IP[TCP[Ack / empty]]	40	52
IP[TCP[IRC[Ping]]]	46	121

Table 5.1: Minimal and maximal sizes of an IRC server name and IRC, TCP and IP messages resp. packets (IRC Ping).

Table 5.1 gives, starting from the restriction that an IRC server's name is maximally 63 characters long (see Chapter 2 and [8]), the maximal sizes of IRC Pings for different layers of the "OSI Reference Model". Since the word "ping" is four characters long, as is the word "pong", the mentioned sizes in Table 5.1 are also valid for Pong messages.

Now let us examine the chronological order of TCP packets exchanged during an "IRC Ping-Pong". Figure 5.1 demonstrates two different chronological orders of Ping and Pong messages. The part a) on the left was seen for the three IRC clients *mIRC*, *ircII* and *ChatZilla*. A self-written bot in Perl (see Appendix B.2) showed the behavior in part b).

The four time values also found in Figure 5.1 mean the following:

- s_S: start time of the server-to-client flow
- e_S: end time of the server-to-client flow
- s_C: start time of the client-to-server flow
- s_C: end time of the client-to-server flow

Ping-Pong signature

Taking the results of Table 5.1 and Figure 5.1 into consideration, then the signature found in Table 5.2 for searching Ping-Pong messages in NetFlow data can be used. It is also assumed, that the only IRC traffic between the server and the client are these periodically exchanged Ping-Pong messages. "Our" IRC server has set this period to three minutes. Since the NetFlow

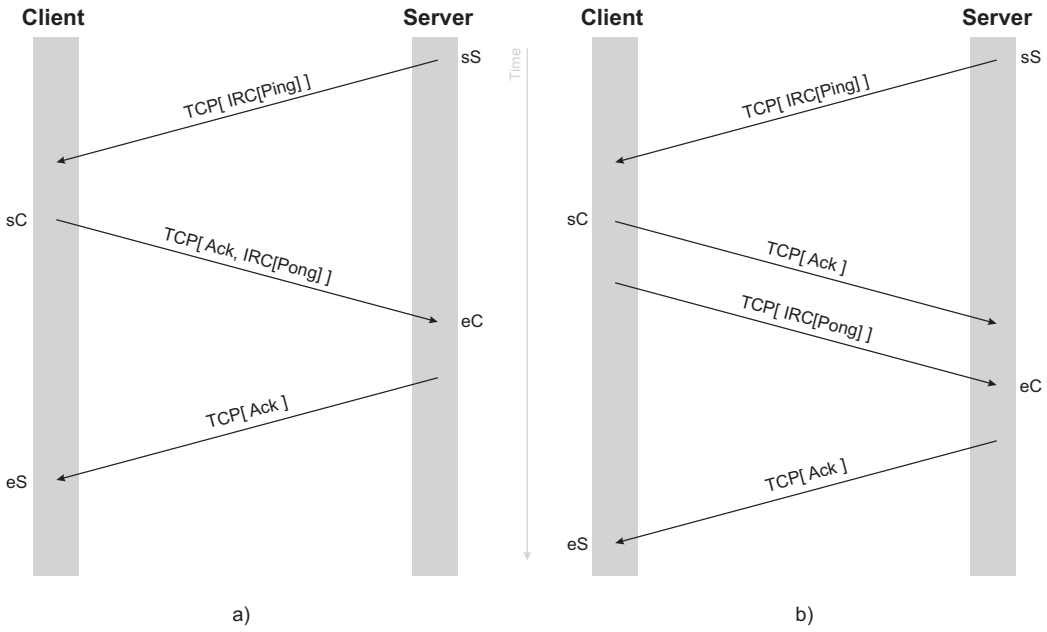


Figure 5.1: Sequence of TCP packets exchanged between an IRC server and client during an "IRC Ping-Pong". Time values s_S, e_S, s_C, e_C.

flow idle time out is in our case 30 seconds, we will get new flows every three minutes and are therefore sure, that there are only packets belonging to the Ping respectively Pong message in it.

It is obvious, that for the start and end times of the flows the following must apply:

$$s_S < e_S \tag{5.1}$$

$$s_C < e_C \tag{5.2}$$

Comparing the start time of the flow for the "Server → Client" connection to the start time of the flow for the "Client → Server" connection leads to

$$s_S < s_C \tag{5.3}$$

For the end times holds

$$e_S > e_C \tag{5.4}$$

and thus

$$s_S < s_C < e_C < e_S \tag{5.5}$$

The Pong flow is therefore "enclosed" by the Ping flow.

Pong signature

Another approach, which is only interested in finding Pong messages without being sure of a corresponding Ping message changes the signature shown above.

By slightly adapting the Ping-Pong signature, we get the Pong signature proposed in Table 5.3.

The time conditions already discussed in the Section about the Ping-Pong signature almost remain the same. Instead of the time s_S (time on which the server sends the first packet belonging to a Ping message) we use the times s_{SC} (start time of a flow from the server to the client) and e_{SC} (end time of a flow from the server to the client):

$$s_{SC} \leq s_S \tag{5.6}$$

$$e_{SC} \geq e_S \tag{5.7}$$

$$\Rightarrow s_{SC} \leq s_S < s_C < e_C < e_S \leq e_{SC} \tag{5.8}$$

or shorter

$$s_{SC} < s_C < e_C < e_{SC} \tag{5.9}$$

5.2.2 The Ping-Pong Algorithm

The proposed algorithm takes as input NetFlow data, will then analyse these data and search for connections matching the Pong signature discussed in Subsection 5.2.1. The output is a list containing client-to-server connections, which are hosts possibly running a bot or being inactive in an IRC channel.

The Ping-Pong algorithm in pseudo-code:

```
############################################################
# Pseudo code of the Ping-Pong algorithm                   #
############################################################

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

# Step 1: Filter for possible candidate connections
############################################################
for (each Flow) {
  getFromFlow(SrcIP, DstIP, SrcPort, DstPort, l, p)

  if (  (l >= 46)
       AND
        (l <= 173)
       AND
        ((p = 1) OR (p = 2))
       AND
```

```
        (DstPort = an IRC server port) ) {
    Connection = (SrcIP, DstIP, SrcPort, DstPort)

    if (Connection does not exist) {
      add Connection to Candidates
    }
  }
}

# Step 2: Search for Ping and Pong candidates
#############################################################
for (each Flow) {
  getFromFlow(SrcIP, DstIP, SrcPort, DstPort, l, p, s, e)

  if (DstPort = an IRC server port) {
    Connection = (SrcIP, DstIP, SrcPort, DstPort)
  }
  else {
    Connection = (DstIP, SrcIP, DstPort, SrcPort)
  }

  if ( Connection exists in Candidates ) {
    if (  (DstPort = an IRC server port)
         AND
          (((l >= 46) AND (l <= 121) AND (p = 1))
           OR
          ((l >= 86) AND (l <= 173) AND (p = 2))) ) {
      add Flow to PongCandidates
    }
    else if ( (p >= 2) AND (l >= 86) ) {
      add Flow to PingCandidates
    }
  }
}

# Step 3: Search for every Pong candidate a corresponding
#         Ping candidate
#############################################################
for (each PongCandidate) {
  getFromPongCandidate(SrcIP1, DstIP1, SrcPort1, DstPort1, l1, p1, s1, e1 ↩
      )

  Connection1 = (SrcIP1, DstIP1, SrcPort1, DstPort1)

    for (each PingCandidate) {
      getFromPingCandidate(SrcIP2, DstIP2, SrcPort2, DstPort2, l2, p2, s2 ↩
          , e2)

      Connection2 = (DstIP2, SrcIP2, DstPort2, SrcPort2)

      if (  (Connection1 = Connection2)
           AND
            (s2 <= s1)
           AND
            (e2 >= e1) ) {
        add Connection1 to InactiveConnections
        go to <LABEL>
      }
    }
  <LABEL>
}
```

5.2.3 Examples of Ping-Pong traffic

An example on how two IRC Ping-Pong messages (compare to Figure 5.1.a) occur in the NetFlow data is given in Table 5.4. Whereas Table 5.5 also shows IRC Ping-Pong messages (compare to Figure 5.1.b), but this time in the way they will appear in *tcpdump* data.

The values l, p, s, and d found in Table 5.4 mean the following:

- l: total number of Bytes in the flow (IP layer)
- p: total number of packets in the flow
- s: start time of the flow
- e: end time of the flow
- d: duration of the flow ($d = e - s$)

5.2.4 Difficulties and drawbacks

Suppose the client once sends

```
PONG :Geneva.CH.EU.Undernet.org
```

and some minutes later

```
PRIVMSG #ddv_ddvax :Hello guys!
```

Considering each of these two messages separately, they will have exactly the same appearance in the NetFlow data. Source IP, destination IP, source port, destination port, length, number of packets (in this case one) and duration (in this case zero) of the two flows will be exactly the same. Only the start and end times will differ.

The example above makes clear, that even if the constraints of the Pong signature (see Table 5.3) are fulfilled, it is by far not sure, that the flow contains an IRC Pong message.

Although the Ping-Pong and the Pong signatures in Subsection 5.2.1 at first sight seem to be a good and reliable method to detect inactive clients, they have an important drawback: By far not all inactive clients will be detected by an algorithm using these signatures.

The reason is simple. Even if a client is inactive, he will most of the time be inactive in a channel where other users or bots join, “chat” and leave that channel. As the servers send, respectively “forward”, messages sent to the channel to everyone in the channel, the Ping messages or packets

will not be the only packets aggregated into a flow for inactive server-to-client connections and therefore making the signatures of Tables 5.2 and 5.3 adequate only in a limited fashion. Also the Pong message will not appear in a flow consisting only of the Pong packets. The TCP Acks sent back to the server will also show up in the client-to-server flow.

Nevertheless the Ping-Pong algorithm was implemented and validated. The results will be shown in Chapter 6.

5.3 Countermeasures

It can be thought of two ways to impede the misuse of IRC for DDoS attacks. First, one could apply countermeasures in the (border gateway) routers or firewalls, thus the network itself. The simplest solutions would be to block all well-known IRC ports. But this would be counterproductive. On the one hand there are a lot of Internet users enjoying IRC chat sessions, on the other hand it is only a matter of configuration to use other ports for connecting clients to an IRC server.

Secondly, there is the possibility to improve the authentication process of IRC. As an example one could think of a much stronger need to identify oneself before being allowed to use an IRC service. This would only prevent the misuse of large IRC networks. Attackers still would have the possibility to "hijack" Internet hosts and install their own IRC servers without security mechanism.

Figure	Connection (from → to)	min. flow size [Bytes]	max. flow size [Bytes]	Number of packets in flow	Start time of flow	End time of flow
5.1.a)	Server → Client	$46 + 40 = 86$	$121 + 52 = 173$	2	s_S	e_S
5.1.a)	Client → Server	46	121	1	s_C	e_C
5.1.b)	Server → Client	$46 + 40 = 86$	$121 + 52 = 173$	2	s_S	e_S
5.1.b)	Client → Server	$46 + 40 = 86$	$121 + 52 = 173$	2	s_C	e_C

Table 5.2: Ping-Pong signature

Figure	**Connection (from → to)**	**min. flow size [Bytes]**	**max. flow size [Bytes]**	**Number of packets in flow**	**Start time of flow**	**End time of flow**
5.1.a)	Server → Client	$46 + 40 = 86$	undefined	≥ 2	s_{SC}	e_{SC}
5.1.a)	Client → Server	46	121	1	s_C	e_C
5.1.b)	Server → Client	$46 + 40 = 86$	undefined	≥ 2	s_{SC}	e_{SC}
5.1.b)	Client → Server	$46 + 40 = 86$	$121 + 52 = 173$	2	s_C	e_C

Table 5.3: Pong signature

Protocol	**Src IP**	**Dst IP**	**Src Port**	**Dst Port**	l	p	s	e	d
TCP	IRC server	A.B.C.D	6661	2936	113	2	15:38:58.047	15:38:58.143	0.096
TCP	A.B.C.D	IRC server	2936	6661	72	1	15:38:58.048	15:38:58.048	0.000
TCP	IRC server	A.B.C.D	6661	2936	113	2	15:41:58.042	15:41:58.145	0.103
TCP	A.B.C.D	IRC server	2936	6661	72	1	15:41:58.044	15:41:58.044	0.000

Table 5.4: Example 1: Two Ping-Pongs found in NetFlow data (mIRC client)

Prot.	Src IP	Dst IP	Src Port	Dst Port	Length of IP packet	Time	Payload (IRC)
TCP	IRC server	A.B.C.D	6661	46833	85	11:50:41.01182	PING :Geneva.CH.EU.Undernet.org
TCP	A.B.C.D	IRC server	46833	6661	52	11:50:41.32498	(Ack)
TCP	A.B.C.D	IRC server	46833	6661	85	11:50:41.32499	PONG :Geneva.CH.EU.Undernet.org
TCP	IRC server	A.B.C.D	6661	46833	52	11:50:41.42385	(Ack)

Table 5.5: Example 2: One Ping-Pong captured with *tcpdump* (self-written bot)

Chapter 6

Results and Evaluation of the Algorithm

In Chapter 5 our "Ping-Pong algorithm" was proposed. Based on the Pong signature given in Table 5.3 this algorithm is aimed at finding inactive IRC connections in a set of NetFlow data.

This chapter evaluates the reliability of the algorithm and mentions the quality of the data basis, namely the captured flows at the border gateway router the IRC server is connected to.

6.1 Evaluation of the Ping-Pong algorithm

In order to be able to make some statements about the reliability of the output of the actual version of the Ping-Pong algorithm (based on the Pong signature), tcpdump data was captured on port 6661 of the IRC server during two distinct one hour periods. The second measurement was done after SWITCH did some tuning of the routing tables and NetFlow parameters of the border gateway routers.

Tables 6.1 and 6.2 show, for the first measurement, the comparison of the effective number (per connection) of Pong messages in the tcpdump data and the number of Pong messages the algorithm has detected in the NetFlow data. A *connection* is characterised by the tuple: *(Source IP, Destination IP, Source Port, Destination Port)*. Furthermore there are six more columns which interpret the result in a way it is used for intrusion detection systems (IDS):

- *True positive* (TP) is the number of detected Pongs, which in reality are Pongs, i.e. the number of correctly detected Pongs.

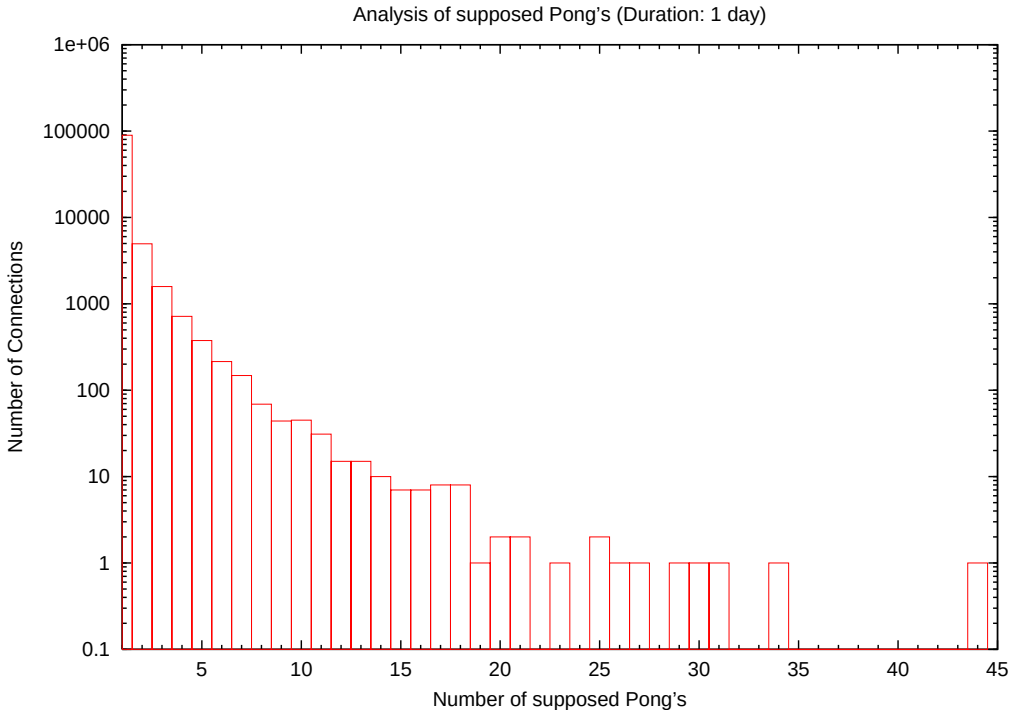


Figure 6.1: Pong analysis of one day

- *True negative* (TN) is the number of Pongs *not* detected by the algorithm, which in reality are Pongs.

- *False positive* (FP) is the number of Pongs detected by the algorithm, which in reality are *not* Pongs.

- (*False negative* (FN) is the number of *not* detected Pongs, which in reality also are *not* Pongs, i.e. the number of correctly discarded Pongs. This number is not available for this analysis.)

The average values for the columns TP(%), TN(%) and FP(%) of Tables 6.1 and 6.2, as well as the average values of the second measurement, are given in Table 6.3.

Referring to Figure 6.1, it is not surprising, that a one-day analysis of NetFlow data with the Ping-Pong algorithm (filtered for our IRC server, IRC server TCP ports 6660-6669, 7000, 7777, 8000) reveals, that there are only few clients being idle over a long period.

SrcIP	SrcPort	eff. Pongs	supp. Pongs	FP abs.	FP %	TN abs.	TN %	TP abs.	TP %
IP001	3973	21	0	0	0.0	21	100.0	0	0.0
IP002	4271	6	0	0	0.0	6	100.0	0	0.0
IP003	30183	20	8	0	0.0	12	60.0	8	40.0
IP004	65117	2	0	0	0.0	2	100.0	0	0.0
IP005	4952	20	15	0	0.0	5	25.0	15	75.0
IP006	1798	8	0	0	0.0	8	100.0	0	0.0
IP007	1079	20	8	0	0.0	12	60.0	8	40.0
IP008	2507	20	19	0	0.0	1	5.0	19	95.0
IP009	2914	16	0	0	0.0	16	100.0	0	0.0
IP010	21091	13	0	0	0.0	13	100.0	0	0.0
IP011	4773	20	21	1	4.8	0	0.0	20	100.0
IP012	2770	20	3	0	0.0	17	85.0	3	15.0
IP013	2964	4	2	0	0.0	2	50.0	2	50.0
IP014	2096	20	0	0	0.0	20	100.0	0	0.0
IP015	3370	20	10	0	0.0	10	50.0	10	50.0
IP016	1038	20	21	1	4.8	0	0.0	20	100.0
IP017	1094	20	8	0	0.0	12	60.0	8	40.0
IP018	33714	12	0	0	0.0	12	100.0	0	0.0
IP019	1114	20	0	0	0.0	20	100.0	0	0.0
IP020	4167	18	0	0	0.0	18	100.0	0	0.0
IP021	4930	20	0	0	0.0	20	100.0	0	0.0
IP022	1031	8	2	0	0.0	6	75.0	2	25.0
IP023	3382	19	22	3	13.6	0	0.0	19	100.0
IP024	1151	1	0	0	0.0	1	100.0	0	0.0
IP025	5149	5	4	0	0.0	1	20.0	4	80.0
IP026	62556	20	6	0	0.0	14	70.0	6	30.0
IP027	3477	5	0	0	0.0	5	100.0	0	0.0
IP028	64184	21	20	0	0.0	1	4.8	20	95.2
IP029	26667	20	17	0	0.0	3	15.0	17	85.0
IP030	1524	20	0	0	0.0	20	100.0	0	0.0
IP031	3356	3	0	0	0.0	3	100.0	0	0.0
IP032	63422	20	0	0	0.0	20	100.0	0	0.0
IP033	1052	2	0	0	0.0	2	100.0	0	0.0
IP034	3298	20	1	0	0.0	19	95.0	1	5.0
IP035	3632	20	7	0	0.0	13	65.0	7	35.0
IP036	61475	28	19	0	0.0	9	32.1	19	67.9
IP037	3058	20	24	4	16.7	0	0.0	20	100.0
IP038	1040	2	5	3	60.0	0	0.0	2	100.0
IP039	1113	12	10	0	0.0	2	16.7	10	83.3
IP040	1485	20	1	0	0.0	19	95.0	1	5.0
IP041	11676	20	6	0	0.0	14	70.0	6	30.0
IP042	1211	19	0	0	0.0	19	100.0	0	0.0
IP043	3041	5	0	0	0.0	5	100.0	0	0.0
IP044	1112	3	0	0	0.0	3	100.0	0	0.0
IP045	30557	3	1	0	0.0	2	66.7	1	33.3
IP046	34415	19	0	0	0.0	19	100.0	0	0.0
IP047	4396	19	24	5	20.8	0	0.0	19	100.0
IP048	2936	20	10	0	0.0	10	50.0	10	50.0
IP049	1278	20	0	0	0.0	20	100.0	0	0.0
IP050	1960	20	0	0	0.0	20	100.0	0	0.0
IP051	61062	12	17	5	29.4	0	0.0	12	100.0
IP052	3883	21	12	0	0.0	9	42.9	12	57.1
IP053	3013	19	2	0	0.0	17	89.5	2	10.5

Table 6.1: Measurement 1: Comparison of supposed Pong messages and effective Pong messages, Part a). All connections to our IRC server, port 6661, during one hour

SrcIP	SrcPort	eff. Pongs	supp. Pongs	FP abs.	FP %	TN abs.	TN %	TP abs.	TP %
IP054	3429	11	0	0	0.0	11	100.0	0	0.0
IP055	1155	1	0	0	0.0	1	100.0	0	0.0
IP056	3069	4	0	0	0.0	4	100.0	0	0.0
IP057	1575	1	0	0	0.0	1	100.0	0	0.0
IP058	1048	14	0	0	0.0	14	100.0	0	0.0
IP059	2708	7	0	0	0.0	7	100.0	0	0.0
IP060	2276	1	0	0	0.0	1	100.0	0	0.0
IP061	1219	4	0	0	0.0	4	100.0	0	0.0
IP062	5397	18	0	0	0.0	18	100.0	0	0.0
IP063	1040	2	1	0	0.0	1	50.0	1	50.0
IP064	1700	6	1	0	0.0	5	83.3	1	16.7
IP065	20525	3	0	0	0.0	3	100.0	0	0.0
IP066	1073	3	0	0	0.0	3	100.0	0	0.0
IP067	1049	3	0	0	0.0	3	100.0	0	0.0
IP068	2452	10	0	0	0.0	10	100.0	0	0.0
IP069	1066	9	1	0	0.0	8	88.9	1	11.1
IP070	2969	6	0	0	0.0	6	100.0	0	0.0
IP071	3015	6	0	0	0.0	6	100.0	0	0.0
IP072	1629	14	0	0	0.0	14	100.0	0	0.0
IP073	1038	1	0	0	0.0	1	100.0	0	0.0
IP074	1366	5	0	0	0.0	5	100.0	0	0.0
IP075	63558	1	0	0	0.0	1	100.0	0	0.0
IP076	4422	1	3	2	66.7	0	0.0	1	100.0
IP077	1093	1	0	0	0.0	1	100.0	0	0.0
IP078	3032	5	3	0	0.0	2	40.0	3	60.0
IP079	3219	8	0	0	0.0	8	100.0	0	0.0
IP080	50071	7	2	0	0.0	5	71.4	2	28.6
IP081	3649	8	4	0	0.0	4	50.0	4	50.0
IP082	1472	1	7	6	85.7	0	0.0	1	100.0
IP083	1033	4	0	0	0.0	4	100.0	0	0.0
IP084	4367	6	6	0	0.0	0	0.0	6	100.0
IP085	3674	4	1	0	0.0	3	75.0	1	25.0
IP086	20133	1	0	0	0.0	1	100.0	0	0.0
IP087	1093	5	1	0	0.0	4	80.0	1	20.0
IP088	1265	2	0	0	0.0	2	100.0	0	0.0
IP089	1086	1	0	0	0.0	1	100.0	0	0.0
IP090	4486	3	2	0	0.0	1	33.3	2	66.7
IP091	1073	1	4	3	75.0	0	0.0	1	100.0
IP092	3331	1	0	0	0.0	1	100.0	0	0.0
IP093	61490	2	0	0	0.0	2	100.0	0	0.0
IP094	61491	2	3	1	33.3	0	0.0	2	100.0
IP095	3004	1	0	0	0.0	1	100.0	0	0.0
IP096	3013	1	0	0	0.0	1	100.0	0	0.0
IP097	34121	0	1	1	100.0	0	0.0	0	0.0
IP098	2258	0	3	3	100.0	0	0.0	0	0.0
IP099	4190	0	4	4	100.0	0	0.0	0	0.0
IP100	1187	0	1	1	100.0	0	0.0	0	0.0
IP101	1036	0	1	1	100.0	0	0.0	0	0.0
IP102	2701	0	2	2	100.0	0	0.0	0	0.0
IP103	29358	0	2	2	100.0	0	0.0	0	0.0
IP104	2912	0	1	1	100.0	0	0.0	0	0.0
IP105	1088	0	1	1	100.0	0	0.0	0	0.0

Table 6.2: Measurement 1: Comparison of supposed Pong messages and effective Pong messages, Part b). All connections to our IRC server, port 6661, during one hour.

	Average	
	Measurement 1	**Measurement 2**
True positive (TP)	27.3 %	55.8 %
True negative (TN)	72.7 %	44.2 %
False positive (FP)	12.5 %	37.0 %
False negative (FN)	n/a	n/a

Table 6.3: Averages of the FP, TN and TP columns of Tables 6.1 and 6.2

	Average	
	Measurement 1	**Measurement 2**
Server → Client	33.7 %	26.2 %
Client → Server	15.3 %	4.7 %

Table 6.4: Packet loss rate in the NetFlow data. (Connections from/to geneva.ch.eu.undernet.org, port 6661)

6.2 Quality of the NetFlow data used for the evaluation

Because of observations (loss of packets) made during the analysis of the NetFlow data on which the Ping-Pong algorithm was tested, there was, as will be shown, the need to know about the quality and the reliability of the data basis.

For this purpose tcpdump data was captured (without packet loss) during two distinct one hour intervals on port 6661 of the IRC server. In a next step this traffic was compared to the flows found in the NetFlow data for the same time periods. Flows overlapping the start and end time of the hour were taken into account in a linear manner (e.g. the number of packets was divided by the duration of the flow and multiplied by the overlapping time). More details on how the comparison was made are given in the Perl script "compare_dumps_to_flows.pl" in Appendix B.4.

The results of this comparison are shown in Table 6.4. In one direction more than 30% of the traffic did not appear in the NetFlow data. After the tuning of the router parameters by SWITCH, the packet loss did reduce in both directions.

Chapter 7

Summary

In this last chapter some conclusions will be drawn and finally stated what needs to be done further.

7.1 Conclusions

There are many different factors that can influence a proper detection of IRC-based DDoS attacks. It starts with the unequal behaviour of different clients and bots and the unpredictable existence or absence of TCP options. It goes further with the fact, that every IRC message in the NetFlow data could be a completely different message and ends with the uncertainty of the captured NetFlow data.

It should also be taken into consideration that the analysis of the captured flows requires the processing of a huge amount of data. Even Scylla[1], an experimental research computer cluster, needed several hours to process a one day log of NetFlow data at the border gateway routers of SWITCH.

There is certainly the need for further measurements to make final conclusions about the quality of both, the Ping-Pong algorithm and the NetFlows. The apparent improvement of the flow quality did not, as could be expected, also improve the results of the algorithm. Although the amount of correctly identified Pongs did increase, the value of wrongly caught Pongs did also rise (see Tables 6.3 and 6.4). Adapted versions of the algorithm with more constraints (e.g. be sure that supposed Pongs have chronological distance of three minutes), could provide better results for the amount of "false positives", but would certainly also reduce the value for the "true positives".

The "good news" are, that IRC itself does not evolve very rapidly. Even if some IRC operators of large IRC networks have developed their own server

[1] `http://www.tik.ee.ethz.ch/~ddosvax/cluster/index.html`

software, the basic functionality remained the same. It is still possible to use every IRC client on every IRC network. Hence one can also conclude, that the way IRC is misused by bots will not change very soon. Naturally this does not mean, that the functionality and the threat of bots will not increase, but rather that continuing research on the topic is worthwhile.

7.2 Outlook

From my point of view it is important to find out the reason(s) of flow loss. Especially short flows (duration), as they occur when a client is idle, seem to be affected. If the flow quality remains a problem to test further algorithms, there is the possibility to use tools like *nProbe*[2] and *tcpreplay*[3] to convert network traffic captured with *tcpdump* into NetFlow data. However, in high-speed networks you most probably will run into performance problems with such a software based solution.

It might also be worth to build up bigger, but still manageable, scenarios in a secured environment. A large number of deliberately infected hosts (with a malicious bot) could be advised to connect to an own IRC server (Appendix C contains working configuration files for the Undernet IRC server software) and then instructed to attack a host in the secured environment. Maybe there are not yet found patterns in the behaviour of bots.

Further ideas found in Section 5.1 could also be looked at. Especially methods that improve step one and solve the problems of steps two and three mentioned in Section 5.1.1 are of interest.

[2] `http://www.ntop.org`
[3] `http://tcpreplay.sourceforge.net/`

Appendix A

A Short Chat

If you use the *ircII* client to connect to "geneva.ch.eu.undernet.org" as user "ddv_argan", join channel "#ddv_ddvax", say "Hello everybody!" and then leave the channel and disconnect from the server you will see the following appear on your terminal:

```
*** Connecting to port 6667 of server
    geneva.ch.eu.undernet.org
*** Looking up your hostname
*** Checking Ident
*** Found your hostname
*** Got ident response
*** Welcome to the UnderNet IRC Network via SPALE Network,
    ddv_argan
*** /etc/irc/script/local V0.5 for Debian finished. Welcome
    to ircII.
*** If you have not already done so, please read the new
    user information with /HELP NEWUSER
*** Your host is Geneva.CH.EU.Undernet.org, running version
    u2.10.11.06
*** This server was created Sun Jan 11 2004 at 04: 06:09 CET
*** umodes available dioswkgx, channel modes available
    biklmnopstvr
*** WHOX WALLCHOPS WALLVOICES USERIP CPRIVMSG CNOTICE
    SILENCE=15 MODES=6 MAXCHANNELS=15 MAXBANS=45 NICKLEN=9
    MAXNICKLEN=15 are supported by this server
*** TOPICLEN=160 AWAYLEN=160 KICKLEN=160 CHANTYPES=#&
    PREFIX=(ov)@+ CHANMODES=b,k,l,imnpstr
    CASEMAPPING=rfc1459 NETWORK=UnderNet are supported by
    this server
*** There are 45824 users and 71385 invisible on 38 servers
*** There are 96 operators online
*** 60 unknown connection(s)
*** 49046 channels have been formed
```

```
*** This server has 3245 clients and 1 servers connected
*** Highest connection count: 7883 (7882 clients)
*** - Geneva.CH.EU.Undernet.org Message of the Day -
*** Welcome to the Swiss Undernet IRC Server
*** Type /MOTD to read the AUP before continuing using this
    service.
*** The message of the day was last changed: 2003-12-5 16:23
*** on 1 ca 1(4) ft 10(10) tr
*** ddv_argan (ddosvax@pc-abcd.ethz.ch) has joined channel
    #ddv_ddvax
*** #ddv_ddvax 1076504066
> Hello everybody!
*** ddv_argan has left channel #ddv_ddvax
```

The short "conversation" above will generate all the messages (without `Client:` or `Server:`) below, which are exchanged between the client and the server:

```
Client: NICK ddv_argan
Server: NOTICE AUTH :*** Looking up your hostname
Client: USER ddosvax debian-sr geneva.ch.eu.undernet.org
        :Debian User
Server: NOTICE AUTH :*** Checking Ident
Server: NOTICE AUTH :*** Found your hostname
Server: NOTICE AUTH :*** Got ident response
Server: PING :1854915901
Client: PONG :1854915901
Server: :Geneva.CH.EU.Undernet.org 001 ddv_argan :Welcome to
        the UnderNet IRC Network via SPALE Network, ddv_argan
Server: :Geneva.CH.EU.Undernet.org 002 ddv_argan :Your host
        is Geneva.CH.EU.Undernet.org, running version
        u2.10.11.06
Server: :Geneva.CH.EU.Undernet.org 003 ddv_argan :This
        server was created Sun Jan 11 2004 at 04:06:09 CET
Server: :Geneva.CH.EU.Undernet.org 004 ddv_argan
        Geneva.CH.EU.Undernet.org u2.10.11.06 dioswkgx
        biklmnopstvr bklov
Server: :Geneva.CH.EU.Undernet.org 005 ddv_argan WHOX
        WALLCHOPS WALLVOICES USERIP CPRIVMSG CNOTICE
        SILENCE=15 MODES=6 MAXCHANNELS=15 MAXBANS=45
        NICKLEN=9 MAXNICKLEN=15 :are supported by this
        server
Server: :Geneva.CH.EU.Undernet.org 005 ddv_argan
        TOPICLEN=160 AWAYLEN=160 KICKLEN=160 CHANTYPES=#&
        PREFIX=(ov)@+ CHANMODES=b,k,l,imnpstr
        CASEMAPPING=rfc1459 NETWORK=UnderNet :are supported
        by this server
Server: :Geneva.CH.EU.Undernet.org 251 ddv_argan :There are
        45824 users and 71385 invisible on 38 servers
```

```
Server: :Geneva.CH.EU.Undernet.org 252 ddv_argan 96
        :operator(s) online
Server: :Geneva.CH.EU.Undernet.org 253 ddv_argan 60 :unknown
        connection(s)
Server: :Geneva.CH.EU.Undernet.org 254 ddv_argan 49046
        :channels formed
Server: :Geneva.CH.EU.Undernet.org 255 ddv_argan :I have
        3245 clients and 1 servers
Server: :Geneva.CH.EU.Undernet.org NOTICE ddv_argan :Highest
        connection count: 7883 (7882 clients)
Server: :Geneva.CH.EU.Undernet.org 375 ddv_argan :-
        Geneva.CH.EU.Undernet.org Message of the Day -
Server: :Geneva.CH.EU.Undernet.org 372 ddv_argan :Welcome to
        the Swiss Undernet IRC Server
Server: :Geneva.CH.EU.Undernet.org 372 ddv_argan :Type /MOTD
        to read the AUP before continuing using this service.
Server: :Geneva.CH.EU.Undernet.org 372 ddv_argan :The
        message of the day was last changed: 2003-12-5 16:23
Server: :Geneva.CH.EU.Undernet.org 376 ddv_argan :End of
        /MOTD command.
Server: :Geneva.CH.EU.Undernet.org NOTICE ddv_argan :on 1 ca
        1(4) ft 10(10) tr
Client: JOIN #ddv_ddvax
Server: :ddv_argan!ddosvax@pc-abcd.ethz.ch JOIN #ddv_ddvax
Server: :Geneva.CH.EU.Undernet.org 353 ddv_argan =
        #ddv_ddvax :@ddv_argan
Server: :Geneva.CH.EU.Undernet.org 366 ddv_argan #ddv_ddvax
        :End of /NAMES list.
Client: MODE #ddv_ddvax
Server: :Geneva.CH.EU.Undernet.org 324 ddv_argan #ddv_ddvax +
Server: :Geneva.CH.EU.Undernet.org 329 ddv_argan #ddv_ddvax
        1076504066
Client: PRIVMSG #ddv_ddvax :Hello everybody!
Client: PART #ddv_ddvax
Server: :ddv_argan!ddosvax@pc-abcd.ethz.ch PART #ddv_ddvax
Client: QUIT :Leaving
```

All of this was done within 16 TCP packets (without counting packets for establishing and terminating a TCP connection and the acknowledgements exchanged) containing all the 37 IRC messages above.

Appendix B

Source Code

B.1 chatter.pl

The Perl script chatter.pl (based on the "HelloBot" [48]):

```
#!/usr/bin/perl -w

use Net::IRC;
#use strict;

######################################################################
# BEGIN
#

use Time::HiRes qw(sleep);
use Getopt::Std;


# Programmargumente verarbeiten:
# Usage:
#   -c <Kanal>   Kanalname
#   -d <Datei>   Gespraechsdatei
#   -h           Benutzung (Hilfe)
#   -i <Server>  IRC-Server
#   -n <Nick>    Nickname
#   -p <PortNr>  Portnummer
#   -s <Dauer>   Sleep-Dauer (Sekunden), Intervall beim Buchstabentippen
#
# getopts: vergleiche Perl Kochbuch Kapitel 15.1
getopts("c:d:hi:n:p:s:", \%args);

# ARGUMENT -h
if ( $args{h} ) {
  print "++ Benutzung:\n";
  print "++   -c <Kanal>   Kanalname\n";
  print "++   -d <Datei>   Gespraechsdatei\n";
  print "++   -h           Benutzung (Hilfe)\n";
  print "++   -i <Server>  IRC-Server\n";
  print "++   -n <Nick>    Nickname\n";
  print "++   -p <PortNr>  Portnummer\n";
  print "++   -s <Dauer>   Sleep-Dauer (Sekunden), Intervall beim
      Buchstabentippen\n";
```

```
  exit;
}

# ARGUMENT -c
if ( $args{c} ) {
  $channelname = $args{c};
  print "++ Kanal: ", $channelname, "\n";
}
else {
  print "WRONG USAGE!!! --> Argument -c falsch\n";
  exit;
}

# ARGUMENT -d
if ( $args{d} ) {
  $Datei = $args{d};
  print "++ Datei: ", $Datei, "\n";
}
else {
  print "WRONG USAGE!!! --> Argument -d falsch\n";
  exit;
}

# ARGUMENT -i
if ( $args{i} ) {
  $ircserver = $args{i};
  print "++ IRC-Server: ", $ircserver, "\n";
}
else {
  print "WRONG USAGE!!! --> Argument -i falsch\n";
  exit;
}

# ARGUMENT -n
if ( $args{n} ) {
  $thisbotname = $args{n};
  print "++ Nickname: ", $thisbotname, "\n";
}
else {
  print "WRONG USAGE!!! --> Argument -n falsch\n";
  exit;
}

# ARGUMENT -p
if ( $args{p} ) {
  $PortNumber = $args{p};
  print "++ Port: ", $PortNumber, "\n";
}
else {
  print "WRONG USAGE!!! --> Argument -p falsch\n";
  exit;
}

# ARGUMENT -s
if ( $args{s} ) {
  $SleepDauer = $args{s};
  print "++ Sleep-Dauer: ", $SleepDauer, "\n";
}
else {
  print "WRONG USAGE!!! --> Argument -s falsch\n";
  exit;
}
```

```
100 # Gespraechs-Datei einlesen
    $t = 0;
102 open(DATEI, "< $Datei") or die "Konnte $Datei nicht oeffnen: $!\n";
    while ( defined($Zeile = <DATEI>) ) {
104     #print "++ ", $Zeile;

106     $TempString = substr($Zeile, 0, 1);

108     if ( $TempString ne "#" ) {
            @Eintraege = split /:::/, $Zeile;
110
            chomp($Eintraege[0]);
112         chomp($Eintraege[1]);

114         #print "++ ", $Eintraege[0], "\n";
            #print "++ ", $Eintraege[1], "\n";
116
            $messages[$t][0] = $Eintraege[0];
118         $messages[$t][1] = $Eintraege[1];

120         #print $messages[$t][0], ":::", $messages[$t][1], "\n";

122         $t = $t + 1;
        }
124 }
    my $AnzahlEintraegeDatei = $t;
126 print "++ Anzahl Eintraege Datei : ", $AnzahlEintraegeDatei, "\n";
    close(DATEI);
128

130 my $MsgNumber   = 0;
    my $MaxMessages = $AnzahlEintraegeDatei;
132 my $DefaultMessage = "";

134
    sub generateMessage {
136   my ($TempMessage) = @_;
      my $TempMessage2 = "";
138   my $Length = length($TempMessage);
      #print "Laenge: ", $Length, "\n";
140
      if ($Length > 400) {
142       print "Message zu lang!!!";
          exit;
144   }

146   for (my $i = 0; $i < $Length; $i++) {
        $TempMessage2 = $TempMessage2 . substr($TempMessage, $i, 1);
148     #print $TempMessage2, "\n";
        sleep($SleepDauer);
150   }

152   return $TempMessage2;
    }
154
    #
156 # END
    ####################################################################
158

160 # create the IRC object
```

```
my $irc = new Net::IRC;

# Create a connection object.  You can have more than one "connection" per
# IRC object, but we'll just be working with one.
my $conn = $irc->newconn(
        Server          => shift || $ircserver,
        Port            => shift || $PortNumber,
        Nick            => $thisbotname,
        Ircname         => 'I like to greet!',
        Username        => $thisbotname
);

# We're going to add this to the conn hash so we know what channel we
# want to operate in.
$conn->{channel} = shift || $channelname;

sub on_connect {
  #shift in our connection object that is passed automatically
  my $conn = shift;

  #when we connect, join our channel and greet it
  print "Joining channel $conn->{channel} ...";
  $conn->join($conn->{channel});
  print " done.\n";
  #$conn->privmsg($conn->{channel}, 'Hello everyone!');
  $conn->{connected} = 1;
}


######################################################################
# BEGIN
#

sub on_public {

  # on an event, we get connection object and event hash
  my ($conn, $event) = @_;

  # this is what was said in the event
  my $text = $event->{args}[0];
  my $nick = $event->{nick};

  if ($messages[$MsgNumber][0] eq $thisbotname) {
       $DefaultMessage = generateMessage($messages[$MsgNumber][1]);
       $conn->privmsg($conn->{channel}, $DefaultMessage);
       $MsgNumber = $MsgNumber + 1;
  }

  if ($MsgNumber == $MaxMessages) {
       sleep(2);
       $conn->privmsg($conn->{channel}, 'ByeBye');
       sleep(2);
       exit;
  }

  $MsgNumber = $MsgNumber + 1;

}

# add event handlers
$conn->add_handler('public', \&on_public);
```

```
222
#
224 # END
######################################################################
226

228 # The end of MOTD (message of the day), numbered 376 signifies we've connect
$conn->add_handler('376', \&on_connect);
230
# start IRC
232 $irc->start();
```

The input file ("-d" option of chatter.pl) containing the messages used for Scenario I (see Section 4.5.1): (The basis of the text was found on [50])

```
# THE IMAGINARY INVALID, Moliere
2 # Act III, Scene XXII
#
4 # Argan, Beralde, Angelique, Cleante, Toinette
# Angelique (337 Bytes)
6 ddv_angel:::Ah! What a delightful surprise! Father, since heaven has given you back to our love, let me here throw myself at your feet to implore one favour of you. If you do not approve of what my heart feels, if you refuse to give me Cleante for a husband, I conjure you , at least, not to force me to marry another. It is all I have to ask of you.
# Cleante (141 Bytes)
8 ddv_clean:::(THROWING HIMSELF AT ARGAN'S FEET). Ah! Sir, allow your heart to be touched by her entreaties and by mine, and do not oppose our mutual love.
# Beralde (37 Bytes)
10 ddv_beral:::Brother, how can you resist all this?
# Toinette (49 Bytes)
12 ddv_toine:::Will you remain insensible before such affection?
# Argan ( 135 Bytes)
14 ddv_argan:::Well, let him become a doctor, and I will consent to the marriage. (TO CLEANTE) Yes, turn doctor, Sir, and I will give you my daughter.
# Cleante (235 Bytes)
16 ddv_clean:::Very willingly, Sir, if it is all that is required to become your son-in-law. I will turn doctor; apothecary also, if you like. It is not such a difficult thing after all, and I would do much more to obtain from you the fair Angelique.
# Beralde (140 Bytes)
18 ddv_beral:::But, brother, it just strikes me; why don't you turn doctor yourself? It would be much more convenient to have all you want within yourself.
# Toinette (124 Bytes)
20 ddv_toine:::Quite true. That is the very way to cure yourself. There is no disease bold enough to dare to attack the person of a doctor.
# Argan (71 Bytes)
22 ddv_argan:::I imagine, brother, that you are laughing at me. Can I study at my age?
# Beralde (125 Bytes)
24 ddv_beral:::Study! What need is there? You are clever enough for that; there are a great many who are not a bit more clever than you are.
# Argan (104 Bytes)
26 ddv_argan:::But one must be able to speak Latin well, and know the different diseases and the remedies they require.
# Beralde (140 Bytes)
```

```
ddv_beral:::When you put on the cap and gown of a doctor, all that will come of itself, and you will afterwards be much more clever than you care to be.
# Argan (75 Bytes)
ddv_argan:::What! We understand how to discourse upon diseases when we have that dress?
# Beralde (118 Bytes)
ddv_beral:::Yes; you have only to hold forth; when you have a cap and gown, any stuff becomes learned, and all rubbish good sense.
# Toinette (74 Bytes)
ddv_toine:::Look you, Sir; a beard is something in itself; a beard is half the doctor.
# Cleante (34 Bytes)
ddv_clean:::Anyhow, I am ready for everything.
# Beralde (53 Bytes)
ddv_beral:::(TO ARGAN). Shall we have the thing done immediately?
# Argan (17 Bytes)
ddv_argan:::How, immediately?
# Beralde (19 Bytes)
ddv_beral:::Yes, in your house.
# Argan (12 Bytes)
ddv_argan:::In my house?
# Beralde (145 Bytes)
ddv_beral:::Yes, I know a body of physicians, friends of mine, who will come presently, and will perform the ceremony in your hall. It will cost you nothing.
# Argan (38 Bytes)
ddv_argan:::But what can I say, what can I answer?
# Beralde (151 Bytes)
ddv_beral:::You will be instructed in a few words, and they will give you in writing all you have to say. Go and dress yourself directly, and I will send for them.
# Argan (26 Bytes)
ddv_argan:::Very well; let it be done.
#
#
#
```

B.2 slave.pl

The Perl script slave.pl (based on the "HelloBot" [48]):

```
#!/usr/bin/perl -w

use Net::IRC;
#use strict;

######################################################################
# BEGIN
#

#use Time::HiRes qw(sleep);
use Getopt::Std;


# Programmargumente verarbeiten:
# Usage:
#   -c <Channel>  Kanalname
```

```
#   -h            Benutzung (Hilfe)
#   -i <Server>   IRC-Server
#   -n <Nick>     Nickname
#   -p <PortNr>   Portnummner
#
# getopts: vergleiche Perl Kochbuch Kapitel 15.1
getopts("c:hi:n:p:", \%args);

# ARGUMENT -h
if ( $args{h} ) {
  print "++ Benutzung:\n";
  print "++   -c <Channel>  Kanalname\n";
  print "++   -h            Benutzung (Hilfe)\n";
  print "++   -i <Server>   IRC-Server\n";
  print "++   -n <Nick>     Nickname\n";
  print "++   -p <PortNr>   Portnummer\n";
  exit;
}

# ARGUMENT -c
if ( $args{c} ) {
  $channelname = $args{c};
  print "++ Kanalname: ", $channelname, "\n";
}
else {
  print "WRONG USAGE!!! --> Argument -c falsch\n";
  exit;
}

# ARGUMENT -i
if ( $args{i} ) {
  $ircserver = $args{i};
  print "++ IRC-Server: ", $ircserver, "\n";
}
else {
  print "WRONG USAGE!!! --> Argument -i falsch\n";
  exit;
}

# ARGUMENT -n
if ( $args{n} ) {
  $thisbotname = $args{n};
  print "++ Nickname: ", $thisbotname, "\n";
}
else {
  print "WRONG USAGE!!! --> Argument -n falsch\n";
  exit;
}

# ARGUMENT -p
if ( $args{p} ) {
  $PortNumber = $args{p};
  print "++ Port: ", $PortNumber, "\n";
}
else {
  print "WRONG USAGE!!! --> Argument -p falsch\n";
  exit;
}

#
# END
#####################################################################
```

```
80
# create the IRC object
82 my $irc = new Net::IRC;

84 # Create a connection object.  You can have more than one "connection" ↩
    per
# IRC object, but we'll just be working with one.
86 my $conn = $irc->newconn(
        Server          => shift || $ircserver,
88      Port            => shift || $PortNumber,
        Nick            => $thisbotname,
90      Ircname         => 'I like to greet!',
        Username        => $thisbotname
92 );

94 # We're going to add this to the conn hash so we know what channel we
# want to operate in.
96 $conn->{channel} = shift || $channelname;

98 sub on_connect {
  #shift in our connection object that is passed automatically
100   my $conn = shift;

102   #when we connect, join our channel and greet it
  print "Joining channel $conn->{channel} ...";
104   $conn->join($conn->{channel});
  print " done.\n";
106   $conn->privmsg($conn->{channel}, "\*\*$thisbotname\*\* logged in");
  $conn->{connected} = 1;
108 }

110
######################################################################
112 # BEGIN
#
114
sub on_public {
116
  # on an event, we get connection object and event hash
118   my ($conn, $event) = @_;

120   # this is what was said in the event
  my $text = $event->{args}[0];
122   my $nick = $event->{nick};

124   if ($text =~ /\!ready/) {
      $conn->privmsg($conn->{channel}, "\*\*$thisbotname\*\* ready for ↩
          test");
126   }
  elsif ($text =~ /\!end/) {
128       $conn->privmsg($conn->{channel}, "\*\*$thisbotname\*\* leaves");
      sleep(2);
130       exit;
  }
132 }

134 # add event handlers
$conn->add_handler('public', \&on_public);
136
#
138 # END
```

```
######################################################################

# The end of MOTD (message of the day), numbered 376 signifies we've connect
$conn->add_handler('376', \&on_connect);

# start IRC
$irc->start();
```

B.3 ircsniffer.pl

```
#!/usr/bin/perl

#    ======================================================================
#    |                    SPALEWARE LICENSE (Revision 0.1)                |
#    |--------------------------------------------------------------------|
#    | This is a little script called "ircsniffer" and is                 |
#    | licensed under SPALEWARE. You may freely modify and distribute     |
#    | this script or parts of it. But you MUST keep the SPALWARE         |
#    | license in it!                                                     |
#    |                                                                    |
#    ======================================================================
#
# Author  : Pascal Gloor
# Date    : 24.10.2003
# Contact : spale@undernet.org
# Version : 1.1
#
# --> adapted by SR
# --> Oct 2003 - Apr 2004
#


# non printable chars are replaced by..
$npchar = '?';

$tcpdump = "/usr/sbin/tcpdump";

if ( !$tcpdump ) {
  print STDERR "FATAL ERROR: tcpdump not found\n";
  print STDERR "make sure tcpdump is in the PATH\n";
  exit 1;
}

if ( @ARGV ) {
  foreach (@ARGV) {
    $options .= sprintf(" %s",$_);
  }
  print "Starting decoder with '$tcpdump -lnx -s 1500$options'\n";
} else {
  print "Usage:     $0 <tcpdump_options>\n\n";
  print "Example 1: $0 -i eth0 tcp and dst port 6667\n";
  print "or\n";
  print "Example 2: $0 -r dumpfile.dump port 6661\n";
  print "\n";
  print "WARNING: you must ensure that you will only match IRC traffic,\n
      ";
```

```
  print "matching non-IRC traffic may break your terminal by displaying\n";
  print "non-printable chars!\n\n";
  exit;
}


$|=1;
open (STDIN,"$tcpdump -lnx -s 1500 $options |");
while (<STDIN>) {

  s/(\r|\n)+$//;  # \r = Carriage Return; \n = Linefeed; $ = matches end of line

  if ( /^[0-9]/ ) { # ^ = matches beginning of line
    # Trying to guess which is the server side.
    if ( $from =~ /\.(666[0-9]|7000|7777|8000|9000)$/ ) {
      $peer = $to;
      $dir  = 'server';
    }
    else {
      $peer = $from;
      $dir  = 'client';
    }

    undef $offset;
    $len = length($packet);

    # So, how long is this header?
    $ip_head_len = unpack("B8",pack('H2',substr($packet,$offset,2)));
    $ip_head_len =~ s/.*(....)$/$1/;
    $ip_head_len = hex(unpack("H1",pack("B4",$ip_head_len))) * 4;
    $offset += $ip_head_len * 2;

    # So, where does data start, must be somewhere in the TCP header .. ; p
    $tcp_head_len = hex(substr($packet,$offset+24,1)) * 4;
    $offset += $tcp_head_len * 2;
    undef $char;
    $lchar = -1;
    #$test = substr($packet,$offset,$len-$offset);
    $test = $packet;                                          # by SR --> also consider Ack's (empty packets)
    $temppacket = substr($packet, $offset, $len-$offset);  # by SR
    $temppacket =~ s/(00)+$//;                             # by SR
    if ( length($temppacket) == 0 ) {                      # by SR
      $len = $offset;                                      # by SR
    }                                                      # by SR

    if ( $test =~ /[1-9a-f]/i ) {
      #printf("%21s %s ",$peer,$dir);
      printf("%15s;;;%21s;;;%21s;;;%5s;;;", $time, $from, $to, $len/2);  # by SR

      for ( $pos = $offset; $pos < $len; $pos+=2 ) {

        #if ( $lchar eq 10 ) { printf("\n%21s %s ",$peer,$dir); }
        if ( $lchar eq 10 ) {
          printf("\n%15s;;;%21s;;;%21s;;;%5s;;;", $time, $from, $to, $len/2);
        }  # by SR
```

```
        $char = hex(substr($packet,$pos,2));
        if ( $char >= 32 ) {
          print chr($char);
        }
        elsif ( $char eq 27 ) {
          print "^";
        }
        elsif ( $char eq 10 ) {
          print "";
        }
        elsif ( $char eq 13 ) {
          print "";
        }
        elsif ( $char eq 3 ) {
          print "^[[7m^C^[[0m";
        }
        else {
          print "$npchar";
        }
        $lchar = $char;
      }
      print "\n";
    }
    ($time,$from,undef,$to)=split(/ /);
    $to =~ s/:$//;
    undef $packet;
  }
  else {
    s/^ +//;
    s/ //g;
    $packet .= $_;
  }
}
```

B.4 compare_dumps_to_flows.pl

```
#!/usr/bin/perl -w

##############################################################################
# used modules                                                               #
##############################################################################

use Getopt::Std;  # fuer 'getopts'-Funktion
use POSIX;        # fuer 'ceil'-Funktion
use Time::Local;  # fuer 'timelocal'-Funktion



##############################################################################
# process program options                                                    #
##############################################################################

# getopts:
# see Perl cookbook
########################################
getopts("d:hi:n:s:t:u:", \%args);

```

```
# ARGUMENT -h
################
if ( $args{h} ) {
  print "++ usage:\n";
  print "++   -d <path>       path to the directory containing netflow data\n";
  print "++   -h              usage of this program\n";
  print "++   -i <IP>         IP address to analyse\n";
  print "++   -n <path>       path to the program 'netflow_to_text' (only path)\n";
  print "++   -s <path>       path to the program 'ircsniffer.pl' (only path)\n";
  print "++   -t <file>       tcpdump file 1 \n";
  print "++   -u <file>       tcpdump file 2 \n";
  exit;
}

# ARGUMENT -d
################
if ( $args{d} ) {
  $PfadNetflowDaten = $args{d};
  printf ("++ path to the netflow data: %s\n", $PfadNetflowDaten);
}
else {
  print "WRONG USAGE!!! --> Argument -d wrong\n";
  exit;
}

# ARGUMENT -i
################
if ( $args{i} ) {
  $IP = $args{i};
  printf ("++ IP address: %s\n", $IP);
}
else {
  print "WRONG USAGE!!! --> Argument -i wrong\n";
  exit;
}

# ARGUMENT -n
################
if ( $args{n} ) {
  $PfadNetflowToText = $args{n};
  printf ("++ path to the program \'netflow_to_text\': %s\n", $PfadNetflowToText);
}
else {
  print "WRONG USAGE!!! --> Argument -n wrong\n";
  exit;
}

# ARGUMENT -s
################
if ( $args{s} ) {
  $PfadIrcsniffer = $args{s};
  printf ("++ path to the program \'ircsniffer.pl\': %s\n", $PfadIrcsniffer);
}
else {
  print "WRONG USAGE!!! --> Argument -s wrong\n";
  exit;
}
```

```
80
82  # ARGUMENT -t
    ###############
84  if ( $args{t} ) {
      $TcpdumpDatei1 = $args{t};
86    printf ("++ tcpdump file 1: %s\n", $TcpdumpDatei1);
    }
88  else {
      print "WRONG USAGE!!! --> Argument -t wrong\n";
90    exit;
    }
92
    # ARGUMENT -u
    ###############
94  if ( $args{u} ) {
      $TcpdumpDatei2 = $args{u};
96    printf ("++ tcpdump file 2: %s\n", $TcpdumpDatei2);
    }
98  else {
      print "WRONG USAGE!!! --> Argument -u wrong\n";
100   exit;
    }
102

104 ##########################################################################
    # initialisation                                                         #
106 ##########################################################################

108 @ListeNetflowDateien = ();
    @ListeNetflowDateien = getFilesToProcessInFolder($PfadNetflowDaten);
110 $AnzahlNetflowDateien = scalar(@ListeNetflowDateien);

112 $Protokoll       = "";
    $SourceIP        = "";
114 $DestinationIP   = "";
    $SourcePort      = "";
116 $DestinationPort = "";
    $AnzahlBytes     = 0.0;
118 $AnzahlPakete    = 0.0;
    $AnfangsZeit     = 0.0; # seconds.milliseconds
120 $AnfangsZeitMSEK = 0.0;
    $AnfangsZeitSEK  = 0.0;
122 $EndZeit         = 0.0; # seconds.milliseconds
    $EndZeitMSEK     = 0.0;
124 $EndZeitSEK      = 0.0;
    $Dauer           = 0.0; # seconds.milliseconds
126 $DauerMSEK       = 0.0;

128 # !!! don't forget to change accordingly !!!
    $Date          = "21.04.2004";
130
    $FormattedTime = "";
132 $Time = 0;

134 $Connection       = "";

136 %ConnectionsToClient      = ();
    %ConnectionsToServer      = ();
138 @ArrayConnectionsToClient = ();
    @ArrayConnectionsToServer = ();
140 $IndexConnectionsToClient = 0;
    $IndexConnectionsToServer = 0;
```

```
142
%ConnectionsToClient2      = ();
144 %ConnectionsToServer2      = ();
@ArrayConnectionsToClient2 = ();
146 @ArrayConnectionsToServer2 = ();
$IndexConnectionsToClient2 = 0;
148 $IndexConnectionsToServer2 = 0;

150 # grep expressions:
# we are only interested in flows and packets
152 # from a certain IP (193.110.95.1)
# and port (6661)
154 $GrepAusdruck1 = "'[ ]" . $IP . "[ ].*[ ]6661[ ]'";
print "++ Grep-Ausdruck 1: ", $GrepAusdruck1, "\n";
156
$GrepAusdruck2 = "'" . $IP . "'";
158 print "++ Grep-Ausdruck 2: ", $GrepAusdruck2, "\n";

160 $TcpdumpDateien[0] = $TcpdumpDatei1;
$TcpdumpDateien[1] = $TcpdumpDatei2;
162

164 ##########################################################################
# read netflow data and filter with grep:                                #
166 # every new connection is put into a hash                                #
# accordingly to the 'direction'                                         #
168 ##########################################################################

170  for ($i = 0; $i < $AnzahlNetflowDateien; $i++) {

172    $AktuelleZeit = getLokaleZeit();

174    $NetflowDataFileName = $ListeNetflowDateien[$i];
    printf("++ \n++ %-38s wird analysiert (Begin: %s)\n",  ↩
        $NetflowDataFileName, $AktuelleZeit);
176
    open(NETFLOW, "$PfadNetflowToText/netflow_to_text -D -f  ↩
        $PfadNetflowDaten/$NetflowDataFileName | grep -E $GrepAusdruck1 |" ↩
        ) or die "Fehler im Durchgang: $!\n";
178    while (defined($NetflowAusgabe = <NETFLOW>)) {

180      getEintraegeZeile($NetflowAusgabe);

182      $Connection = $SourceIP . " " . $SourcePort . " " . $DestinationIP ↩
           . " " . $DestinationPort;

184      # consider only flows between 16:00 and 17:00
      # 21.04.2004 16:00:00 = 1082556000
186      # 21.04.2004 17:00:00 = 1082559600
      # !!! don't forget to change accordingly !!!
188      if ( ($EndZeit >= 1082556000) && ($AnfangsZeit < 1082559600) ) {

190        if ( $AnfangsZeit < 1082556000 ) {
          # include overlapping flow in a linear way
192          $AnzahlPakete = $AnzahlPakete * (($EndZeit - 1082556000) / ( ↩
               $EndZeit - $AnfangsZeit));
          $AnzahlBytes  = $AnzahlBytes * (($EndZeit - 1082556000) / ( ↩
               $EndZeit - $AnfangsZeit));
194        }
        elsif ( $EndZeit > 1082559600 ) {
196          # include overlapping flow in a linear way
```

```
        $AnzahlPakete = $AnzahlPakete * ((1082559600 - $AnfangsZeit) / ( ↩
            $EndZeit - $AnfangsZeit));
198     $AnzahlBytes  = $AnzahlBytes * ((1082559600 - $AnfangsZeit) / ( ↩
            $EndZeit - $AnfangsZeit));
      }
200
      if ( $SourcePort == 6661 ) {
202     if (exists $ConnectionsToClient{$Connection}) {
          $ArrayConnectionsToClient[$ConnectionsToClient{$Connection ↩
              }][4] += $AnzahlPakete;
204       $ArrayConnectionsToClient[$ConnectionsToClient{$Connection ↩
              }][6] += $AnzahlBytes;
        }
206     else {
          $ConnectionsToClient{$Connection}  = $IndexConnectionsToClient;
208       $ArrayConnectionsToClient[$IndexConnectionsToClient][0] =  ↩
              $SourceIP;
          $ArrayConnectionsToClient[$IndexConnectionsToClient][1] =  ↩
              $SourcePort;
210       $ArrayConnectionsToClient[$IndexConnectionsToClient][2] =  ↩
              $DestinationIP;
          $ArrayConnectionsToClient[$IndexConnectionsToClient][3] =  ↩
              $DestinationPort;
212       $ArrayConnectionsToClient[$IndexConnectionsToClient][4] =  ↩
              $AnzahlPakete;
          $ArrayConnectionsToClient[$IndexConnectionsToClient][5] =  ↩
              $Connection;
214       $ArrayConnectionsToClient[$IndexConnectionsToClient][6] =  ↩
              $AnzahlBytes;
          $IndexConnectionsToClient++;
216       #print "++ ", $Connection, "\n";
        }
218   }
      else {
220     if (exists $ConnectionsToServer{$Connection}) {
          $ArrayConnectionsToServer[$ConnectionsToServer{$Connection ↩
              }][4] += $AnzahlPakete;
222       $ArrayConnectionsToServer[$ConnectionsToServer{$Connection ↩
              }][6] += $AnzahlBytes;
        }
224     else {
          $ConnectionsToServer{$Connection} = $IndexConnectionsToServer;
226       $ArrayConnectionsToServer[$IndexConnectionsToServer][0] =  ↩
              $SourceIP;
          $ArrayConnectionsToServer[$IndexConnectionsToServer][1] =  ↩
              $SourcePort;
228       $ArrayConnectionsToServer[$IndexConnectionsToServer][2] =  ↩
              $DestinationIP;
          $ArrayConnectionsToServer[$IndexConnectionsToServer][3] =  ↩
              $DestinationPort;
230       $ArrayConnectionsToServer[$IndexConnectionsToServer][4] =  ↩
              $AnzahlPakete;
          $ArrayConnectionsToServer[$IndexConnectionsToServer][5] =  ↩
              $Connection;
232       $ArrayConnectionsToServer[$IndexConnectionsToServer][6] =  ↩
              $AnzahlBytes;
          $IndexConnectionsToServer++;
234       #print "++ ", $Connection, "\n";
        }
236   }
    }
238 }
```

```
    close(NETFLOW);
 }
 print "++ \n";
 print "++ IndexConnectionsToClient: ", $IndexConnectionsToClient, "\n";
 print "++ IndexConnectionsToServer: ", $IndexConnectionsToServer, "\n";
 print "++ \n";


#########################################################################
# read from tcpdump files and filter packets:                           #
# every new connection is put into a hash                               #
# accordingly to the 'direction'                                        #
#########################################################################

for ($i = 0; $i < 2; $i++) {

  $TcpdumpDatei = $TcpdumpDateien[$i];
  print "Tcpdumpdatei: ", $TcpdumpDatei,"\n";

  open(DUMP, "$PfadIrcsniffer/ircsniffer.pl -r $TcpdumpDatei port 6661 | grep -E $GrepAusdruck2 |") or die "Fehler im Durchgang: $!\n";
  #open(DUMP, "/large2/analyses/tools/tcpdump -lnx -s 1500 -r $TcpdumpDatei port 6661 | grep -E $GrepAusdruck2 |") or die "Fehler im Durchgang: $!\n";
  while ( defined($SnifferAusgabe = <DUMP>) ) {

    getEintraegeZeile2($SnifferAusgabe);

    $Connection = $SourceIP . " " . $SourcePort . " " . $DestinationIP . " " . $DestinationPort;
    #print "Connection: ", $Connection,"\n";

    # consider only flows between 16:00 and 17:00
    # 21.04.2004 16:00:00 = 1082556000
    # 21.04.2004 17:00:00 = 1082559600
    # !!! don't forget to change accordingly !!!
    if ( ($Time >= 1082556000) && ($Time < 1082559600) ) {

      if ( $SourcePort==6661 ) {
        if (exists $ConnectionsToClient2{$Connection}) {
          $ArrayConnectionsToClient2[$ConnectionsToClient2{$Connection}][4] += 1;
          $ArrayConnectionsToClient2[$ConnectionsToClient2{$Connection}][6] += $AnzahlBytes;
        }
        else {
          $ConnectionsToClient2{$Connection} = $IndexConnectionsToClient2;
          $ArrayConnectionsToClient2[$IndexConnectionsToClient2][0] = $SourceIP;
          $ArrayConnectionsToClient2[$IndexConnectionsToClient2][1] = $SourcePort;
          $ArrayConnectionsToClient2[$IndexConnectionsToClient2][2] = $DestinationIP;
          $ArrayConnectionsToClient2[$IndexConnectionsToClient2][3] = $DestinationPort;
          $ArrayConnectionsToClient2[$IndexConnectionsToClient2][4] = 1;
          $ArrayConnectionsToClient2[$IndexConnectionsToClient2][5] = $Connection;
          $ArrayConnectionsToClient2[$IndexConnectionsToClient2][6] = $AnzahlBytes;
          $IndexConnectionsToClient2++;
```

```
288           #print "++ ", $Connection, "\n";
              }
290         }
            else {
292           if (exists $ConnectionsToServer2{$Connection}) {
                $ArrayConnectionsToServer2[$ConnectionsToServer2{$Connection
                    }][4] += 1;
294             $ArrayConnectionsToServer2[$ConnectionsToServer2{$Connection
                    }][6] += $AnzahlBytes;
              }
296           else {
                $ConnectionsToServer2{$Connection} = $IndexConnectionsToServer2
                    ;
298             $ArrayConnectionsToServer2[$IndexConnectionsToServer2][0] =
                    $SourceIP;
                $ArrayConnectionsToServer2[$IndexConnectionsToServer2][1] =
                    $SourcePort;
300             $ArrayConnectionsToServer2[$IndexConnectionsToServer2][2] =
                    $DestinationIP;
                $ArrayConnectionsToServer2[$IndexConnectionsToServer2][3] =
                    $DestinationPort;
302             $ArrayConnectionsToServer2[$IndexConnectionsToServer2][4] = 1;
                $ArrayConnectionsToServer2[$IndexConnectionsToServer2][5] =
                    $Connection;
304             $ArrayConnectionsToServer2[$IndexConnectionsToServer2][6] =
                    $AnzahlBytes;
                $IndexConnectionsToServer2++;
306             #print "++ ", $Connection, "\n";
              }
308         }

310     }
      }
312   close(DUMP);
}
314
print "++ \n";
316 print "++ IndexConnectionsToClient2: ", $IndexConnectionsToClient2, "\n";
print "++ IndexConnectionsToServer2: ", $IndexConnectionsToServer2, "\n";
318 print "++ \n";
print "++ \n";
320

322 ########################################################################
# compare the netflow data to the tcpdump data                           #
324 # and print them out                                                   #
########################################################################
326
$zf = 0.0;
328 $ProzentSummeToClient = 0.0;
$ProzentSummeToServer = 0.0;
330 print "++ Connections to client:\n";
for ($i = 0; $i < $IndexConnectionsToClient2; $i++) {
332   $Connection = $ArrayConnectionsToClient2[$i][5];

334   if (exists $ConnectionsToClient{$Connection}) {
        printf("++ %45s %9d %10.1f %14d %15.1f %10.2f\n", $Connection,
            $ArrayConnectionsToClient2[$ConnectionsToClient2{$Connection
            }][4], $ArrayConnectionsToClient[$ConnectionsToClient{$Connection
            }][4], $ArrayConnectionsToClient2[$ConnectionsToClient2{
            $Connection}][6], $ArrayConnectionsToClient[$ConnectionsToClient{
            $Connection}][6], ($ArrayConnectionsToClient[$ConnectionsToClient
```

```
        {$Connection}][4]/$ArrayConnectionsToClient2[ ↩
        $ConnectionsToClient2{$Connection}][4]));
    $ProzentSummeToClient += ($ArrayConnectionsToClient[ ↩
        $ConnectionsToClient{$Connection}][4]/$ArrayConnectionsToClient2[ ↩
        $ConnectionsToClient2{$Connection}][4]);
  }
  else {
    # packets not found in netflow data
    printf("++ %45s %9d %10.1f %14d %15.1f %10.2f\n", $Connection, ↩
        $ArrayConnectionsToClient2[$ConnectionsToClient2{$Connection ↩
        }][4], $zf, $ArrayConnectionsToClient2[$ConnectionsToClient2{ ↩
        $Connection}][6], $zf, $zf);
  }
}

print "++ \n";
print "++ Connections to server:\n";
for ($i = 0; $i < $IndexConnectionsToServer2; $i++) {
  $Connection = $ArrayConnectionsToServer2[$i][5];

  if (exists $ConnectionsToServer{$Connection}) {
    printf("++ %45s %9d %10.1f %14d %15.1f %10.2f\n", $Connection, ↩
        $ArrayConnectionsToServer2[$ConnectionsToServer2{$Connection ↩
        }][4], $ArrayConnectionsToServer[$ConnectionsToServer{$Connection ↩
        }][4], $ArrayConnectionsToServer2[$ConnectionsToServer2{ ↩
        $Connection}][6], $ArrayConnectionsToServer[$ConnectionsToServer{ ↩
        $Connection}][6], ($ArrayConnectionsToServer[$ConnectionsToServer ↩
        {$Connection}][4]/$ArrayConnectionsToServer2[ ↩
        $ConnectionsToServer2{$Connection}][4]));
    $ProzentSummeToServer += ($ArrayConnectionsToServer[ ↩
        $ConnectionsToServer{$Connection}][4]/$ArrayConnectionsToServer2[ ↩
        $ConnectionsToServer2{$Connection}][4]);
  }
  else {
    # packets not found in netflow data
    printf("++ %45s %9d %10.1f %14d %15.1f %10.2f\n", $Connection, ↩
        $ArrayConnectionsToServer2[$ConnectionsToServer2{$Connection ↩
        }][4], $zf, $ArrayConnectionsToServer2[$ConnectionsToServer2{ ↩
        $Connection}][6], $zf, $zf);
  }
}

# print out: 'packet-found-rate' (not 'packet-loss-rate'!)
print "++ Average: packets to client: ", 100 * ($ProzentSummeToClient/ ↩
    $IndexConnectionsToClient2), "\n";
print "++ Average: packets to server: ", 100 * ($ProzentSummeToServer/ ↩
    $IndexConnectionsToServer2), "\n";




################################################################################
# S U B R O U T I N E S                                                        #
################################################################################

# call: getLokaleZeit();
sub getLokaleZeit {
  my ($Se, $Mi, $St, $Ta, $Mo, $Ja) = (localtime)[0,1,2,3,4,5];
  $Mo = $Mo + 1;
  $Ja = $Ja + 1900;

```

```
  if ($Ta < 10) {
    $Ta = "0" . $Ta;
  }
  if ($Mo < 10) {
    $Mo = "0" . $Mo;
  }
  if ($St < 10) {
    $St = "0" . $St;
  }
  if ($Mi < 10) {
    $Mi = "0" . $Mi;
  }
  if ($Se < 10) {
    $Se = "0" . $Se;
  }

  # Format: "TAG.MONAT.JAHR STUNDEN:MINUTEN:SEKUNDEN"
  my $LokaleZeit = sprintf("%s.%s.%s %s:%s:%s", $Ta, $Mo, $Ja, $St, $Mi, $Se);
  return $LokaleZeit;
}


# call: epochseconds2dmyhms($zeit_in_sekunden);
sub epochseconds2dmyhms {
  # (Funktionsuebergabewert an localtime() muss in Sekunden sein!)
  #($Sekunden, $Minuten, $Stunden, $Tag, $Monat, $Jahr, $wday, $yday, $isdst) = localtime($LogStartZeitSEK);
  #$Monat = $Monat + 1;
  #$Jahr  = $Jahr + 1900;
  #print "TIME " . $Tag . "." . $Monat . "." . $Jahr . " " . $Stunden . ":" . $Minuten . ":" . $Sekunden . "\n";

  my ($ZeitSEK) = @_;

  my ($Sekunden, $Minuten, $Stunden, $Tag, $Monat, $Jahr, $wday, $yday, $isdst) = localtime($ZeitSEK);
  $isdst = $isdst;
  $yday  = $yday;
  $wday  = $wday;
  $Jahr  = $Jahr + 1900;
  $Monat = $Monat + 1;

  if ($Tag < 10) {
    $Tag = "0" . $Tag;
  }
  if ($Monat < 10) {
    $Monat = "0" . $Monat;
  }
  if ($Stunden < 10) {
    $Stunden = "0" . $Stunden;
  }
  if ($Minuten < 10) {
    $Minuten = "0" . $Minuten;
  }
  if ($Sekunden < 10) {
    $Sekunden = "0" . $Sekunden;
  }

  # Format: "TAG.MONAT.JAHR STUNDEN:MINUTEN:SEKUNDEN"
  my $ZeitFormatiert = $Tag . "." . $Monat . "." . $Jahr . " " . $Stunden . ":" . $Minuten . ":" . $Sekunden;
```

```
434 }
436
# call: dmyhms2epochseconds("01.12.2003 23:59:13");
438 sub dmyhms2epochseconds {
  my ($StringDMYHM) = @_;
440
  my $Second = 0;
442   my $Minute = 0;
  my $Hour   = 0;
444   my $Day    = 0;
  my $Month  = 0;
446   my $Year   = 0;

448   @TempArray = split(/\s+/, $StringDMYHM);
  @DateArray     = split(/\./, $TempArray[0]);
450   @TimeArray     = split(/:/, $TempArray[1]);

452   $Second = sprintf("%u", $TimeArray[2]);
  $Minute = sprintf("%u", $TimeArray[1]);
454   $Hour   = sprintf("%u", $TimeArray[0]);
  $Day    = sprintf("%u", $DateArray[0]);
456   $Month  = sprintf("%u", ($DateArray[1] - 1));
  $Year   = sprintf("%u", ($DateArray[2] - 1900));
458
  my $EpochSeconds = timelocal($Second, $Minute, $Hour, $Day, $Month, ↩
      $Year);
460
  #print "Input dmyhm:  ", $StringDMYHM, "\n";
462   #print "Output dmyhm: ", $EpochSeconds, "\n";

464   return $EpochSeconds;
}
466


468
# call: getLogTime($netflow_data_file_name);
470 sub getLogTime {
  my ($NetflowDatenDateiname) = @_;
472   my $TempString = $NetflowDatenDateiname;
  my $LogZeit = "";
474
  # test for '.dat.bz2'
476   if ($TempString =~ /\.dat\.bz2/) {
    $TempString =~ s/\.dat\.bz2/\.0/;
478     $TempString = substr($TempString, -12, 12);
    $LogZeit    = $TempString;
480     $LogZeit    = sprintf("%.1f", $LogZeit);
  }
482   elsif ($TempString =~ /\.dat\.ixt/) {
    $TempString =~ s/\.dat\.ixt/\.0/;
484     $TempString = substr($TempString, -12, 12);
    $LogZeit    = $TempString;
486     $LogZeit    = sprintf("%.1f", $LogZeit);
  }
488   elsif ($TempString =~ /\.dat/) {
    $TempString =~ s/\.dat/\.0/;
490     $TempString = substr($TempString, -12, 12);
    $LogZeit    = $TempString;
492     $LogZeit    = sprintf("%.1f", $LogZeit);
  }
494   else {
```

```
    print "Fehler in Erzeugung \'LogZeit\'\n";
  }

  return $LogZeit;
}



# call: getEintraegeZeile($zeile);
sub getEintraegeZeile {
  my ($Zeile) = @_;

  @EintraegeZeile = split(/\s+/, $Zeile);

  if ($EintraegeZeile[2] eq "pr") {
    $Protokoll = $EintraegeZeile[1];
  }
  else {
    print "split: ERROR!!!\n";
    exit;
  }

  if ($EintraegeZeile[4] eq "si") {
    $SourceIP = $EintraegeZeile[3];
  }
  else {
    print "split: ERROR!!!\n";
    exit;
  }

  if ($EintraegeZeile[6] eq "di") {
    $DestinationIP = $EintraegeZeile[5];
  }
  else {
    print "split: ERROR!!!\n";
    exit;
  }

  if ($EintraegeZeile[8] eq "sp") {
    $SourcePort = $EintraegeZeile[7];
  }
  else {
    print "split: ERROR!!!\n";
    exit;
  }

  if ($EintraegeZeile[10] eq "dp") {
    $DestinationPort = $EintraegeZeile[9];
  }
  else {
    print "split: ERROR!!!\n";
    exit;
  }

  if ($EintraegeZeile[12] eq "le") {
    $AnzahlBytes  = $EintraegeZeile[11];
    $AnzahlBytes  = sprintf("%.1f", $AnzahlBytes);
  }
  else {
    print "split: ERROR!!!\n";
    exit;
  }
```

```
558   if ($EintraegeZeile[14] eq "pk") {
        $AnzahlPakete = $EintraegeZeile[13];
560     $AnzahlPakete = sprintf("%.1f", $AnzahlPakete);
      }
562   else {
        print "split: ERROR!!!\n";
564     exit;
      }
566
      if ($EintraegeZeile[16] eq "st") {
568     $AnfangsZeit  = $EintraegeZeile[15]; # seconds.milliseconds
        $AnfangsZeit  = sprintf("%.4f", $AnfangsZeit);
570
        $AnfangsZeitMSEK = 1000 * $AnfangsZeit;
572     $AnfangsZeitMSEK = sprintf("%.1f", $AnfangsZeitMSEK);

574     $AnfangsZeitSEK = $AnfangsZeitMSEK / 1000;
      }
576   else {
        print "split: ERROR!!!\n";
578     exit;
      }
580
      if ($EintraegeZeile[18] eq "en") {
582     $EndZeit      = $EintraegeZeile[17]; # seconds.milliseconds
        $EndZeit      = sprintf("%.4f", $EndZeit);
584
        $EndZeitMSEK     = 1000 * $EndZeit;
586     $EndZeitMSEK     = sprintf("%.1f", $EndZeitMSEK);

588     $EndZeitSEK = $EndZeitMSEK / 1000;
      }
590   else {
        print "split: ERROR!!!\n";
592     exit;
      }
594
      if ($EintraegeZeile[20] eq "du") {
596     $Dauer        = $EintraegeZeile[19]; # seconds.milliseconds
        $Dauer        = sprintf("%.4f", $Dauer);
598
        #$DauerMSEK       = 1000 * $Dauer;
600     $DauerMSEK       = $EndZeitMSEK - $AnfangsZeitMSEK;
        $DauerMSEK       = sprintf("%.1f", $DauerMSEK);
602   }
      else {
604     print "split: ERROR!!!\n";
        exit;
606   }
}
608


610
# call: getFilesToProcessInFolder($directory);
612 sub getFilesToProcessInFolder {
      my ($PathToNetflowData) = @_;
614   my $TempFile   = "";
      my $TempString = "";
616   my $LogZeit    = 0.0;
      my @FilesToProcess;
618   my $DateAndTime = "";
```

```
  my $Counter91 = 0;
620 my $Counter93 = 0;
  my @Min91;
622 my @Min93;
  my @Max91;
624 my @Max93;
  my @TempStrings2;
626
  #$MinTime = sprintf("%.1f", $MinTime);
628 #$MaxTime = sprintf("%.1f", $MaxTime);

630 #print "MinTime: ", $MinTime, "\n";
  #print "MaxTime: ", $MaxTime, "\n";
632
  # get all files from directory
634 # see Perl cookbook
  print "++ Suche nach Netflow-Dateien:\n++ \n";
636
  $f = 0;
638 opendir(DIR, $PathToNetflowData) or die "Konnte Verzeichnis ↩
      $PathToNetflowData nicht oeffnen: $!\n";
  while ( defined($TempFile = readdir(DIR)) ) {
640
    $TempString = $TempFile;
642
    # consider only '*.dat' files
644 if ($TempString =~ /\.dat/) {
      $LogZeit = getLogTime($TempFile);
646   $FilesToProcess[$f] = $TempFile;
      $f = $f + 1;
648   printf ("++ %-38s gefunden", $TempFile);
      print " --> Netflow-Datei";
650   print " --> hinzugefuegt (im Bereich)\n";
      $DateAndTime = epochseconds2dmyhms($LogZeit);
652   print "++ Log-Startzeitpunkt: ", $LogZeit, " = ", $DateAndTime, "\n ↩
          ++ \n";

654   @TempStrings2 = split(/\_/, $TempFile);
      $TempStrings2[-2] = sprintf("%d", $TempStrings2[-2]);
656
      if ($TempString =~ /19991\_/) {
658     $Counter91 = $Counter91 + 1;
        #print "91: ", $TempStrings2[-2], "\n";
660     if ($Counter91 == 1) {
          $Min91[0] = $TempStrings2[-2];
662       $Min91[1] = $LogZeit;
          $Min91[2] = $DateAndTime;
664       $Max91[0] = $TempStrings2[-2];
          $Max91[1] = $LogZeit;
666       $Max91[2] = $DateAndTime;
        } else {
668       if ($TempStrings2[-2] < $Min91[0]) {
            $Min91[0] = $TempStrings2[-2];
670         $Min91[1] = $LogZeit;
            $Min91[2] = $DateAndTime;
672       }
          if ($TempStrings2[-2] > $Max91[0]) {
674         $Max91[0] = $TempStrings2[-2];
            $Max91[1] = $LogZeit;
676         $Max91[2] = $DateAndTime;
          }
678     }
```

```
      }
      if ($TempString =~ /19993\_/) {
        $Counter93 = $Counter93 + 1;
        #print "93: ", $TempStrings2[-2], "\n";
        if ($Counter93 == 1) {
          $Min93[0] = $TempStrings2[-2];
          $Min93[1] = $LogZeit;
          $Min93[2] = $DateAndTime;
          $Max93[0] = $TempStrings2[-2];
          $Max93[1] = $LogZeit;
          $Max93[2] = $DateAndTime;
        } else {
          if ($TempStrings2[-2] < $Min93[0]) {
            $Min93[0] = $TempStrings2[-2];
            $Min93[1] = $LogZeit;
            $Min93[2] = $DateAndTime;
          }
          if ($TempStrings2[-2] > $Max93[0]) {
            $Max93[0] = $TempStrings2[-2];
            $Max93[1] = $LogZeit;
            $Max93[2] = $DateAndTime;
          }
        }
      }
    } else {
      #print "\n";
    }
  }
  closedir(DIR);

  print "++ \n";

  $AnzahlNetflowDateien = $f;

  #print "Min91: ", $Min91[0], "\n";
  #print "Max91: ", $Max91[0], "\n";
  #print "Min93: ", $Min93[0], "\n";
  #print "Max93: ", $Max93[0], "\n";

  printf ("++ Anzahl 19991-Dateien:                %-10d\n", $Counter91);
  printf ("++ Anzahl 19993-Dateien:                %-10d\n", $Counter93);
  printf ("++ Anzahl zu verarbeitender Dateien:    %-10d\n",
      $AnzahlNetflowDateien);

#   if ($Counter91 != $Counter93) {
#     print "FEHLER!!! --> Anzahl 19991- und 19993-Dateien nicht gleich
    !";
#     exit;
#   } else {
#     print "++ --> Anzahl OK\n";
#   }

#   if ( ($Min91[0] != $Min93[0]) || ($Max91[0] != $Max93[0]) ) {
#     print "FEHLER!!! --> Zeitbereich 19991 und 19993 nicht gleich!";
#     exit;
#   } else {
#     print "++ --> Bereiche:\n++     ", $Min93[2], " bis ", $Max91
    [2], "\n";
#   }

  return @FilesToProcess;
}
```

```
738
# call: getEintraegeZeile2($zeile);
740 sub getEintraegeZeile2 {
  my ($Zeile) = @_;
742
  chomp($Zeile);
744
  @EintraegeZeile = split(/;;;\s*/, $Zeile);
746
  #print $EintraegeZeile[0], "\n";
748 #print $EintraegeZeile[1], "\n";
  #print $EintraegeZeile[2], "\n";
750 #print $EintraegeZeile[3], "\n";
  #print $EintraegeZeile[4], "\n";
752
  $Zeit = $EintraegeZeile[0];
754 $Zeit = substr($Zeit, 0, 8);
  #print "++ Zeit: ", $Zeit, "\n";
756 $FormattedTime = $Date . " " . $Zeit;

758 # WINTER TIME --> SUMMER TIME
  # TCPDUMP: UTC!!!
760 $Time = dmyhms2epochseconds($FormattedTime);
  #$Time = dmyhms2epochseconds($FormattedTime) - 3600;
762
  @TempArray = split(/\./, $EintraegeZeile[1]);
764 $SourceIP   = $TempArray[0] . "." . $TempArray[1] . "." . $TempArray ↩
      [2] . "." . $TempArray[3];
  $SourcePort = $TempArray[4];
766
  @TempArray = split(/\./, $EintraegeZeile[2]);
768 $DestinationIP   = $TempArray[0] . "." . $TempArray[1] . "." .  ↩
      $TempArray[2] . "." . $TempArray[3];
  $DestinationPort = $TempArray[4];
770
  $AnzahlBytes = $EintraegeZeile[3];
772 }
```

Appendix C

Configuration Files

C.1 IRC test server configuration files

To connect two Undernet IRC servers to each other (one server acts as Hub, the other as Leaf) use the following two (tested) configuration files. For more detailed explanation of the different "lines" please look at the documentation of the Undernet IRC server (ircu) which can be found on [20].

C.1.1 Configuration file for the Hub

```
################################################
# !!! Replace with the corresponding values !!! #
#                                              #
# pc-irc-hub.ethz.ch (A.B.C.D)                 #
# pc-irc-leaf.ethz.ch (E.F.G.H)                #
# xy.motd                                      #
################################################


# [M:line]
M:pc-irc-hub.ethz.ch:A.B.C.D:IRC-Testserver::97


# [A:line]
A:IRC-Testserver:Administered by xy:Visit Homepage


# [Y:lines]
# Servers (Hubs)
Y:90:90:300:1:9000000
# Servers (Leafs)
Y:80:90:300:0:9000000
# Clients
Y:1:90:0:400:160000


# [I:lines]
# allow everyone to connect
```

```
I:*::*::1

# [T:lines]
T:1:xy.motd

# [U:lines]

# [K:lines]

# [C:lines]
C:E.F.G.H:irchub:pc-irc-leaf.ethz.ch:4400:80

# [H:lines]

# [D:lines]

# [O:lines]
o:*@*.ch:irctest:oppy::1

# [P:lines]
# Clients
P:::C:6667
# Servers
P:::S:4400

# [Q:lines]

# [F:lines]
F:CRYPT_OPER_PASSWORD:FALSE
F:HIS_SERVERNAME:"pc-irc-hub.ethz.ch"
F:HIS_SERVERINFO:"The IRC-Testserver (Undernet Serversoftware)"
F:HIS_URLSERVERS:"http://www.undernet.org/servers.php"
F:NETWORK:"Testnetwork"
F:URL_CLIENTS:"ftp://ftp.undernet.org/pub/irc/clients"
```

C.1.2 Configuration file for the Leaf

```
##################################################
# !!! Replace with the corresponding values !!! #
#                                                #
# pc-irc-hub.ethz.ch (A.B.C.D)                   #
# pc-irc-leaf.ethz.ch (E.F.G.H)                  #
# xy.motd                                        #
##################################################

# [M:line]
M:pc-irc-leaf.ethz.ch:E.F.G.H:IRC-Testserver::66

```

```
14 # [A:line]
   A:IRC-Testserver:Administered by xy:Visit Homepage
16
18 # [Y:lines]
   # Servers (Hubs)
20 Y:90:90:300:1:9000000
   # Servers (Leafs)
22 Y:80:90:300:0:9000000
   # Clients
24 Y:1:90:0:400:160000

26
   # [I:lines]
28 # allow everyone to connect
   I:*::*::1
30
32 # [T:lines]
   T:1:xy.motd
34
36 # [U:lines]

38
   # [K:lines]
40
42 # [C:lines]
   C:A.B.C.D:irchub:pc-irc-hub.ethz.ch:4400:90
44
46 # [H:lines]

48
   # [D:lines]
50
52 # [O:lines]
   o:*@*.ch:irctest:oppy::1
54
56 # [P:lines]
   # Clients
58 P:::C:6667
   # Servers
60 P:::S:4400

62
   # [Q:lines]
64
66 # [F:lines]
   F:CRYPT_OPER_PASSWORD:FALSE
68 F:HIS_SERVERNAME:"pc-irc-leaf.ethz.ch"
   F:HIS_SERVERINFO:"The IRC-Testserver (Undernet Serversoftware)"
70 F:HIS_URLSERVERS:"http://www.undernet.org/servers.php"
   F:NETWORK:"Testnetwork"
72 F:URL_CLIENTS:"ftp://ftp.undernet.org/pub/irc/clients"
```

Bibliography

[1] *DDoSVax*.
`http://www.tik.ee.ethz.ch/~ddosvax/`.

[2] *LEO – Link Everything Online*.
`http://dict.leo.org/`.

[3] *SWITCH – The Swiss Education and Research Network*.
`http://www.switch.ch/`.

[4] J. Oikarinen and D. Reed. *Internet Relay Chat Protocol, RFC 1459*. May 1993.

[5] C. Kalt. *Internet Relay Chat: Architecture, RFC 2810*. April 2000.

[6] C. Kalt. *Internet Relay Chat: Channel Management, RFC 2811*. April 2000.

[7] C. Kalt. *Internet Relay Chat: Client Protocol, RFC 2812*. April 2000.

[8] C. Kalt. *Internet Relay Chat: Server Protocol, RFC 2813*. April 2000.

[9] IRChelp.org. *Internet Relay Chat (IRC) Help*, 2003.
`http://www.irchelp.org/`.

[10] Kai Seidler. *Internet Relay Chat – Eine möglichst kurze Einführung*, 2003.
`http://irc.fu-berlin.de/einfuehrung.html`.

[11] Daniel Stenberg. *History of IRC (Internet Relay Chat)*, September 2002.
`http://daniel.haxx.se/irchistory.html`.

[12] *The Undernet IRC network*.
`http://www.undernet.org`.

[13] Jarkko Oikarinen. *IRC History by Jarkko Oikarinen.* http://www.irc.org/history_docs/jarkko.html.

[14] Klaus Zeuge, Troy Rollo, Ben Mesander. *The Client-To-Client Protocol (CTCP)*, 1994. http://irchelp.org/irchelp/rfc/ctcpspec.html.

[15] Troy Rollo. *A description of the DCC protocol.* http://irchelp.org/irchelp/rfc/dccspec.html.

[16] Undernet User Committee. *CTCP and DCC Frequently Asked Questions*, 1997. http://www.user-com.undernet.org/documents/ctcpdcc.html.

[17] Michael Sandrof. *ircII project.* http://www.eterna.com.au/ircii/.

[18] Khaled Mardam-Bey. *mIRC – An Internet Relay Chat program.* http://www.mirc.com.

[19] The Mozilla Organization. *Mozilla home page.* http://www.mozilla.org.

[20] Undernet Coder Committee. *Undernet Ircd Development.* http://coder-com.undernet.org.

[21] Undernet Coder Committee. *Undernet P10 Protocol and Interface Specification.* http://cvs.undernet.org/viewcvs.py/undernet-ircu/ircu2.10/doc/p10.html?rev=1.6.

[22] EverythingIRC. *SearchIRC Network overview*, 2003. http://searchirc.com.

[23] Andreas Gelhausen, netsplit.de. *Summary of IRC networks*, 2003. http://irc.netsplit.de/networks/.

[24] T. Dierks, C. Allen. *The TLS Protocol, RFC 2246.* January 1999.

[25] Steve Gibson. *The Strange Tale of the Denial of Service Attacks Against GRC.COM*, May 2001. http://grc.com/dos/grcdos.htm.

[26] Kevin J. Houle (CERT/CC) and George M. Weaver (CERT/CC). *Trends in Denial of Service Attack Technology*, October 2001. http://www.cert.org/archive/pdf/DoS_trends.pdf.

[27] Rocky K. C. Chang. *Defending against Flooding-Based Distributed Denial-of-Service Attacks: A Tutorial*. October 2002.

[28] c't 2004, Heft 5. *Ferngesteuerte Spam-Armeen*. Heise Zeitschriften Verlag GmbH & Co, Februar 2004.

[29] Federal Computer Incident Response Center (FedCIRC). *BotNets: Detection and Mitigation*, February 2003.
`http://www.fedcirc.gov/library/documents/botNetsv32.doc`.

[30] Ramneek Puri. *Bots & Botnet: An Overview*, August 2003.
`http://www.sans.org/rr/papers/36/1299.pdf`.

[31] *Trojan list*.
`http://www.simovits.com/trojans/trojans.html`.

[32] "hypnosis". *Analysis of DDoS IRC bots*.
`http://www.netsys.com/library/papers/ddos-ircbot.txt`.

[33] Wong Natepetcharachai and Bo Zhang. *DDoS Attack Tools and Incidents*.
`http://www-scf.usc.edu/~bozhang/personal/PDFs/ddos.pdf`.

[34] David Moore, Geoffrey M. Voelker and Stefan Savage. *Inferring Internet Denial-of-Service Activity*, 2001.
`http://www.caida.org/outreach/papers/2001/BackScatter/usenixsecurity01.pdf`.

[35] Allen Householder (CERT/CC), Art Manion (CERT/CC), Linda Pesante (CERT/CC), George M. Weaver (CERT/CC). *Managing the Threat of Denial-of-Service Attacks*, October 2001.
`http://www.cert.org/archive/pdf/Managing_DoS.pdf`.

[36] James Etherton. *Internet Relay Chat – Pros, Cons and Those Pesky Bots*, April 2001.
`http://www.giac.org/practical/gsec/James_Etherton_GSEC.pdf`.

[37] Thomas Dübendorfer and Arno Wagner. *Past and Future Internet Disasters: DDoS attacks*, April 2003.
`http://www.tik.ee.ethz.ch/~ddosvax/talks/ddos_td.pdf`.

[38] Corey Merchant and Joe Stewart. *Detecting and Containing IRC-Controlled Trojans: When Firewalls, AV, and IDS Are Not Enough*, July 2002.
`http://www.securityfocus.com/infocus/1605`.

[39] Posted by "tj". *Dangers in BotNets?*, February 2003.
http://www.ndnn.org/blog/archives/2003_02.html.

[40] By "Curve". *Just What Is a Botnet?*, 2002.
http://zine.dal.net/previousissues/issue22/botnet.php.

[41] SwatIt.Org. *Bots, Drones, Zombies, Worms and other things that go bump in the night*, 2003.
http://swatit.org/bots/.

[42] SwatIt.Org. *Gallery of BotNet Observation Screen Captures*, 2003.
http://swatit.org/bots/gallery.html.

[43] *NetFlow Overview*.
http://www.cisco.com/en/US/products/sw/iosswrel/ps1831/products_configuration_guide_chapter09186a00800ca6cb.html.

[44] Philipp Jardas. *Bachelor's Thesis: P2P Filesharing Systems: Real World NetFlow Traffic Characterization*. February 2004.

[45] *tcpdump*.
http://www.tcpdump.org/.

[46] *Ethereal*.
http://www.ethereal.com/.

[47] *PlanetLab*.
http://www.planet-lab.org/.

[48] *A very simple bot: HelloBot*.
http://www.wholok.com/irc/.

[49] V. Jacobson, R. Braden, D. Borman. *TCP Extensions for High Performance, RFC 1323*. May 1992.

[50] *Project Gutenberg*.
http://www.gutenberg.org/.