Bot-Malware Data Acquisition System

Master Thesis

Network and Distributed Systems
Thomas Jacobsen (15gr1022)

Aalborg University
Department of Electronic Systems
Frederik Bajers Vej 7B
9220 Aalborg Øst
Denmark

Department of Electronic Systems
Fredrik Bajers Vej 7
9220 Aalborg Øst
http://es.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Bot-Malware Data Acquisiton System

Project Period:
Sprint 2015

Project Group:
NDS 15gr1022

Participant(s):

Thomas Jacobsen

Supervisor(s):
Jens Myrup Pedersen
Matija Stevanovic

Copies: 4

Page Numbers: 66

Appendix Page Numbers: 3

Date of Completion:
June 3, 2015

Abstract:

Botnets are one of the most serious security threats to Internet security today. The prerequisite to defeat botnets is to be able to detect them. Present detection systems use data acquisition systems, that are often limited by not being both scalable and cooperative.

This project identifies four main criteria and functionalities which, if simultaneously fulfilled, will improve present data acquisition systems. These criteria and functionalities are, that the system should be scalable, collaborative, not vulnerable to evasion techniques and independent of the C&C channel topology and protocol.

A proof of concept system is designed and implemented to prove, that a data acquisition system can be made, which improves present data acquisition systems. The proven system is better than present data acquisition systems by being both scalable and collaborative as well as less vulnerable to evasion.

Further, it is demonstrated how this data acquisition system can be used as part of a detection system to give good detection results.

Preface

This master thesis has been written by Thomas Jacobsen who is a 10'th semester student on the master Networks and Distributed Systems at the department of Electronic Systems, Aalborg University. The project has been devised in the period February, 2. to June, 3. - 2015.

Reading Instructions

This report is addressed to supervisors, students and others who are interested in the field of Networks and Distributed Systems.

The report contains references following the Harvard-method; [Surname,Year]. These references point to a bibliography in chapter 10. The bibliography contains information about the source like author, title, year of release etc.

Figures, tables, formulas, equations and code are numbered according to their location in the report as e.g. figure 4.1. The first number denotes that the figure is in chapter 4 and the second number the figure number in the specific chapter. Unless otherwise specified, the figures and tables in the report have been produced by the author of this report.

The digital version of the report is found on the attached CD. The CD also contains source code and other material made by the author as well as relevant literature used in the report. The appendix is placed in the back of the report.

Contents

Chapter 1

Introduction

In this chapter problems with botnets and present detection systems are introduced. This leads to the initiating problem, which sets the starting point for this project.

Botnets are one of the most serious security threads to Internet security today [Shin et al., 2013], [Gu et al., 2008b], [Gu et al., 2008a], [Stevanovic and Pedersen, 2014]. Latest reports estimates that 1.9 million computers are infected and are active in a botnet in 2014 [Symantec, 2015]. Botnets are organized distributed networks of bot-malware which enables them to organize malicious activities. A botnet consists mainly of bots or zombies, a bot-master and a channel for communication called the Command and Control (C&C) channel. The bot-master is dependent on the C&C channel to command the bots to perform the desired actions. By their organization they are capable of cooperating to spread, to perform harmful attacks on large IT systems or to collect malicious data. It has been reported [Symantec, 2015] that botnets are growing as more and more devices are being connected to the Internet. Further, botnets are evolving [Matija Stevanovic, 2013] and migrating to new platforms e.g mobile devices.

The prerequisite to defeat botnets is to be able to detect them. As botnets are evolving the ability to detect the botnets should stay updated to be able to cope with the new and more complex botnets. Systems of detecting botnets is referred to as detection systems. Detection system performance depends entirely on the data it can acquire. The data is used by a detector to prepare a model and makes decisions based on this model. Together data acquisition, storage and a detector form a detection system as illustrated in figure 1.1.

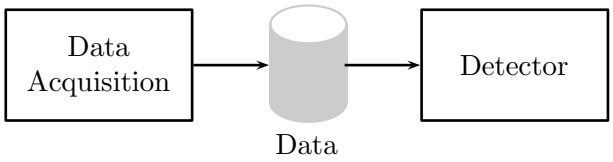

Figure 1.1: Detection system definition.

Current detection systems are based on either host, network or hybrid, depending on where the detection system is applied and where the data is acquired [Matija Stevanovic, 2013] [Stevanovic et al., 2012]. Host based detection systems are using data acquired on the host which can be from external interfaces, operating system specific data and similar. Host based detection systems are therefore usually limited to the single host which means it is not capable of cooperating or comparing data from other detection systems. This could be a disadvantage against distributed botnets. Network based detectors are usually deployed in network edges, giving them the opportunity to monitor network traffic for all hosts in the

network. However, this limits them to handle the traffic on the network edge. Having to handle data parsing a single point, implies that the network based detection system is not scalable. Hybrid detection systems are the combination of both host and network meaning they combine detection systems located each place. It still means they are not scalable due to the network detection part.

Present detection systems use data acquisition systems which are based on either host, network or hybrid. This limits the system to be either cooperative or scalable. Both could be key properties in the fight against a distributed and organized opponent. This leads to the initiating problem:

"How to improve data acquisition systems for bot-malware detection systems?"

This initiating problem will initiate and form the frame of the problem analysis.

Chapter 2

Problem analysis

In this chapter it is investigated, how present data acquisition systems, for bot-malware detection, can be improved. To do this, it is needed to investigate the challenges caused by botnets. Present data acquisition systems are investigated and compared to identify possible improvements. This leads to definition of the functionalities and characteristics needed for improving present data acquisition systems, i.e. problem definition.

2.1 Botnet

Botnets, as described in the introduction chapter 1, are defined as a distributed network of bots controlled by a bot-master. They communicate with their C&C channel which can utilize various protocols and utilize various topologies [Matija Stevanovic, 2013].

2.1.1 Topologies

Earlier botnets were utilizing simpler topologies and were centralized [Stevanovic et al., 2012]. These are simple to deploy as all communication is through a single point which also makes it simple to take down. The centralized topology is illustrated in figure 2.1.An example a of botnet, which used centralized topologies is the botnet Agobot which used IRC protocols and an IRC server for its C&C channel. This botnet has been taken down and is no longer being used [Stevanovic et al., 2012].

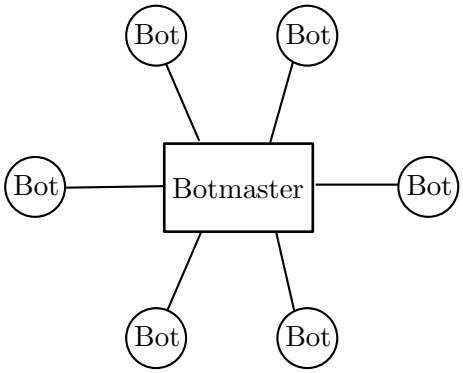

Figure 2.1: Centralized topology.

The more advanced topology is the decentralized topology. This does not have a single point of failure, but is most likely harder to deploy and maintain as it requires some routing. The advantages of a decentralized structure is its resilience. The network can not be taken down down by disabling a single bot and a bot-master would still be capable of contacting the bots in the network from any of the bots within the network. For a network with this topology to exist P2P protocols would most likely be used.

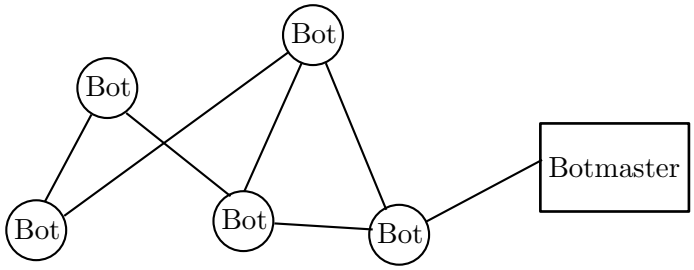

Figure 2.2: Decentralized topology.

As botnets evolved so did the used topologies of the C&C channel. The focus was moved to combining the advantages of centralized and decentralized topologies in the search to make the network more resilient. The combination of centralized and decentralized topology is called hybrid topology. Hybrid topologies consists of multiple layers which can be either centralized or decentralized. Some of these layers can act as proxy layers to the next ones and provide a interface between the network of working bots and botmasters. Working bots would be those which perform the actual malicious activities. A hybrid topology consisting of a decentralized proxy layer of bots and a layer of working bots connected centralized is illustrated in figure 2.3

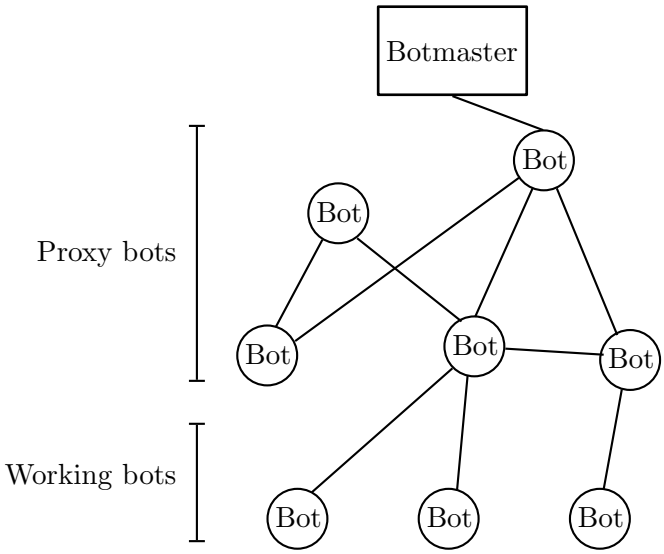

Figure 2.3: Hybrid topology.

Conclusion It can be concluded that bot-malware is capable of utilizing different combinations of centralized and decentralized topologies into hybrid topologies. The data acquisition system can for that reason not assume that the botnet are utilizing a specific topology. An assumption on topologies could for example be used by the data acquisition to determine where to place the data acquisition probes and their topology. The botnet C&C channel is its external communication channel meaning that the presence of bot-malware dan be detected by the C&C channel.

2.1.2 Protocol

The protocols used in the C&C channel is important to consider in the attempt to detect botnets or bot-malware as the protocol defines how the C&C channel works. It seems that botnets utilize a range of protocols for their C&C channel according to [Matija Stevanovic, 2013]. The earliest botnets which where centralized used usually the IRC or HTTP protocols. Later, with the move to decentralized topologies, botnets moved to P2P protocols [Stevanovic et al., 2012]. Custom protocols was later seen being used, which gave the opportunity for botnets to incorporate the exact functionality they wanted. Custom protocols are harder to interpret and detect as they are non standard. Further, the C&C channel can use standard protocols which are not designed to message exchange (like DNS) or it can use benign public available services like chatting services (like twitter). There is no evidence that a specific C&C channel protocol will be used in the future, it seems the used protocols are very diverse.

Conclusion It will have to be concluded that the choise of protocol used by the C&C channel is very diverse and ranges from custom to standard and misused standard protocols. The data acquisition system cannot assume that any particular protocol is used by the botnets. The data acquisition system should for that reason aim to acquire protocol independent data.

2.1.3 Evasion

The threat from botnets has naturally lead to a consequence which is the increasing interest in defeating and study them. The consequence of this is that botnets evolve evasion techniques to avoid being detected and analyzed.

According to [Matija Stevanovic, 2013] botnet developers have developed a lot on bot-malware and evasion techniques. There is evidence that bot-malware is actively attempting to detect whether there is a risk that it is being monitored. If it detects that it is being monitored it might delete itself, suppress external activities, change to random behavior or attempt to mimic benign application behavior [Symantec, 2014]. The bot-malware can on the host check for the presense of “unwanted” programs or check whether other programs are hooking the same functions in the operating system as the bot-malware. The bot-malware can also check whether it is being running in a virtualized environment [Symantec, 2014] or experiences limited emulated internet access. If the bot changes behavior there is a risk that the detector is trained with bad data, in the sense that it will be trained to look for evading bot-malware and could therefore be less good at detecting normally behaving bot-malware. To confuse signature based detectors it might also use polymorphism and metamorphism to create a slightly different version of itself or make sure its binary is different [Matija Stevanovic, 2013].

Conclusion It can be concluded that the data acquisition system should be careful about acquiring data which can be evaded.The data acquisition should not rely on specific patterns in packet payload or signatures, as these can be encrypted or changed with polymorphism or metamorphism. Further the data acquisition system should be careful about how which data it acquired on the host, as it increases the risk that the bot-malware is detecting that it is being monitored. Data acquisition systems for bot-malware detection systems have to take into account how botnets behave. This has been done in terms

of analyzing the topologies botnets can utilize, which protocols the C&C channel utilizes and how the bot-malware is capable of utilizing evasion techniques. The data acquisition system will also have to adjust to the detectors needs which will be analyzed in the following section.

2.2 Detection methods

The detector in the detection system is also setting various requirements for the data acquisition system. The author of [Matija Stevanovic, 2013] defines three types of detection methods which are:

- Signature.
- Anomaly.
- Machine Learning.

Signature based detection is relying on identifying signatures in payload, binaries or similar. Signature based detection can be very accurate in detecting bot-malware, but is also very vulnerable to evasion techniques. Bot-malware can alter the pattern by deploying encryption or change its binary patterns by poly/metamorphism. Further, signature based detection will have very small change of detecting zero-day bot-malware since it is unknown for the detector.

Anomaly based detection relies on a model determining what is to be considered normal. Anomaly detection is used by several Intrusion Detection Systems (IDS) to identify malicious attacks. It is preferable to detect bot-malware before it initiate attacks to avoid the attacks. Abnormal based detection might suffer in the situations where bot-malware is using C&C channel and communication patterns which is attempting to have a normal appearance.

Machine learning based detection has gained a lot of interest in recent years [Matija Stevanovic, 2013]. One of the reasons is that machine learning based detection is capable of identifying underlying patterns, which might otherwise be hidden [Matija Stevanovic, 2013]. It is therefore better suitable for bot-malware detection as the patterns from bot-malware might not be easy to determine due to the diversity of protocols and behaviors. Machine learning based detection relies on data mining, meaning it needs a comprehensive large datasets to properly extracts the underlying patterns. Machine learning techniques which are supervised needs data for:

- Traning (detecting the underlying patterns).
- Testing (verity that the detection accuracy is sufficient using the found patterns).

A perfect testing and training set would include a representative set of data and features which are as discriminative as possible. These would in the case of detecting bot-malware, with supervised machine learning detectors, require a set of malicious data and a set of benign data.

A representative set of benign data depends on a large range of factors such as social environment, country and age just to name a few. To acquire representative set of benign data the data acquisition system would have to acquire data in the real world, as it would be difficult to gather a set of representative data in a lab.

A representative set of bot-malware data depends on the availability of bot-malware samples, which is often hard get. Especially it is hard to get samples which also are live in the real world. Found bot-malware samples could be analyzed either dynamically or statically. A dynamic analysis involves executing the bot-malware sample and static analysis is about looking into the binaries and identify their content. In the real world detection of bot-malware binaries is hard to do as they might quickly be deleted upon installation or otherwise hidden on the host. Dynamic analysis attempts to capture live characteristics. To conduct dynamic analysis it is needed to contain the samples to ensure they do not harm external systems, spread or perform other malicious activities. Acquiring data from bot-malware samples in containment would usually be done by conducting experiments. In experiments the data acquisition system would have to interact with other parts of the experiment to ensure everything is done after a certain schedule. The reliability of the data acquisition system would also be a concern as an experiment could fail if the data is not acquired.

Conclusion It can be concluded that in the area of detectors for bot-malware, machine learning based detectors are gaining ground. It is better than traditional detection methods such as signature or abnormal, to identify the hidden pattern produced by the behavior of bot-malware. Machine learning does however need both benign and malicious data. The data acquisition system should be capable of mining data or acquiring data in real world environment and in contained environment.

2.3 Present data acquisition systems

In this section characteristics of present data acquisition systems is discussed in order to identify what could be improved. These judgments will be done with the knowledge about botnets from 2.1 and detection from 2.2. The data acquisition system will be compared on the terms of scalability, collaborability, vulnerability to evasion and independence of protocol and topology.

The data acquisition systems, which will be investigated and compared, are all a part of a detection system, which means it can be difficult to isolate the data acquisition system and its contribution. To better discuss the data acquisition part each data acquisition system is visualized with the principles from figure 2.4 and 2.5.

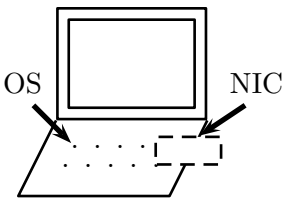

Figure 2.4: Host probe locations.

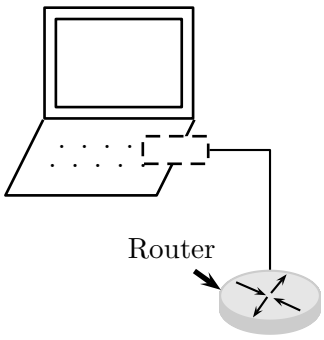

Figure 2.5: Network probe location.

Host based data acquisition systems can acquire data from either the operating system or from external network interfaces also denoted Network Interface Card (NIC). Network based data acquisition can acquire data from a router at a network edge. Hybrid is a combination of these. The symbols used for host data acquisition can be seen in figure 2.4 and the symbols used in network data acquisition can be seen in figure 2.5. An arrow in the figure indicates an active probe where data is acquired. In the next section the following present data acquisition systems will be discussed:

- Botminer [Gu et al., 2008a].
- Correlating log files [Masud et al., 2008].
- Suspicious groups [Zeng et al., 2010].
- EFFORT [Shin et al., 2013].

The first and one of the earlier data acquisitions systems are presented is Botminer.

2.3.1 Botminer

Botminer [Gu et al., 2008a] is a detection system which utilizes a network based data acquisition system. The location of the probe is at a router edge as is illustrated in figure 2.6. It uses one of the first open source botnet detection systems Bothunter [Gu et al., 2007] which uses anomaly detection to detect abnormal activities. Bothunter is also used in [Gu et al., 2008b]. Botminer targets to detect infected groups of computers in the network and therefore assumes that multiple computers will be infected with the same bot-malware. It acquires statistics from the computers generated network flows such as its length, packet size and byte rate, which means it is vulnerable to time evasion techniques. Further it uses Bothunter to acquire port scanning rates. Botminer gets information about DNS lookups as well. Non of the data acquired assumes that the botnets uses specific protocols or utilizes a specific topology.

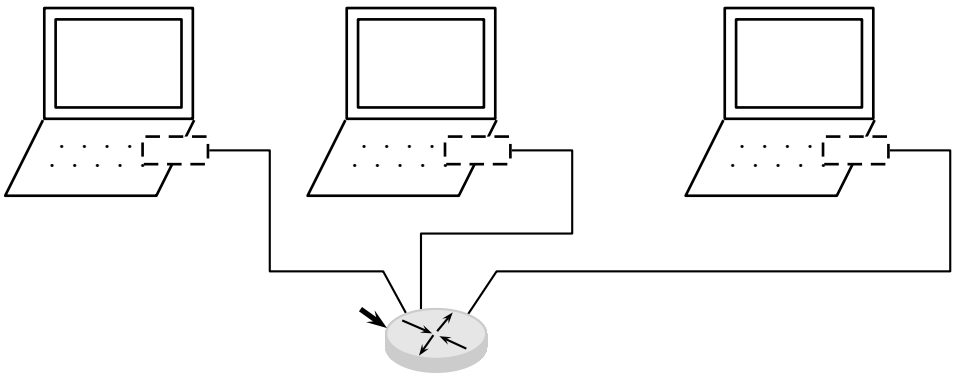

Figure 2.6: Botminer probe overview.

Verdict The verdict of botminer on earlier defined criteria is sumarized in the following table 2.1. Notice that '-' is used for a negative finding about Botminer, '+' is used for a positive findings about Botminer and '*' for a inconclusive verdict. This table layout is used throughout this report.

Criteria	Verdict	Comment
Scalability	-	Botminer is using a network based data acquisition, which means that the system is not scalable.
Collaborability	*	Botminer uses data from the hosts in the network.
Vulnerability to evasion	-	The detector depends on data which can be evaded by time evasion techniques.
Independence of protocol and topology	+	It is not assuming any topology of the botnet and is protocol independent.

Table 2.1: Botminer verdict.

2.3.2 Correlating log files

The detection system developed by [Masud et al., 2008] is using a host based data acquisition system. It acquires data from both network interfaces and the OS on the host. There is no connection between the hosts, they are independent of each other. This is illustrated in figure 2.7. Information about creation of flows and process are correlated. The assumption used for the detection, is that bot-malware processes are automated processes, which have faster response time than humans. Processes, which are created within a threshold of a flows creation, are marked as bot processes.

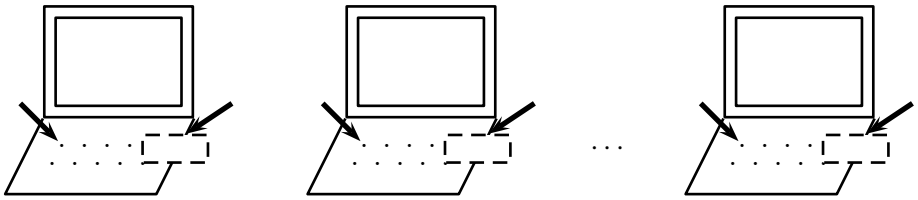

Figure 2.7: Correlating log files probe overview.

The acquired data is not dependent on a specific protocol or topology as it is mainly creation times of network flows and processes which are acquired. As the features are time dependent bot-malware could evade detection by mimicking human reaction times. The system is scalable as it consists of independent hosts, there is not bottleneck. The computers do however not cooperate as they function as individual detection systems.

Verdict The verdict on correlating log files is summarized in the table 2.2.

Criteria	Verdict	Comment
Scalability	+	The host are individual the system is scalable.
Collaborability	-	As the hosts are individual and do not collaborate.
Vulnerability to evasion	-	The detection system is vulnerable to time evasion.
Independence of protocol and topology	+	Not dependent on protocol or topology.

Table 2.2: Correlating log files verdict.

2.3.3 Suspicious groups

The work presented in [Zeng et al., 2010] is a hybrid detection system meaning that it acquires data from both hosts and network. It uses the network acquired data to identify groups of suspicious hosts and the data acquired on the host to decide whether the host is infected. The locations of the probes is illustrated in figure 2.8.

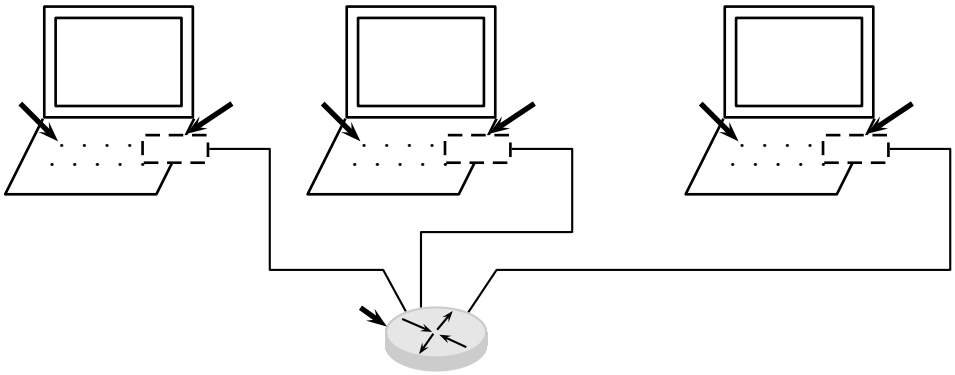

Figure 2.8: Suspicious groups probe overview.

To acquire data on the host, operating system hooks are used to identify whether, for example, files in critical system folders or registry keys are being created, deleted or modified. The network probe acquires data to create statistics of flows and contacted external systems such as the quantity of unique IPs, SMTP flows and suspicious ports.

The use of a network data acquisition makes the system non scalable. A possible improvement to the system would be to move the network data acquisition to the individual hosts, which would provide the same data to the detector and make the system scalable. As the system uses several operating system hooks, it increases the risk of being detected by bot-malware. However, by combining information from two different domains, here host and network, are referred to as collaborative and is expected to increase detection accuracy [Stevanovic et al., 2012].

Verdict The verdict of the data acquisition used in the detection system presented in [Zeng et al., 2010] is summarized in table 2.3.

Criteria	Verdict	Comment
Scalability	-	The data acquisition system is not scalable.
Collaborability	+	The data acquisition acquires data from both network and host data.
Vulnerability to evasion	*	The system could be less likely to be detected if it used less operating system hooks to acquire data.
Independence of protocol and topology	+	Not dependent on protocol or topology.

Table 2.3: Suspicious groups verdict.

2.3.4 EFFORT

EFFORT is a detection system presented in [Shin et al., 2013]. This detection system is using a host based data acquisition system which acquires data from both the external network interfaces and from the operating system. Probe location is therefore similar to the one illustrated in figure 2.7. The idea of the detection system is to use data acquired from different data domains and combine these in sub detection systems. One sub detection domain is regarding user interaction and another is regarding process creations on the host. A third is collecting network flow data. There is a total of five sub detection systems in the detection system. The system acquires data from multiple data domains which makes the system collaborative. It does however not acquire data from multiple hosts. Since the total detection system is only running on a single computer, it can be deployed on as many as wanted. This makes the system scalable. However as it acquires a lot of information on the host it increases the change of bot-malware detecting it. The data acquisition system is independent of both topology and protocol.

Verdict The verdict of the data acquisition system presented in [Shin et al., 2013] detection system is summarized in table 2.4.

Criteria	Verdict	Comment
Scalability	+	The system is scalable.
Collaborability	*	The system is collaborative in the sense of its use of multiple data domains.
Vulnerability to evasion	-	The system is likely to be detected by bot-malware and therefore also in risk of being evaded.
Independence of protocol and topology	+	Not dependent on protocol or topology.

Table 2.4: EFFORT verdict.

2.4 Problem definition

In this section the problem analysis is summarized. This leads to a problem definition of the functionalities and characteristics needed for improving present data acquisition systems.

From the botnet section 2.1 it was found that data acquisition systems used for modern botnets should not be dependent on either topology or protocol. Further it should be careful not to acquire data with methods which increases the likelihood of bot-malware detects its presence. The data acquisition system should also be careful not to acquire data to the detection system which can be evaded.

In section 2.2 it was found that data acquisition system should be capable of mining data for both benign data and malicious data. This includes that the data acquisition system should function on a large set of real world computers to acquire representative data. To acquire malicious data the data acquisition system should be capable of being incorporated into a contained environment where it would take parts in experiments. It is found as well that as botnets evolves and merges into new platforms, the data

acquisition systems which are host based should be able to migrate with it to the new platforms.

In section 2.3 four present data acquisition systems where discussed. The verdict on each is merged into table 2.5 for overview and comparison.

Criteria	Botminer	Correlating log files	Suspicious groups	EFFORT
Scalability	-	+	-	+
Collaborability	*	-	+	*
Vulnerability to evasion	-	-	*	-
Independence of protocol and topology	+	+	+	+

Table 2.5: Present data acquisition systems comparison.

The four main characteristics; scalability, collaborability, vulnerable for evasion and independence of protocol and topology are all wanted characteristics of a data acquisition. It can be noticed that those data acquisition systems which are scalable are also host based as they consists of individual hosts. All network data acquisition systems are note scalable. Only suspicious groups is collaborative as it combines information from various hosts and uses data from both host and network domains. EFFORT is not truly collaborative as it cannot combine data from different hosts. A combination of these two can be achieved by making the data acquisition system distributed and collect the data to a centralized location.

Nearly all of the discussed present data acquisition systems are vulnerable to evasion techniques, but none are dependent on a specific protocol or topology for the C&C channel. All of the discussed present data acquisition are independent on protocol and topology. It can be concluded that a data acquisition system which simultaneous fulfills the characteristics and functionalities listed below, will improve data acquisition systems.

- Be scalable.
- Be collaborative meaning it can acquire data from multiple data domains and use multiple devices to collaborate on the detection of botnets.
- Avoid acquiring data which bot-malware can evade and manipulate.
- Acquire topology and protocol independent data.

Further it is found that the following functionalities and characteristics are desirable to fulfill.

- Be distributed and acquire data to a central location.
- Support multiple platforms.
- Function on several real world hosts.
- Be usable to acquire data in experiments in contained environment.

This problem definition will be form the basis of the requirements for the system to be designed and implemented.

Chapter 3

Requirements

In this chapter requirements for an improved data acquisition system will be derived. The requirements for the system will be based on the problem definition. First the system overview will be defined and presented followed by a limitation in the project scope by defining the system as a proof of concept. Then the requirements for the proof of concept system and its subsystems will be defined. Finally the chapter will define how the requirements of the system will be verified.

3.1 System overview

In this section the system overview will be presented. The system structure will be distributed as deriveved in the problem definition section 2.4 to make the system both scalable and collaborative.

The system to be designed and implemented in this project will be a distributed data acquisition system with host based clients which are acquiring both network and host data. The acquired data is collected to a centralized server where a detector can access the data. The layout is sketched in figure 3.1 showing the probes on the host and the centralized server with a database where all acquired data is stored.

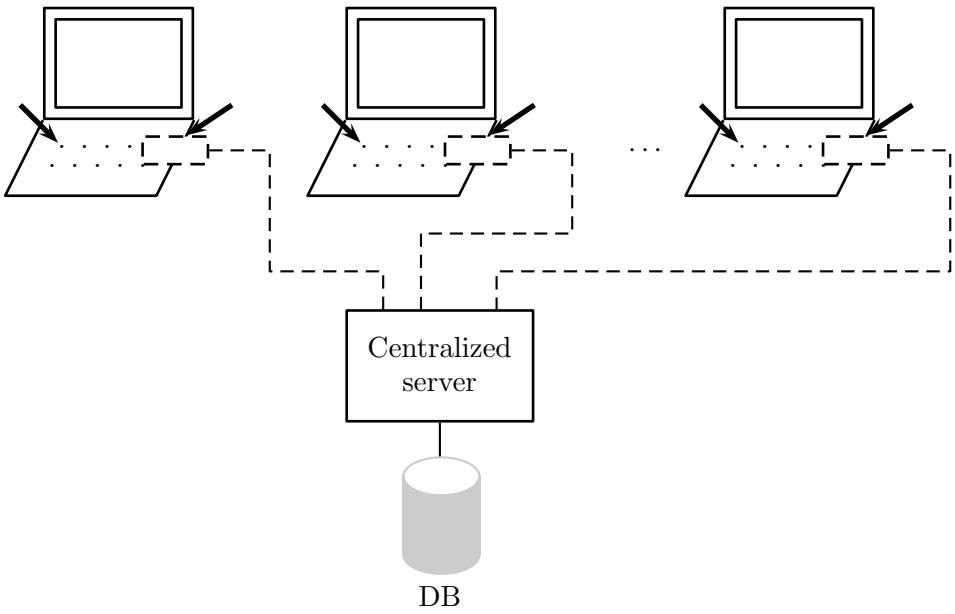

Figure 3.1: Data acquisition system parts overview.

In the following section the scope of the project will be limited as the system will be defined as a proof of concept. Following the proof of concept section the requirements for the system and its two parts, client

and server will be derived.

3.2 Proof of concept

The system to be designed and implemented will be defined as proof of concept. The system will aim at proving a data acquisition system which fulfills the four main characteristics scalability, collaborability, not vulnerable to evasion techniques and independence of C&C channel topology and protocol can exist.

The concept will be proved by designing and implementing the data acquisition system which fulfills the requirements derived in this chapter and fulfills the four main characteristics. The proof of concept will prioritize fulfilling the four main characteristics, but will also attempt to fulfill as many of the additional characteristics and functionalities form the problem definition.

As this system is a data acquisition system it have to be defined what data it should acquire for the system to be designed and implemented. The discussed present data acquisition systems acquired data to calculate network flow statistics. Some additionally acquired DNS related information. DNS can be used actively by botnets to maintain their network connectivity. The use of DNS by botnets be seen in a lot of registered IPs to the same domain for redundancy and they will have a short time to live to protect the anonymity of the hosts [Matija Stevanovic, 2013]. TCP flags can be used to indicate malicious activity such as port scanning. Similar the sequence and acknowledgement numbers for TCP packets will be acquired. These can be used to estimate packet loss and round trip time, which has not been used as features in present data acquisition systems as far of the authors knowledge. The proof of concept data acquisition system will limit to acquire only IPv4 traffic and will not acquire local area network traffic. As it is wanted that the data acquisition system should enable the detection system to be collaborative, the data acquisition system should acquire host based data as well. This data will be limited to process names of the process which has created network flow acquired. Below is the data which will be acquired by the proof of concept data acquisition listed.

- Packet length.
- Packet header length.
- Packet direction.
- Packet timestamp.
- TCP flags indication.
- TCP sequence and acknowledgement numbers.
- DNS A response time to live.
- DNS A response quantity of answers.
- DNS domain level depth.

To demonstrate the data acquisition system and to get a feeling of what can be achieved using this data acquisition system in a detection system, it will be used in a detection system in chapter 7.

3.3 System requirements

In this section requirements for the system will be defined. Chosen and identified for the system is three from the problem definition which is; scalability, benign and malicious data acquisition and open source.

One of the main requirements found mentioned in the problem definition in section 2.4 as an improvements for present data acquisition systems, are that the system should be scalable. Scalability though have to be defined. In this project scalability, as it is referred to earlier in the project, is defined from the system structure and the presence of bottlenecks. In order to test the system scalability, scalability will be defined as a quantity of clients, the server is able to concurrently serve. The scalability will therefore depend on the quantity of concurrent clients, and the characteristics of the clients. In order to test the scalability these have to be defined. Since this system is a proof of concept the scalability of the system will be defined such that the server is capable of serving 3 clients in a worst case scenario. This number is chosen by empirical measurements. The clients worst case scenario is defined as a situation where they are experiencing a file transfer with a mean data rate between $1.2\,\frac{\text{MB}}{\text{s}}$ and $1.8\,\frac{\text{MB}}{\text{s}}$ for 10 min. The requirement is parsed if the system does not stop operating.

One of the additional improvements listed in the problem definition is the data acquisition system capability to acquire data in the real world and to be used in a contained environment. The system will not aim at real time operation. The client should transmit its acquired data in chunks or packages instead of continuously transmitting its acquired data. The packages of acquired data should be automatically transmitted to the server. Then the data acquisition system is used in the contained environment the client is likely to be a part of experiments which often will follow a specific schedule. Automatically transmitting its acquired data, might not be sufficient in an experiment and it could be needed that the client is capable to interact on input and follow a predetermined deadline. In this proof of concept the client will have to be able to listen and react to input from a user of the system and start transmitting its data to the server within 30 s from the input is determined with a succesrate of 90 %. These requirements will be placed in the client requirement section.

The requirement for the system is listed below.

Sy1. The system is able to handle 3 concurrent clients experiencing a worst case situation where it is acquiring data from a file transfer with mean data rate between 1.2 MB and 1.8 MB for 10 min. By handling it is meant that the system should not break and stop operation.

In the following sections the requirements for the client and server part of the system will be derived.

3.3.1 Client requirements

The client is responsible of acquiring data on the host. The improvements listed in the problem definition 2.4 related to the the data acquisition is; acquire data which is not vulnerable for evasion techniques, acquire topology and protocol independent data and work on multiple platforms. Further as derived in the system requirements section, the client should react to user input.

The client should acquire data which is difficult to evade. It will be limited to data which is vulnerable to time evasions, meaning the client should acquire data which is not dependent on time. The data which the client should acquire is already defined in section 3.2.

The client should acquire data which is not dependent on a specific topology or protocol. This means that the client should not assume that the botnet C&C channel does not use a specific protocol or topology. It is also found that data acquisition system should be able to acquire data from multiple platforms. The clients supported platforms will be limited to computers running Windows or Linux operating systems.

Defined in the system requirements is that the the client must be able to react to user inputs and transmit its acquired data to the server. It is defined that the in 90 % of the user inputs the time from input to the data is transmitted may not take any longer than 30 s. In case of no connection to the server, the client must be able continue acquiring data and store the acquired data for later transmission in order not to loose the acquired data.

The requirements for the client part of the system is listed below.

C1. The client must acquire data which is is not dependent on topology, protocols or time.
C2. React to user input and transmit its acquired data to the server within 30 s in 90 % of times.
C3. Automatically transmit acquired data in packages to the server.
C4. Capable of storing data and transmit the data when the client has reestablished a connection to the server.
C5. Operate on both Windows and Linux operating systems.

3.3.2 Server requirements

The server is responsible for receiving and processing the packages of acquired data which is transmitted from the client. The received data is stored in a database from which a detection system would be able to extract data.

None of the improviement listed in the problem defintion is regarding a server as the server is a element to achieve the distributed structure of the data acquisition system. Defined in the system requirements is the scalability of the system which also defines the requirement for the server. In a worst case situation, defined in section 3.3, the tollorated server behavior needs to be defined. The maximum package loss rate of transmitted data from the clients will be 1 %.

The requirements for the server is listed below.

Se1. Receive all data from the clients in the defined worst case situation. Maximum loss tolerated is 1 %.

With the requirements defined how they will be tested also needs to be defined in a set of acceptance tests.

3.4 Acceptance test

The acceptance test will verify the system requirements. These acceptance tests will be defined in this section. Three acceptance tests will be designed which together will cover all the requirements set for the system and its parts; the client and the server.

3.4.1 Scalability test

This acceptance test will verify whether the scalability requirements for the system, which is Sy1 and Se1, are fulfilled. The scalability requirement is defined as a situation where 3 clients concurrently experience a worst case scenario. This worst case scenario on the client is created by starting a file transfer with must have a mean data rate between 1.2 MB and 1.8 MB and last for at least 10 min. This will generate a lot of packets which the client acquires and processes and eventually transmit to the server in packages. The output in the log files from the client and server will be used to determine whether packages of acquired data, transmitted from the client, is reported to have been lost or failed in processing.

3.4.2 Data acquisition

This acceptance test will test requirement C1, C3, and C4 which are all related to the system ability to acquire data. To test these requirements the client will be deployed on 5 computers where at least one of them will be running a Linux operating system. The server will be deployed on a computer with has a global accessible IP to enable the clients to contact it. The client will be running on the computers for 14 days and from the log files it can be confirmed whether the server has received data from all the clients. This will mean that the clients work on both Windows and Linux and that they are capable of automatically transmitting the acquired data. Requirement C4 is about the client ability to handle situations where the server is not reachable therefore have to wait to transmit its acquired data. This will be tested by blocking the server for a duration of 2 h on the Linux client with iptables. It is inspected that the client is transmitting its acquired data upon successfully reconnected to the server. The requirement C1 regarding which data is acquired is confirmed by inspecting the server database.

3.4.3 User interaction

Requirement C2 is stating that the client should be able to react to user input and transmit its data to the server within 30 s from the input with a succesrate of 90 %.

This will be tested by an automated test where the client will be instructed to transmits its acquired data to the server. Upon experiment start the user input will be set after a uniform randomly chosen time between 30 s and 90 s. This is done in order to eliminate the risk of getting in synchronization with eventual internal components in the client and therefore get misleading results. Flows will be created during the experiment to ensure that there is data to acquire and transmit to the server. The experiment will be repeated 100 times. The time it takes from the client is instructed to it has initiated the transmission to the server is calculated for all experiments using the clients log and the experiment script log file.

Chapter 4

Design

In this chapter the system will be designed. It has been chosen to base the design on a system called Volunteer Based System (VBS), which will be described first. Then the needed modifications are derived and designed. VBS already includes a lot of the desired functionalities. It is derived, which changes to VBS are needed to fulfill the requirements. These changes are mainly about acquiring DNS data and enable user interaction.

The system to be designed is a proof of concept data acquisition system. It is chosen that the system should fulfill the structure shown in figure 3.1 and is intended to acquire network and host data.

The system to be designed and implemented will be built on top of an existing system called VBS. VBS is developed by Tomaz Bujlow [Bujlow et al., 2012] as a part of his PhD and is designed to acquire network traffic traces from volunteers computers. The data collected is meant to be used for studies on Internet behavior and traffic, and is mainly used in the development of a traffic classifier. VBS is a host based data acquisition system where clients are installed on the host and the data acquired is transmitted to a centralized server. Further the VBS client is designed to work on both Windows and Linux. VBS has during the years had 92 different clients, where some has been active for years, indicating that it is a solid and well proven system.

VBS has a lot of the functionalities which is wanted for this project data acquisition system, which includes its already defined interface between client and server. The VBS client is already acquiring a lot of the wanted data and automatically transmitting it to the server. This almost covers requirements C1, C3, C4. Further the client is implemented in java and works on both Linux and Windows which is defined in requirement C4. VBS in its present state not capable of acquiring DNS data and it is not possible for a user to interact with the system. Further VBS clients are designed to be light on resources as it is a system for volunteers and a high resource usage would annoy the volunteers who would simply remove the system again. For that reason it should be capable of handling the worst case scenario.

Before the changes are being designed into VBS, a overview of how VBS works is needed.

4.1 VBS overview

VBS is designed to acquire real world internet traffic data from volunteers [Bujlow et al., 2012]. It is designed run in the background as a daemon and attempts not to annoy the volunteer by consuming a minimum of system resources [Netforsk,]. The structure of VBS is similar to the one presented in the system overview figure 3.1.

VBS is designed to acquire real world internet traffic for research purposes. It acquires data from packet headers and supports IPv4, TCP, UDP, HTTP and ICMP protocols. All packets are sorted into flows by the five tumble key (source IP, destination IP, source port, destination port and protocol name). Along with flow and packet information the VBS client acquires which processes has created the sockets and collect the CPU usage used by the client.

VBS acquires the following data:

- Packet source IP.
- Packet destination IP.
- Packet source port.
- Packet destination port.
- Packet protocol.
- Packet direction.
- Packet length.
- Packet payload length.
- Packet timestamp.
- Presence of TCP SYN, ACK, PSH, FIN, RST, CWR, ECN and URG flag.
- TCP packet sequence and acknowledgement number (since a revision in 2013 [Jacobsen, 2013]).
- HTTP content type.
- Flow starttime.
- The identified global IP.
- CPU usage of OS and the VBS client.
- Process name of the process which created the socket.

The interface between the client and server used for three operations:

- Transfer acquired data to the server.
- Remote update of the client.
- Register and validate the clients ID.

The VBS client stores its acquired data in SQLite [SQLite,] database file (denoted `.fdb` files) and transferred to the server. The interface is between the client and server is defined in the structure of this database. Remote update allows updating of the connected clients from the server. Registrating and validating client id is done prior to the client data acquisition startup.

VBS consists of several independent threads each with their own task. Instead of describing all objects in VBS only those with having a thread as describing these described how the VBS client and VBS server works rather well. The classes from the VBS client listed below have individual threads and their relationship can be seen in the class diagram in figure 4.1.

- `VbsClient` is the main thread and creates the other threads and starts and monitors them.

- `PacketCapturer` places and monitors `Capturer` threads, one for each NIC on the host which is not a localhost interface. Packet information is saved in `CapturedPacket` objects and saved in a `CapturedPacketQueue` which is a synchronized interface.
- `SocketMonitor` initiates the program netstat on Linux and tcpvcon.exe on Windows to acquire information of open sockets and its processes on the system.
- `FlowGenerator` groups packets obtained from the `CapturedPacketsQueue` into flows and associates socket information from `SocketMonitorQueue` with flows. Flow information are stored in a `Flow` object. Closed flows are placed in the `ClosedFlowsQueue`.
- `DatabaseHandler` collects closed flows from the `ClosedFlowsQueue` and saves their information (including all packets and socket information) in a SQLite database. When the database contains more packets than a predefined threshold, it is saved into a `.fdb` file and a new SQLite database is created.
- `DataTransmitter` detects the presence of `.fdb` files and transmits them to the VBS server.
- `PerformanceMonitor` collects CPU usage data from the OS on a predefined interval. The information is stored in a `PerformanceMonitorQueue` which is obtained from the `DatabaseHandler`.
- `ShutdownManager` is responsible of proberbly shutting down the client upon request of forced due to detected errors. It contains a hook to all threads, so it can initiate their shutdown sequences and close the client in the right order. This thread is initiated by the VbsServer upon a detected failure the program is requested to be closed.

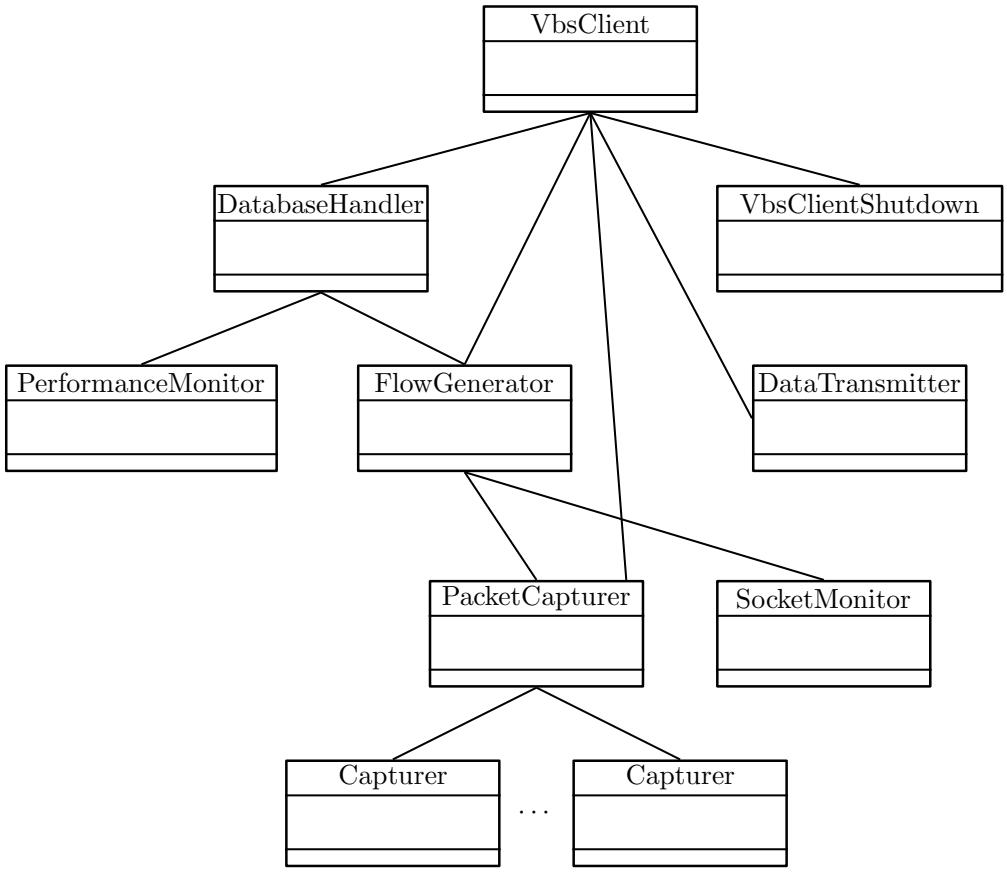

Figure 4.1: Class diagram for VBS client classes with threads.

The server is structured in a similar manner as the client. It contains the following threads witch class

diagram can be found in figure 4.2.

- `VbsServer` is the main thread and initiates the other threads and monitors them.
- `FileTransferServer` is the thread responsible for receiving files from the clients.
- `DataExtractor` identifies newly received files, opens them and extracts their data to the database.
- `RegisterServer` is responsible for providing new clients a unique client id.
- `UpgradeServer` is receiving requests upon the client startup, whether there is a update to them.
- `VbsServerShutdown` contains a hook to all threads and initiates their shutdown procedure in the right order to ensure everything is closed properly. This thread is initiated by the VbsServer upon a detected failure the program is requested to be closed.

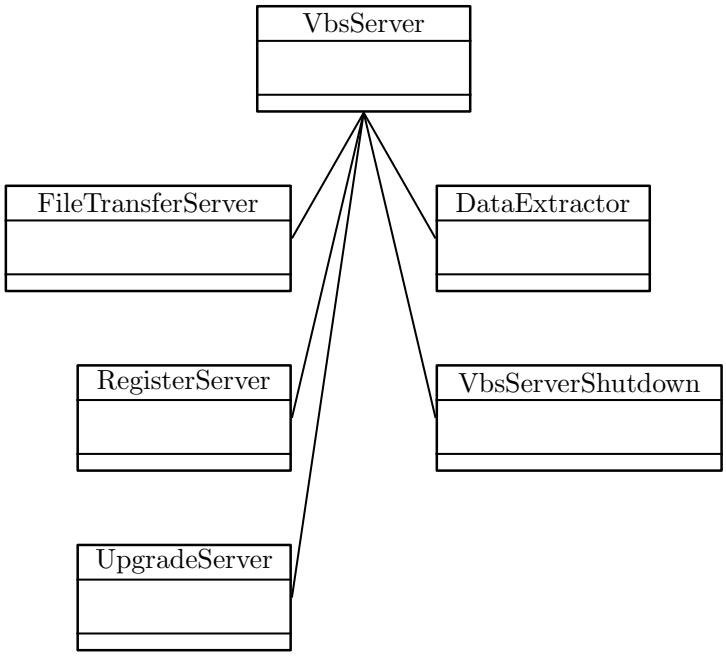

Figure 4.2: Class diagram for VBS server classes with threads..

With the knowledge of how VBS works it is the system can be designed.

4.2 System design

In this section it will be derived which changes are needed to be made to VBS in order to design the proof of concept system. To identity the changes which needs to be made to VBS in order to design the system, table 4.1 shows the requirements and what the system needs to do to fulfill them.

In order to identify what additional data is needed to be acquired by the system the data which is already acquired by the VBS client and the data the system is defined to acquire can be seen in table 4.2. The table reveals that only DNS A record response data is missing. The handling of DNS packet will be done in a new thread which will be called `DNSMonitor`. Table 4.1 also shows that the only other change what needs to be made to VBS is to enable the client to react to user inputs.

Requirement	Design plan
Sy1. The system is able to handle 3 concurrent clients experiencing a worst case situation where it is acquiring data from a file transfer with mean data rate between 1.2 MB and 1.8 MB for 10 min. By handling it is meant that the system should not break and stop operation.	The server file transfer should be multithreaded, which it is in VBS. The clients should only send their package after they reach a certain size. In VBS this is set to 488 kB. No change needed.
C1. The client must acquire data which is is not dependent on topology, protocols or time.	Check if all required data is acquired. See table 4.2.
C2. React to user input and transmit its acquired data to the server within 30 s in 90 % of times.	Modify the client to react to user inputs. This change will be designed in section 4.4.
C3. Automatically transmit acquired data in packages to the server.	This is already designed and implemented in VBS.
C4. Capable of storing data and transmit the data when the client has reestablished a connection to the server.	The independent thread design enables the `FileTransmitter` to do this, so no change needed.
C5. Operate on both Windows and Linux operating systems.	No change needed as the VBS client already works on both Windows and Linux. Potential plugins and extensions should be validated in both Windows and Linux to keep it compatible.
Se1. Receive all data from the clients in the defined worst case situation. Maximum loss tolerated is 1 %.	The `DataTransmitter` should incorporate a check whether the transmission was successful and only delete files upon successful upload. This check is already existing in VBS.

Table 4.1: Requirements and design plan for the system.

Data needed	Already in VBS?
Packet length.	Yes.
Packet header length.	Yes.
Packet direction.	Yes.
Packet timestamp.	Yes.
TCP flag indicator.	Yes.
TCP sequence and acknowledgement number.	Yes.
DNS A response time to live.	No.
DNS A response quantity of answers.	No.
DNS A response query domain name.	No.
Process name.	Yes.

Table 4.2: What required data is already acquired by VBS?

4.3 DNSMonitor

In this section the extension to VBS which is responsible to DNS A response data will be designed. The A DNS type is for IPv4 domain name queries. In order to keep the same design pattern as the other parts of VBS which acquires data, a new part will be designed and is intended to process DNS packets. This part will be in a package which will be called `dnsMonitor` and its main thread will be located in the class `DNSMonitor`. The part should contain interfaces and procedures similar to the other parts of VBS so it needs to contain:

- `DNSMonitor` needs a `start()`, `stop()`, `isAlive()`, `run()` and `maintenance()` methods so it can be started and stopped, checked, perform operations while running and is capable of maintenance itself.
- `DNSMonitor` needs a thread safe input and output.
- A method capable of linking the DNS info with a flow.
- It needs to get the DNS packets.
- The data acquired will need to be incorporated into the SQLite database in `DatabaseHandler` and the server database in `DataExtractor` (not shown in the figure).
- Eventually it needs to be incorporated along the other threads in the `VbsClient` and in the `VbsClientShutdown`

The `dnsMonitor` package with it s classes are shown in the class diagram on figure 4.3. The class diagram showing all changes related to adding the `DNSMonitor` can be found on figure 1 in Appendix. The following of the this section will design the `DNSMonitor` from its input, output and run time operations.

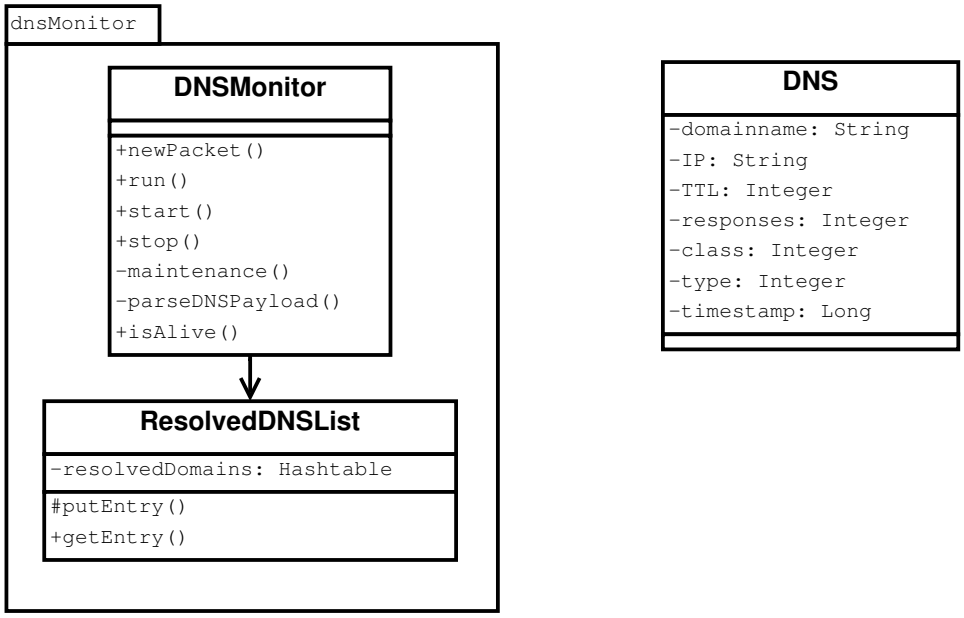

Figure 4.3: `DNSMonitor` class diagram

Input The input to the `DNSMonitor` will be captured DNS packets with payload. DNS packets can be identified by being UDP or TCP packets and have source port 53 [Mockapetris, 1987]. As it needs to be

incoming the source port becomes the remote port in VBS. This project will limit to only consider DNS packets transmitted with UDP. For that reason all packets which have remote port 53 and is UDP will have its UDP payload saved in the `CapturedPacket` object and the packet is given "DNS" as application name attribute. All packets are retrieved by the `FlowGenerator` which will check the `CapturedPacket` application name. If this is "DNS" it will parse it to the `DNSMonitor` through the `newPacket()` method.

Output The output from the thread should be usable by the `FlowGenerator` to match DNS data with flows. This will be done by matching the flows remote IP with resolved IP found in DNS responses.
The resolved DNS domains and IPs are saved in a hashtable indexed by a hash of the resolved IP. The content is the `DNS` object. The hashtable is located in the `ResolvedDNSList` where lookups can be done with the `getEntry()` method. The `FlowGenerator` thread will attempt to match flows with DNS records using the `matchWithDNS()` method called from every new `createNewFlow()` and `updateFlow()` method which will be in the `FlowGenerator`.

Run The `run()` method is where all operations for the `DNSMonitor` thread while running is done. It first thing it needs to do is to look whether there is new packet in the queue. If so it will parse its content. Then it will conduct maintenance which will be done by checking all saved DNS entries in the hashtable, to see whether some have expired by checking their Time To Live (TTL) value and the time the packet was captured Then the thread will sleep for a predefined amount of time to free up CPU resources for other tasks.

Database structure The DNS data will also have to get saved in the databases along with the other data acquired by the client. The first database which the DNS data needs to be saved into is the SQLite database which is managed by the `DatabaseHandler` and is used as a interface to the server. On the server the data is extracted and saved into the large server database, which is done by the `DataExtractor`. In both databases the DNS data will be saved in a new table. In order to link a DNS row with a flow the DNS table will contain the flow id of its corresponding flow. On the server each DNS row will further have a unique id to uniquely identify the DNS entry. An example of a DNS row saved in the server database can be seen in table 4.3.

dns_id	flow_id	IP	domainname	ttl	responses
92	93	216.58.209.98	www.googletagservices.com	224	2

Table 4.3: DNS database table structure with example.

4.3.1 parseDNSPayload()

The data in the DNS packet payload needs to be parsed to be extracted. The DNS A record response packet structure can be seen in figure 3 in Appendix.

All DNS packets contains the DNS header. The DNS header identifies whether the DNS packet is a request or an answer by the `QR` bit. Then the packet is a DNS request it will contain a question section and if the packet is a DNS response it will contain at least one response section after the question

part. The quantity of questions in the DNS packet are given in the `QDCOUNT` field and the quantity of answers are given in the `ANCOUNT` field. The structure supports multiple questions, but it is rarely used [Mockapetris, 1987], so this system will not support DNS packets with more than one question. A requests and responses are identified by the class value 0x01 [Mockapetris, 1987]. It is often seen that answers with class CNAME answers comes before A answers as they are used to redirect to an other DNS server. So the parser needs to be able to identify A answers and skip the other classes.

The algorithm for parsing a DNS packets payload is written in pseudo code in algorithm 1.

```
if QR bit == 1 and QDCOUNT == 1 then
    responses ⟵ ANCOUNT;
    move payload pointer to QNAME;
    while an 0x00 is not encountered in the payload do
        Read subdomain length;
        Extract the subdomain;
    end
    if QTYPE == 1 and QCLASS == 1 then
        foreach responses do
            dns ⟵ new DNS object;
            dns.domainname ⟵ extracted concatenated subdomains;
            if TYPE != 1 then
                skip this response;
            end
            dns.responses ⟵ responses;
            dns.type ⟵ TYPE;
            dns.class ⟵ CLASS;
            dns.ttl ⟵ TTL;
            dns.ip ⟵ RDATA;
            add dns to the resolved domain list;
        end
    end
end
```

Algorithm 1: Parse a DNS packet

The `QNAME` has a variable length and there is no single length field to define the total length of the requested domain name. Instead each subdomain in the domainname (subdomains are splitted by dots in a webbrowser) are started with a field [Mockapetris, 1987]. Therefore to parse the entire domain name it is needed to iterate through each subdomain. The domain name field entry end is marked with with a zero.

The question and response class will have to be IN or 0x01 for Internet in order be sure the question is meant for normal Internet DNS queries. The other classes for questions are only used in special cases. With the question parsed it is time to parse all the responses. The entry `NAME` can contain the name of the owner of the registrated resource. This entry will be ignored as it is not needed. If a response is not of type A (e.g. `CNAME` or 0x05) this response should be skipped, so the parser should read until the end of the response and start parsing on the next response instead. If the response is a type A and in the Internet class the 32 bit long `TTL` is parsed. The IP in an A response is located in the

RDATA which length is identified by the RDLENGTH entry. Information from A responses is putted in separate DNS objects and when the DNS packet payload is fully parsed, it is placed into the resolved DNS list.

A class diagram showing all changes made to VBS in order to incorporate the DNS part into VBS can be seen in figure 1 in the Appendix.

4.4 User input

The requirements for the client states that the client should be able to react to user inputs and transmit its stored data within 30 s in 90 % of the cases. This will be achieved by enabling the VbsClient to periodically read a configuration file which will contain a "push" flag. If using the system for experiments it might further be wanted to group a set of flows. As VBS does only group flows by client ID it would be needed to add an additional tag. This will be done by adding a customizable "tag" entry in the configuration file which the VbsClient will read and parse to the DatabaseHandler which will append it to all flows.

The VBS client already has implemented routines to read and write XML formatted files for configuration purposes, so the obvious choice is to create a new file give it the same structure as the existing configuration files. The new file will be called custom.cfg and contain XML entries called "tag" and "push". The tag requires change to both the SQLite database handled by the DatabaseHandler and the server database which is handled by the DataExtractor. These modifications will not be presented, instead this section will focus on the "push" flag instead.

The requirement states that the client should have transmitted its files within 30 s. The threads involved in handling the acquired flows are FlowGenerator, DatabaseHandler and FileTransmitter. These threads are running individually by depends on each other as a chain to get the acquired flows transmitted. The thread responsible for detecting finalized databases files is the DataTransmitter which is in VBS set check for new files every 10 s. The thread responsible for generating finalized database files is the DatabaseHandler which is awake every 10 s as well. The thread which contains current open flows and handles newly arrived packets is the FlowGenerator which is awake every 1 s and only sleeps if there is no new packets. The thread which will be responsible for reading the configuration file is the VbsClient which also is awake every 10 s.

These threads operated in a chain order, meaning that the worst case delay where the "push" flag is read or set just after it was last checked gives a total delay of $10\,\text{s} + 1\,\text{s} + 10\,\text{s} + 10\,\text{s} = 31\,\text{s}$ plus processing time. This is above the requirement of 30 s. In an attempt to make sure the requirement is met, the DataTransmitter will check every 8 s instead lowereing the worst case delay from user input to 29 s. The 90 % value will probably be even lower so that requirement should be fulfilled.

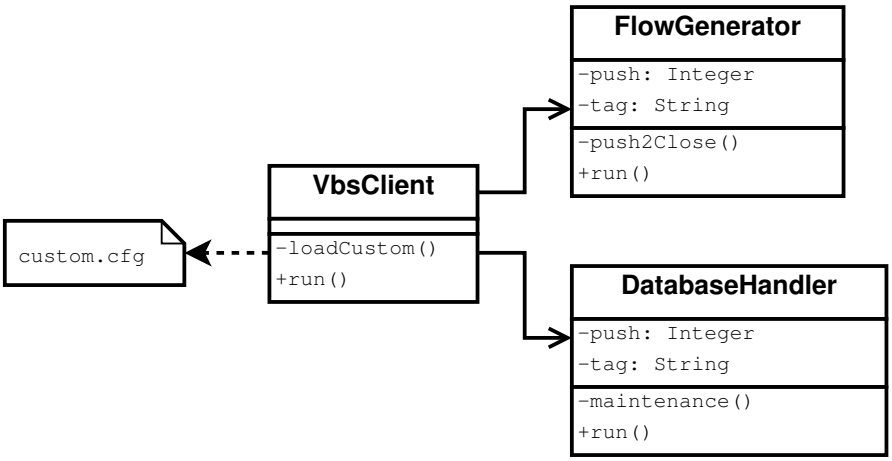

Figure 4.4: Class diagram of interaction incorporation into VBS.

The class diagram for the user input changes is seen in figure 4.4. As illustrated in the class diagram in figure 4.4 the `VbsClient` reads the "push" and "tag" from the `custom.cfg` file. If the "push" flag is set it will:

1. Instruct the `FlowGenerator` to close all open flows. This is done by calling the `setPush()` function. The `FlowGenerator` will check if there is set a push flag every 10 s.
2. Then it will instruct the `DatabaseHandler` to flush its data, meaning it will write all data to a database file and finish it so it wil become a `.fdb` database. This is done by parsing the push flag to the `DatabaseHandler` using the `setPush()` function. The `DatabaseHandler` will check for the "push" flag every 10 s.
3. The `DataTransmitter` will detect the created `.fdb` database files every 8 s and send them to the server.

The complete client class diagram can be found in figure 2 in the Appendix, where all classes are present and those already existing in VBS but modified is blue and those which are created is green. With the system designed to fulfill all requirements it is ready to be implemented.

Chapter 5

Implementation

In this chapter the implementation of the designed system will be presented. The implementation presented in this chapter will focus on the parts of the system related to capturing a packet, its way through the system and to the server database. There will be special attention to the extensions to VBS designed in this project, with the new `DNSMonitor` *part and the clients ability to react to user input.*

5.1 Client

In this section the implementation related to how a packet is captured, sorted into flows, stored into the client database and eventually transmitted to the server will be presented. Further it will show the DNS addition and how the client user interaction is implemented.

5.1.1 Capturer

The `Capturer` class is containing the code used to capture packets on the computers network interfaces. The methods of capturing packets is from VBS, which uses the library JNetPcap [Technologiesy, 2014] to interface to `winpcap.exe` for Windows and `libpcap` for Linux operating systems. Interfacing JNetPcap requires overwriting its `PcapPacketHandler` and its method `nextPacket` which is the method called from JnetPcap when a new packet has been captured. Parts of the `nextPacket` method is shown in the code snippet 5.1. Notice that throughout the code snippets in this chapter, three dots like "..." means that lines have been omitted from the snippet. This is done to simplify the code snippet and make the important parts of the implementation more clear.

Code snippet 5.1: Capturer nextPacket method.

```
1  public void nextPacket(PcapPacket packet, Integer intValue) {
2    ...
3    // A new CapturedPacket object is created ready to get the data from the PcapPacket
         object belonging to JNetPcap.
4    CapturedPacket newPacket = new CapturedPacket();
5    newPacket.setTimestamp(packet.getCaptureHeader().timestampInMicros());
6    newPacket.setPacketSize(ip.length());
7    ...
8    // The packet direction by calculated by comparing src and dst IP with NIC IPs.
9    ...
10   // If a TCP header is identified, extract the relevant data.
11   if (packet.hasHeader(tcp)) {
12     ...
13     newPacket.setFlagACK(tcp.flags_ACK());
14     ...
15   // If a UDP header is identified.
16   } else if (packet.hasHeader(udp)) {
```

```
17        newPacket.setProtocolName("UDP");
18        newPacket.setPayloadSize(udp.getPayloadLength());
19        ...
20        // And if the packet is incoming and the remote port is 53, save the packet
              payload and mark it as a DNS packet.
21        if (newPacket.getRemotePort() == DNSMonitor.DEFAULT_DNS_PORT && newPacket.
              getPacketDirection() == 'I') {
22          newPacket.setPayload(udp.getPayload());
23          newPacket.setApplicationName("DNS");
24          log.debug("Captured DNS Packet");
25        }
26      }
27      // Add the CapturedPacket with the saved information to the CapturedPacketQueue.
28      myPacketQueue.queueUpPacket(newPacket);
29    }
```

The `nextPacket` method initiates by creating a `CapturedPacket` object which is the object representing a packet in the system In this object relevant data from the packet is acquired from the JnetPcap library. Depending on the packet protocols present in the packet the relevant information from the headers are extracted. If a UDP header is identified and it has the remote port 53 and its direction is inbound, its payload is saved in the `CapturedPacket`, the application name is set to "DNS" such that the `FlowGenerator` can identify the packet as a DNS packet. The created `CapturedPacket` with the retrieved data is eventually added to the `CapturedPacketQueue`, ready to be retrieved by the `FlowGenerator`.

5.1.2 FlowGenerator

The `FlowGenerator` is responsible for grouping all information acquired into flows. It acquires the `CapturedPacket` in its thread loop, which is shown in code snippet 5.2. If there is a new packet, it will investigate whether the packet is a DNS packet. If it is a DNS packet it will be parsed to the `DNSMonitor` and skips to the next packet in to queue. If the `CapturedPacket` is not a DNS packet it will be checked whether the flow already exists, by checking if the packets five tumble key (used as a hash), exists in the hashtable containing active flows. If the flow already exists, it is updated so the new packet is added and if it does not exist a new `Flow` is created. Eventually the `FlowGenerator` will check whether the "push" flag has been set. If it has, it will close all its active and flows waiting for timeout and reset the "push" flag.

Code snippet 5.2: FlowGenerator run method.

```
1   ...
2   // Infinite loop.
3   while (runFlag) {
4     // Attempt to retrieve a packet from the CapTuredPacketQueue.
5     newPacket = myCapturedPacketsQueue.getNextPacket();
6     if (newPacket == null) {
7       ...
8     } else {
9       // If the retrieved packet is marked as a DNS packet, parse it to the DNSMonitor
            thread and end this loop run.
10      if (newPacket.getApplicationName().startsWith("DNS")) {
```

```
11          this.dnsMonitor.addPacket(newPacket);
12          continue;
13        }
14        // If the retrieved packet is not a DNS packet check if there is current open
              flow in the open flow hashtable. If there is update the flow, otherwise create
               a new one.
15        currentFlow = openFlowList.get(newPacket.hashCode());
16        if (currentFlow == null) {
17          createNewFlow(newPacket);
18        } else {
19          updateFlow(newPacket, currentFlow);
20        }
21      }
22      // If the push flag has been set flush all open flows including those who are
            waiting to timeout.
23      if (this.push == 1) {
24        this.flushTimeoutFlows();
25        this.closeAllFlows();
26        this.push = 0;
27      }
28      ...
```

Every time a flow is either created, updated or closed it will be checked whether there has been acquired a DNS response which can be associated with the flow. This is done using the `matchWithDNS` method seen in code snippet 5.3. It starts by checking whether the flow does already have a `DNS` object associated with it. If not, then it will check whether an entry with the flows remote IP exists in the `ResolvedDNSList`. The `dnsMonitor` package is implemented in section 5.1.5.

Code snippet 5.3: FlowGenerator matchWithDNS method.

```
1   private Flow matchWithDNS(Flow flow) {
2     // If there is no DNS attacked to the flow.
3     if (flow.getDNS() == null) {
4       // Check if there is a DNS retrieved to the flows remote IP.
5       DNS dns = dnsList.getEntry(flow.getRemoteIP());
6       try {
7         if (dns.isOK()) {
8           // Attach the found DNS to the flow.
9           flow.setDNS(dns);
10        }
11      } catch (Exception e) { }
12    }
13    return flow;
14  }
```

5.1.3 DatabaseHandler

As flows are closed by the `FlowGenerator`, they are collected by the `DatabaseHandler`. The method `emptyStorage` is activated by either a "push" flag or when enough flows and packets or performance measurements are acquired. This last behavior is unchanged from VBS. Whether is it time to acquire

closed flows is determined in the `maintenance` method shown in code snippet 5.4. If a "push" flag is set all closed flags will be putted in the current database by the `emptyStorage` method shown in code snippet 5.5 following a replacement of the current database with a new one so the old can be finalized to a `.fdb` file ready for transmission to the server by the `DataTransmitter`.

Code snippet 5.4: DatabaseHandler maintenance method.

```
1  private void maintenance() {
2    // Defined in VBS when to put closed flows into the database.
3    if (this.myClosedFlowsQueue.getNumberOfFlows() + this.myClosedFlowsQueue.
         getNumberOfPackets() > 20000) {
4      emptyStorage();
5    }
6    if (this.myPerformanceMonitorQueue.getNumberOfItems() > 100) {
7      emptyStorage();
8    }
9    // If the push flag is set, collect all closed flows using the empthystorage method
         and finish the current database by initiating a new one. Remember to reset the
         push flag again.
10   if (this.push == 1) {
11     log.info("Emptying storage");
12     emptyStorage();
13     setDatabase();
14     this.push = 0;
15   }
16 }
```

The empthyStorage method acquires closed flows from the `FlowGenerator` and saved into the current open sqlite database. This is done by inserting the flows information in the flows table, packets into the packets table, application information into the application table and DNS into the DNS table. In the code snippet only inserting of the DNS data is shown form the DNS object saved in the flow, if there is one. If the file is becoming larger than 500 000 B = 488 kB the database is changed as well which is also unmodified from VBS.

Code snippet 5.5: DatabaseHandler emptyStorage method.

```
1  ...
2  // Retrieve the next closed flow from the FlowGenerator
3  Flow savedFlow = this.myClosedFlowsQueue.takeClosedFlow();
4  ...
5  // Extract information from the Flow and the belonging CapturedPackets into their
       representative tables.
6  ...
7  // Extract the attached DNS object from the Flow.
8  DNS savedDns = savedFlow.getDNS();
9  ...
10 // If there is a attacked DNS save its attributes into the table.
11 if (savedDns != null) {
12   this.db.executeUpdate("INSERT INTO DNS(" +
13   "flow_id, " +
14   "ip, " +
```

```
15      "domain, " +
16      "ttl," +
17      "responses" +
18      ") VALUES(" +
19      dbFlowId + ", '" +
20      savedDns.getIP() + "', '" +
21      savedDns.getDomain() + "', " +
22      savedDns.getTTL() + ", "+
23      savedDns.getResponses() + "" +
24      ");");
25    }
26    ...
27    // If the database file is large enough, finalize the current database file and start
          using a new one.
28    File fileChecker = new File(this.databasePath);
29    if (fileChecker.length() > 500000) {
30      setDatabase();
31    }
32    ...
```

5.1.4 DataTransmitter

The `DataTransmitter` is responsible for identifying finalized database files ready to be transmitted to the server and transmit them. This is done in the `transmitUnsentFiles` method which is seen in code snippet 5.6. The transmitUnsentFiles will as long there are files to detect, identify the newest one in as a LIFO queue. This file is transmitted using a `FileTransferClient` object which corresponds to the `FileTranferServer` object. If the file is transmitted without error, it is deleted.

Code snippet 5.6: DataTransmitter transmitUnsentFiles method.

```
1   ...
2   // Infinite loop in the thread.
3   while(true) {
4     ...
5     // For all finalized files in the data directory, select a finalized database file.
6     for (File file : fdbFiles) {
7       if (file.getName().endsWith(".fdb")) {
8         ...
9         currentFile = file;
10        ...
11      }
12    }
13    try {
14          // Attempt transmitting the file to the server.
15        if (!myTransferClient.uploadFile(currentFile)) {
16            log.error("Could not transfer the file: [" + currentFile + "]");
17        break;
18      }
19    } catch (FileNotFoundException e) {
20      log.error("Could not find file to transfer: [" + currentFile + "]", e);
```

```
21        break;
22      }
23      // If there was no errors in transmitting the file, it is assumed it was successful
            and the file is deleted.
24      if(!currentFile.delete()) {
25        log.error("Could not delete file after transfer: [" + currentFile + "]");
26      }
27    }
```

This part followed a packet from being acquired by the client until it was transmitted to the server. The next part will implement the `DNSMonitor`.

5.1.5 DNSMonitor

The `DNSMonitor` is responsible of receiving, parsing and storing DNS A records. It needs a input which is a `ConcurrentLinkedQueue` where `CapturedPackets` can be putted, stored and retrieved synchronized, avoiding a mismatch caused by asynchronious threads. The input to the queue is through the `addPacket` method which leads to the `queueUpPackets` method which is seen in code snippet 5.7.

Code snippet 5.7: DNSMonitor queueUpPackets method.

```
1   protected boolean queueUpPacket(CapturedPacket packet) {
2     return listOfPackets.add(packet);
3   }
```

The output of the `DNSMonitor` is through the object `ResolvedDNSList` which contains a hashtable indexed by remote IP hash and contains resolved domain in `DNS` objects. This `ResolvedDNSList` methods is synchronized to avoid mismatch between the asynchronous threads. The look up in the table is done through the `getEntry` method shown in the code snippet 5.8.

Code snippet 5.8: ResolvedDNSList getEntry method.

```
1   public synchronized DNS getEntry(String IP) {
2     return resolvedDomains.get(IP.hashCode());
3   }
```

The `DNSMonitor` is designed to be run as an independent thread and in its run routine shown in code snippet 5.9 it will look for new packets in the input queue and if there is one, it will use the `parseDNSPayload` method to parse its content. The `parseDNSPayload` method returns an array of `DNS` objects containing the DNS response data, which all are inserted to the hashtable in `ResolvedDNSList`.

Code snippet 5.9: DNSMonitor run method.

```
1   // Infinite loop in the thread.
2   while (runFlag) {
3     ...
4     // Retrieve the next packet from the input queue.
5     newPacket = this.getNextPacket();
6     // If there is no packet in the queue, sleep to free/save CPU resources and
          maintain the resolved DNS list.
7     if (newPacket == null) {
```

```
8       Thread.sleep(1);
9       cleanUpDNSList();
10    }
11    // If there is a packet in the queue, parse its content and save all parsed DNS
          responses in the resolved DNS list.
12    else {
13      dnsListReturn = parseDNSPayload(newPacket);
14      for (DNS newDNS : dnsListReturn) {
15        if (newDNS.isOK()) {
16           this.myResolvedDNSList.putEntry(newDNS);
17        }
18      }
19      ...
20    }
21  }
```

The `parseDNSPayload` method is shown in code snippet 5.10. It starts by initiating the `ArrayList` intended to be returned with DNS entries. Parsing the payload is a matter of keeping track of the pointer index in the payload so the entries are read at the right location in the payload. The payload starts with the DNS header which contains the flag which identifies whether the DNS packet is a response. If it is not the method returns a null straight away. If it is the quantity of queries and responses are extracted. In order to read an entry in the payload masking of the bits is needed, both for entire bytes and to extract single bits. Bit shift is used to convert bytes into to integers. The query name is defined as a chain of subdomains where each level starts with a length byte and finish is identified with a zero. For every subdomain the bytes are read into a char array and upon a new domain level start a dot is inserted to mark the beginning of the new subdomain. All the time the index is updated as it is vital in order to parse the rest of the payload. The method only supports one query in the payload, as defined in the design. For every response in the payload a new `DNS` object is created and the saved domain is stored in it along side the parsed response type, class, quantity of responses, the response TTL and the IP. If a response is not a A response, it will be skipped, meaning the index needs to be updated appropriately as well.

Code snippet 5.10: DNSMonitor parseDNSPayload method.

```
1   private ArrayList<DNS> parseDNSPayload(CapturedPacket newPacket) {
2     // Prepare the ArrayList of DNS objects to return and get the packets payload.
3     ArrayList<DNS> returnList = new ArrayList<DNS>();
4     byte[] payload = newPacket.getPayload();
5     Integer index = 2;
6     // Extract the response flag in the DNS header and return if it is not a response.
7     Integer is_response = Integer.valueOf((payload[index]&0b1000_0000)>>7);
8     if (is_response != 1) { return null; }
9     ...
10    // Extraction of quantity of queries and responses are not shown
11    ...
12    // Extract the queried domain by traversing through the payload but do not exceed
          the length of the payload.
13    for(int i = index; (i<payload.length && i<query_name.length); i++){
```

```
14      // If a 0x00 byte or the offset from the index is becoming longer than the name,
            break the loop and the query name is parsed.
15      if (payload[i] == 0x00 || offset>query_name.length-1){
16        break;
17      }
18      // Extract the length of the subdomain.
19      level_length = (payload[i]&0xff);
20      // If this is not the first byte of a sybdomain, add a dot and progress to the
            next byte.
21      if (i == index) {
22      } else {
23        query_name[offset] = 0x2e;
24        offset = offset+1;
25      }
26      i = i+1;
27      // Extract the entire subdomain.
28      for(h = 0; (h<level_length); h++) {
29        query_name[offset] = (byte) (payload[h+i]&0xff);
30        offset++;
31      }
32      // Update the index for the next subdomain.
33      i=i+h-1;
34    }
35    ...
36    // Reading query type and class and skipping packet if it is not class IN or an A
          query.
37    ...
38    // For all responses in the DNS response.
39    for(int i = 0; i<responses; i++) {
40      // Create a new DNS object.
41      newDNS = new DNS();
42      index = index+2;
43      // Save the DNS query name and type into the new DNS object.
44      newDNS.setDomain(query_name_string);
45      newDNS.setType(Integer.valueOf((payload[index]<<8)+(payload[index+1])&0xffff));
46      // If this response is not a A response but a CNAME response, update the index to
            the next response and skip this response.
47      if (newDNS.getType() != 1) {
48        if (newDNS.getType() == 5) {
49          index = index + 8; //Lets jump over class and ttl
50          index = index + 2 + Integer.valueOf((payload[index]<<8)+(payload[index+1])&0
                xffff);
51        }
52        continue;
53      }
54      ...
55      // Extract the TTL which is a 32 bit integer which takes 4 bytes.
56      newDNS.setTTL(Integer.valueOf(((payload[index+4]<<24)&0xffffffff)+((payload[index
            +5]<<16)&0xffffff)+((payload[index+6]<<8)&0xffff)+(payload[index+7]&0xff)));
57      ...
58      index = index + 10;
```

```
59        // Extract the IP in the response.
60        newDNS.setIP(FormatConverter.decimalFormat(Arrays.copyOfRange(payload, index,
              index+response_length)));
61        // If everything went okay mark the DNS object as OK and and save it into the
              return ArrayList.
62        newDNS.setOK(true);
63        newDNS.setResponses(responses);
64        returnList.add(newDNS);
65        // Update the index for the next response in the payload.
66        index = index+response_length;
67      }
68      return returnList;
```

5.1.6 VbsClient

The `VbsClient` is responsible for initiating the client meaning it loads the needed libraries, configuration files, created objects, starts threads as well as maintaining and supervises the threads. All these procedures for the existing threads in VBS are already implemented. They need to be created for the `DNSMonitor` as well. The part of the `VbsClient` main loop which is regarding the `DNSMonitor` and the custom configuration file is shown in code snippet 5.11. The main loop in the `VbsClient` starts by loading the custom configuration file. This is done with the `loadCustom` method. Then it sets up the `DNSMonitor` with the `setupDNSMonitor` method which creates the class. Then the system is started with the `startSystem` method which starts all the threads. Then the system starts a maintenance loop which checks all threads and reloads the custom configuration file. If something goes wrong, the system quits. VBS is launched by a Java service wrapper YAJSW which is configured to restart VBS if it quits, so it will effectively be a restarted. If the "push" flag has been set it is written to the log and the `FlowGenerator` thread and `DatabaseHandler` thread is giving the "push" flag as well. Eventually the `VbsClient` sleeps for 10 s and the garbage collector is executed remove unused objects from the memory.

Code snippet 5.11: VbsClient run method.

```
1   public void run() {
2     ...
3     // Load the configuration files.
4     loadCustom();
5     ...
6     // Configure all the clients components.
7     setupDNSMonitor();
8     ...
9     // In VBS it is implemented that if there was an error in starting the system
          threads completely exit.
10    if (!startSystem()) {
11      System.exit(1);
12    }
13    ...
14    // If the system was started successfully, monitor all threads.
15    while (areThreadsAlive()) {
16      ...
```

```
17      // If the DNSMonitor is suddently dead, log it and quit the problem.
18      if (!myDNSMonitor.isAlive()) {
19    log.info("DNS Monitor is for some reason dead. Restarting the client");
20    System.exit(1);
21    break;
22      }
23      // Remember to check the cuscom donfiguration file and if the push flag is set,
            parse the flag to the FlowGenerator and DatabaseHandler.
24      loadCustom();
25      if (push == 1) {
26        log.info("Pushing data to server");
27        this.myFlowGenerator.setPush(push);
28        this.myDatabaseHandler.setPush(push);
29        push = 0;
30      }
31      // Sleep for 10 sec and run the garbage collector.
32      Thread.sleep(10000);
33      System.gc();
34    }
35  }
```

This concludes the implementation of the designed client as a modification of VBS.

5.2 Server

The VBS server also had to be modified to become the server designed for this system. The modifications are mainly in the tables in the database. Data acquired from the client is received by the `FileTransferServer` thread which for each new connection initiates a `FileReceiver` object to receive the database file. The saved database file is identified by the `DataExtractor` which opens it and inserts the data into the server database.

5.2.1 DataExtractor

The `DataExtractor` will at the first run of the server create the servers database and its tables and indexes in the table. The database is created in the `createDatabase` method of which the parts related to the DNS table is shown in code snippet 5.12. First the connection to the database is established, then a statement is created which is a method of keeping the interface to the mysql database structured. This was already implemented in VBS. After the database is created the DNS table is created with the columns; DNS ID, flow ID, IP, domain, TTL and quantity of responses in the packet. The primary key is the DNS ID making each, meaning it is used as guide when new rows are inserted. To make queries faster when using flow ID as an entry indexes are created on flow ID which is the association with the flow.

Code snippet 5.12: DataExtractor createDatabase method.

```
1  private void createDatabase() throws Exception {
2    ...
3    // Initiate the connection to the database and prepare a statement.
```

```
4     Connection con = DriverManager.getConnection(url, dbUsername, dbPassword);
5     Statement stmt1 = con.createStatement();
6     ...
7     // Execute a query to create the database.
8     sql = "CREATE DATABASE vbs COLLATE utf8_bin;";
9     stmt1.executeUpdate(sql);
10    ...
11    // Create the DNS table and create a index lists for the table.
12    stmt1.executeUpdate("CREATE TABLE vbs.DNS(dns_id bigint unsigned not null
          auto_increment, flow_id bigint unsigned not null, ip char(20), domain varchar
          (255), ttl smallint unsigned not null, responses smallint unsigned not null,
          primary key(dns_id))");
13    stmt1.executeUpdate("CREATE INDEX index_flow_id ON vbs.DNS (flow_id);");
14    ...
15  }
```

A received SQLite database file from the client is identified by its `fdb` file extension and is processed by the `processSQLite` method shown in code snippet 5.13. A lot of the method is not shown, only the cruxial parts related to the DNS table. The first part of the method is about identifying files and choosing the newest received file. This means the database files are received in a Least In First Out (LIFO) style as it is designed in VBS. Then the chosen file is opened and subject to several checks to ensure it is not malformed. These checks includes retrieving the client ID and database version. Then for all flows which is in the SQLite database, all packets, DNS and application which is associated with the flow is retrieved and inserted into the server database. A flow from the client can be splitted into several flows, so here it is needed to identify whether the flow in the SQLite database is new or a continuation of an excisting flow. When all this is sorted it is time to insert the DNS entry from the SQLite database in the server database. In the end if everything went well all changes are committed and the SQLite databse is closed and the file is deleted. If an error occurred in the process of parsing the SQLite database into the server database, the server database is rolled back, so it is avoided to malform the server database.

Code snippet 5.13: DataExtractor processSQLite method.

```
1   ...
2   // Open the received fdb file with the SQLite database.
3   db = new SQLiteDB(currentFile.getCanonicalPath());
4   ...
5   // Insertion of client id and various checks on the sqlite database file is omitted
        here.
6   ...
7   // Start a transaction. This forces the database not to apply all the changes before
        the SQLite file is completely parsed.
8   stmt.executeUpdate("START TRANSACTION;");
9   ...
10  // For all flows in the SQLite database.
11  for (int i = 0; i < resFlows.size(); i++) {
12    resRow = resFlows.get(i);
13    ...
14    // Here it is decided whether the flow is a continuation of an other flow, and the
          new ID is therefore retreived. If it is a continuation the flow in the database
```

```
        is updated, if not a new is saved.
15      ...
16      // Get the DNS entry which is associated with Flow
17      ArrayList<Object[]> resDNS = db.executeQuery("SELECT ip, domain, ttl, responses
            FROM DNS WHERE flow_id = " + oldFlowId + ";");
18      // If there was found a DNS entry which matched, insert its information to the
            server database.
19      if (resDNS != null) {
20        for (int g = 0; g<resDNS.size(); g++) {
21          resRow = resDNS.get(g);
22          stmt.executeUpdate("INSERT IGNORE INTO DNS(flow_id, ip, domain, ttl, responses)
                VALUES(" +
23          newFlowId + ", '" +
24          resRow[0] + "', '" +
25          resRow[1] + "', " +
26          resRow[2] + ", " +
27          resRow[3] + ")");
28        }
29      }
30      ...
31      // Close and delete the SQLite database and commit the changes.
32      db.close();
33      currentFile.delete();
34      stmt.executeUpdate("COMMIT;");
35      ...
```

With both the client and server designed and implemented it is time to test whether the system fulfills the requirements.

Chapter 6

Acceptance test

In this chapter the acceptance tests described in chapter 3 will be conducted. The acceptance tests are designed to test whether the system requirements are fulfilled in the implemented design.

6.1 Method

The accceptance tests are designed to test whether the system is fulfilling the requirements set in chapter 3. According to the V-model system development model, the acceptance tests is the feedback to the requirements. The V-model also shows how the system is decomposed during the design and at its smallest bits and implemented and gradually built together. During the build of subcomponents to components and eventually to the system a combination of white-box testing and black-box testing should have been done o verify that each component and interface is as intended, also when being build together. White-box tests involves inspecting the code, whereas the opposite black-box is observing the output from when applying a specified input. This has somewhat been done during the implementation, but not shown, for the parts in the client which involves the `DNSMonitor` part. It has however not been done with the existing VBS foundation. These acceptance tests are black-box tests for the system components client and server and together as a system.

For each acceptance test a test procedure is specified where the input, expected output and actual output is defined and presented. The expected output is checked with the actual output to judge whether the test is parsed. If it is parsed the requirements tested in the acceptance test is fulfilled.

6.2 Scalability test

This acceptance test will test the scalability requirements set for the system. The test is described in section 3.4 and states that the system should be able to handle at least 3 concurrent clients. The 3 client should experience a worst case situation defined as acquiring data from a file transfer with a mean data rate between 1.2 $\frac{\text{MB}}{\text{s}}$ and 1.8 $\frac{\text{MB}}{\text{s}}$. This system should cope with this situation for at least 10 min.

The test is carried out by deploying three clients. The file transfer is initiated with the program `scp` which is a secure file transfer program using TCP. All clients are deployed on the same network and the file transfer is from the same server. This is done in an attempt to equalize the file transfer speeds between the clients. The clients are deployed on different 3-5 year old commercial computers where two are laptops (running eventually Windows and Linux) and one is a desktop (running Windows). The client is started on all computers after they all have initialized the file transfer. All clients and the server log the file transfer and receive and processing events. This makes it possible to check that whether all clients has

successfully transmitted their files and the server has received and processed all the received data correctly.

The expected result from the acceptance test is that no or only very few database files from the clients are malformed. The packets are transferred with TCP which should provide necessary retrainsmissions to avoid malforming the database. However it is unknown how the TCP server and client is configured and how sensitive they are to lost packets.

The test was started at 20.04 o'clock and ended again at 20.16 giving it a total of 12 min. The log files from the client and the server was carefully investigated and checked that it was successfully transmitted from the client, successfully received on the server and successfully processed on the server. The results are summarized in table 6.1 and an example of log file tracking can be seen in code snippet 6.1.

Client id	Mean download rate	Transmitted files	Received files	Processed files	Loss
Windows Laptop	1.26 $\frac{\text{MB}}{\text{s}}$	42	42	42	0 %
Windows Desktop	1.66 $\frac{\text{MB}}{\text{s}}$	46	46	46	0 %
Linux Laptop	1.26 $\frac{\text{MB}}{\text{s}}$	40	40	40	0 %

Table 6.1: Results form scalability acceptance test

Code snippet 6.1: Windows laptop client to server file tracking.

```
1  // On the client the following line is found.
2  FINEST|5012/0|Service VBSclient|15-05-24 20:06:35|2805729 [DataTransmitterThread]
       INFO  dataTransmitter.FileTransferClient  - File [data_client\data1432490794234.
       fdb] from clientId [146702c1-1431090364172] of size [1075200 bytes] succesfully
       uploaded in 1412 ms
3
4  // On the server the corresponding receiving file is identifies.
5  FINEST|18045/0|Service VBSserver|15-05-24 21:06:35|5426116 [FileReceiverThread] INFO
       vbsServer.FileReceiver  - Sucessfully received file from client with ID [146702c1
       -1431090364172] and renamed to data1432490794498.fdb
6  ...
7  // And eventually processes and inserted into the server database.
8  FINEST|18045/0|Service VBSserver|15-05-24 21:06:59|INFO: [sqlite] DB[195]:
       instantiated [/opt/bfd_server/app/data_server/data1432490794498.fdb]
9  FINEST|18045/0|Service VBSserver|15-05-24 21:06:59|INFO: [sqlite] DB[195]: opened
10 FINEST|18045/0|Service VBSserver|15-05-24 21:06:59|5449289 [DataExtracterThread] INFO
       vbsServer.DataExtracter  - Inserted 9 flows from client id 4
11 FINEST|18045/0|Service VBSserver|15-05-24 21:07:04|INFO: [sqlite] DB[195]: connection
       closed
```

The expected output was none to a few lost or malfunctioned database files, but during the test no database files was either malfunctioned or lost and the 0 % loss is lower than accepted threshold of 1 % meaning the acceptance test is parsed. This means that the requirement Sy1 and Se1 is fulfilled.

6.3 Data acquisition

This acceptance test is testing the system ability to acquire the defined data including some which are non topology, protocol and time dependent. Further it tests the clients ability to function on different platforms and transmit data automatically and function on Linux and Windows. The requirements tested are; C1, C3, C4 and C5.

The test is defined to be done by deploying the client on 5 computers (at least one is running Linux) for a 14 day long period. By inspecting the acquired data in the database the requirements C1, C3, C5 can be checked. To test requirement C4 it is defined that a client should have blocked access to the server for a period of 2 h. Upon reallowing the client access to the server it is observed whether the clients transmits its acquired files created in the period where the connection was blocked.

The expected output of the test is first that there has been acquired flows from all 5 clients in the 14 day period. The quantity of data activity depends on the usage of the computers, so no prior assumption are made upon the quantity of flows acquired or when the computers are used online. Then it is expected that the server database contains all the defined data which is acquired on the client. Eventually it is expected that the client which is blocked access to the server for a period, will reconnect and transmit its acquired data upon reallowed connection to the server.

The test was done in the period between the 10'th of May 2015 and the 24'th of May 2015. During this period 5 clients was deployed on 5 different computers for both work and home appliance. The computers are located in different parts of northern Jutland in Denmark. In figure 6.1 the activity for each client is shown as a histogram with bins for every 12 h duration, showing each client flow occurrence rate in the 14 day period. In the activity figure "client 2" is a client running on a Linux computer and the rest is Windows computers. Some have joined later in the period than others.

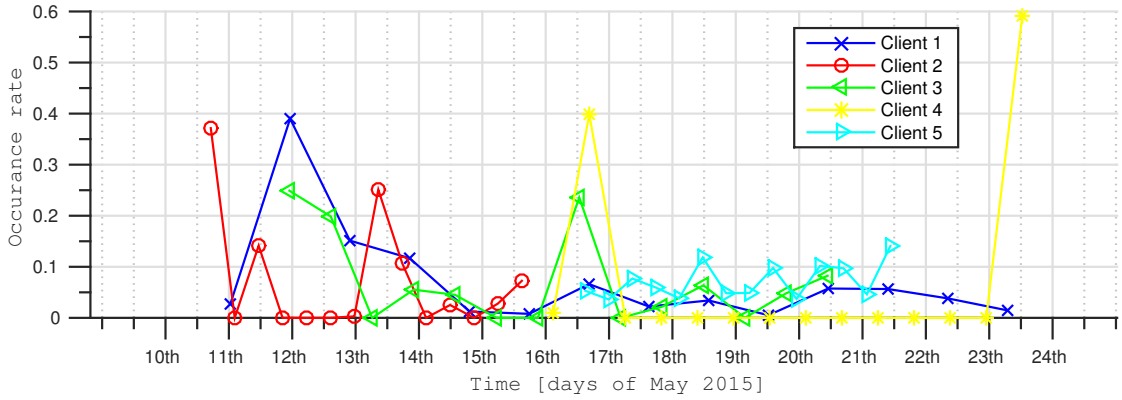

Figure 6.1: Recorded flows during a 14 day period.

Requirement C4 was tested with a client running on a Linux computer. The server IP address was blocked by access using iptables. The test began at 27'th of May 2015 at 10.00 and ended at 12.00 the same day. Prior to removing the block one of the created `fdb` files was opened with `SQLite3` and the timestamps from a few of the flows was checked and found to be within the blocked timeframe. Posterior

to removing the block, it was inspected that the client transmitted all the `fdb` files, which it was and the test is therefore passed and the requirement fulfilled.

In the database the acquired data is acquired and stored as intended. An example below shows a flow and the information stored in the different tables in the database. This flow is captured and given ID 490. A DNS packet is acquired and associated with the flow and it is saved in the DNS table. The process name is also acquired and stored in the application table. Data from the tables assiciated with this flow ID is shown in code snippet 6.2 which is taken directly from the server database.

Code snippet 6.2: A flow acquired by the system and retrieved from the server database.

```
// Flow table entry (table is too long to be in one line).
+---------+-----------+------------------+----------------+----------------+ ...
| flow_id | client_id | start_time       | local_ip       | remote_ip      |...
+---------+-----------+------------------+----------------+----------------+...
|     490 |         2 | 1431102946486347 | 192.168.43.236 | 54.192.130.173 |
+---------+-----------+------------------+----------------+----------------+...
------------+-------------+---------------+-------------+-----------------+
 local_port | remote_port | protocol_name | tag         | application_id |
------------+-------------+---------------+-------------+-----------------+
     49472 |         443 | TCP           | default_tag |               6 |
------------+-------------+---------------+-------------+-----------------+

// Packets (only the first 3 packets are shown. Some of the TCP flag columns. The table
    was too long to be shown on one line and is therefore splitted).
+-----------+---------+-----------+-------------+--------------+------+------+...
| packet_id | flow_id | direction | packet_size | payload_size | SYN  | ACK  |...
+-----------+---------+-----------+-------------+--------------+------+------+...
|    306428 |     490 | O         |          52 |            0 |    1 |    0 |...
|    306429 |     490 | I         |          52 |            0 |    1 |    1 |...
|    306430 |     490 | O         |          40 |            0 |    0 |    1 |...
+-----------+---------+-----------+-------------+--------------+------+------+...
------+------+...+------------+------------+--------------------+
 PSH  | FIN  |...| ACK_number | SEQ_number | relative_timestamp |
------+------+...+------------+------------+--------------------+
    0 |    0 |...|          0 | 3630356392 |                  0 |
    0 |    0 |...| 3630356393 | 3514311552 |              81201 |
    0 |    0 |...| 3514311553 | 3630356393 |                178 |
------+------+...+------------+------------+--------------------+

// DNS
+--------+---------+----------------+-----------------------+------+-----------+
| dns_id | flow_id | ip             | domain                | ttl  | responses |
+--------+---------+----------------+-----------------------+------+-----------+
|    532 |     490 | 54.192.130.173 | client-cf.dropbox.com |  17  |         6 |
+--------+---------+----------------+-----------------------+------+-----------+
// Application
+----------------+------------------+
| application_id | application_name |
+----------------+------------------+
```

```
39 |               6 | dropbox          |
40 +-----------------+------------------+
```

The data acquired is not assuming any protocol or topology. Some of the data is not time dependent, such as the packet size. The wanted data to be acquired is and which is actually acquired is summarized in table 6.2.

Data specified to be acquired.	Is data acquired by the system?
Packet length.	Yes.
Packet header length.	Yes.
Packet direction.	Yes.
Packet timestamp.	Yes.
TCP flags indication.	Yes.
TCP sequence and acknowledgement numbers.	Yes.
DNS response TTL.	Yes.
DNS quantity of answers.	Yes.
DNS domain level depth.	Yes.
Some topology independent data.	Yes.
Some protocol independent data.	Yes.
Some time independent data.	Yes.

Table 6.2: Results from data acquisition acceptance test.

As expected the acceptance test revealed flows had been acquired from all 5 clients as seen in figure 6.1. Further the system acquires the expected data and correctly placed it in the server database as seen in table 6.2 and code snippet 6.2. The client which was block access to the server for 2 h period had no problem in transmitting its finalized database file upon reconnecting to the server and the files contained acquired data from the period. This means that the actual output of the test and the expected output is identical and the acceptance test is parsed. It is concluded that the requirements C1, C3, C4 and C5 are fulfilled.

6.4 User interaction

The last acceptance test is about testing the client reaction time to user input. The requirement states that an experiment should be conducted a 100 times where each experiment instructs the client to push its data to the server after a random period. The period is a random uniformly distributed value between 30 s and 90 s. Background traffic is initiated during the experiments to ensure the client has data to acquire.

The expected output from this acceptance test is a CDF showing a maximum value of 29 s as this is the value the system is designed to and a 90 % which is slightly lower.

This experiment has been done using a python stript, which for every minute starts a new experiment. It starts by drawing a random number of seconds to wait, waits and then sets the "push" flag in the custom file on the client and logs the time. For every 10 s the scripts pings google to create background data. The log from the client is collected and correlated with the log file from the python script to identify the reaction time for each experiment. An example of log file collection can be seen in code snippet 6.3. A CDF is made with the reaction times from the 100 experiments and used to identify the 90 % threshold. The CDF is found in figure 6.2.

Code snippet 6.3: Log files from python script and the client, here the reaction time was 9 s.

```
1 // Python script setting the ‘‘push’’ flag
2 8, 2015-05-25 12:55:34.258553
3
4 // The client registers the flag
5 INFO|16977/0|VBSclient|15-05-25 12:55:37|477568 [main] INFO  vbsClient.VbsClient  -
    Pushing data to server
6 ...
7 // The client has starts transmitting the created database file.
8 INFO|16977/0|VBSclient|15-05-25 12:55:43|484231 [DataTransmitterThread] INFO
    dataTransmitter.FileTransferClient  - File [data_client/data1432551342939.fdb] from
    clientId [145702bd-1432488568265] of size [26624 bytes] succesfully uploaded in 47
    ms
```

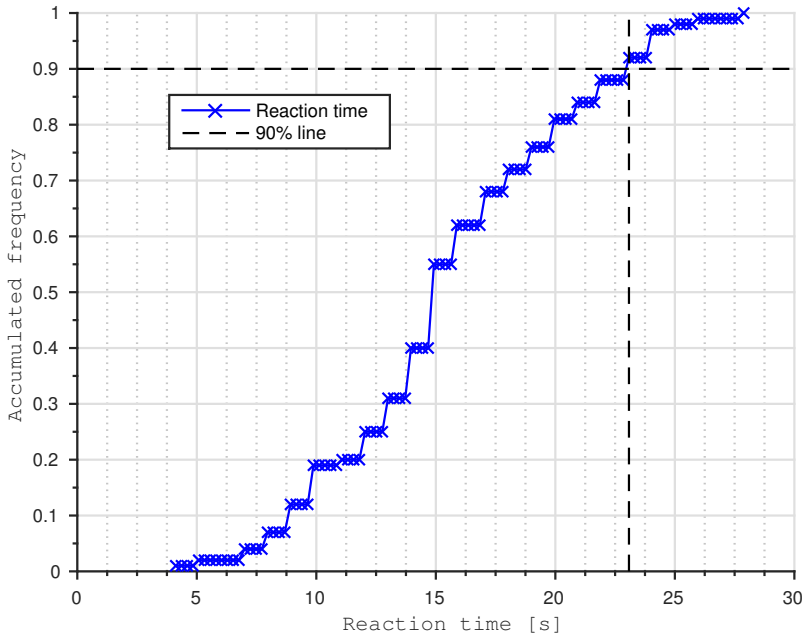

Figure 6.2: CDF of the clients reaction times with 90 % lineup.

From the CDF the 90 % value is found to be 23 s, meaning that 90 of the 100 reaction times was below 23 %. This is below the required threshold of 30 s. The acceptance test is therefore parsed. This means that the requirement C2 is therefore fulfilled. It is notices that the maximum value is slightly less but

close to 29 s which suggests that the design matches the reality and that the only reason the maximum expected value is not directly observed is due to the stochastic nature of the experiment.

6.5 Summary

In this section the results from the acceptance tests will be summarized. Table 6.3 summarizes the each requirements, which acceptance test what tested it and whether it was found to be fulfilled. All requirements was found to be fulfilled by the acceptance tests.

Requirement	Acceptance test	Fulfilled?
Sy1.	Scalability.	Yes
C1.	Data acquisition.	Yes
C2.	User interaction.	Yes
C3.	Data acquisition.	Yes
C4.	Data acquisition.	Yes
C5.	Data acquisition.	Yes
Se1.	Scalability.	Yes

Table 6.3: Summary of acceptance tests

Chapter 7

Detection

In this chapter the designed, implemented and tested data acquisition system is used in a detection system targeting bot-malware. This is done in order to get a feeling of what can be achieved in detection systems by using this data acquisition system. This involves designing and implementing a containment environment for containing malicious bot-malware samples, procedures to conduct experiments, a feature extractor and eventually the detector.

7.1 Containment environment

The containment environment is intended to minimize the risk that the malicious software harms external systems. Executing samples of bot-malware includes a lot of issues and challenges, but it is needed in order to acquire malicious data for this data acquisition system.

Containment of malicious software as bot-malware is done by different methods according to how security and goodness of data is weighted. Bot-malware is assumingly a part of a botnet and communicated with it using its C&C channel to receive instructions of what to do, tell it is alive, transmit acquired/stolen data etc. All this is through the Internet, so the bot-malware is dependent on the Internet to function. If however the bot-malware was allowed unrestricted access to the Internet, it would potentially be capable of full functionality including taking part in malicious activities which might harm external systems. The opposite weight is fully contain to bot-malware, meaning it will have no access to the Internet at all. The risk of the bot-malware to harm external systems would then be zero, but the bot-malware would not be able to contact its botnet with the C&C channel and therefore not function properly. It could be considered to utilize emulation of the Internet as an alternative, but it was found that this has a too large effect on data used for e.g. statistics of flows. The reason is emulation services limitations in diversity of services and responses to e.g. HTTP requests. It is chosen that bot-malware should have access to the Internet, but not too much. The right balance between risking to harm external systems and avoid containing too much so the bot-malware detects it and evades.

The containment policy used in this project will be to highly restrict the bot-malware access to the Internet through a proxy. This is very similar to the solution used in [Rossow et al., 2011]. This proxy acts as a safety component between the live bot-malware on a infected host and the Internet. Further it will be able to provide real time information during the experiments and at any time block the bot-malware from accessing the Internet. The monitoring is done by the program tcpdump [Jacobson et al., 2013]. The restrictions consists of general packet rate (one for all and for TCP, UDP and ICMP), a tokenbucket and a open connection limitation. The values of these restrictions are found by empirical measurements and are adjusted such that the host Internet browser can actually access a website, but also limit the traffic such that it can be visually supervised with tcpdump on the proxy.

- Max TCP SYN, ACK, FIN, RST packet rate of $10\,\frac{1}{\mathrm{s}}$.
- Max ICMP packet rate set to $5\,\frac{1}{\mathrm{s}}$.
- Max active flows limited to 15.
- Max UDP packet rate is set to $10\,\frac{1}{\mathrm{s}}$.
- Max packet rate is set to $20\,\frac{1}{\mathrm{s}}$.
- Token bucket set to a mean rate of $1\,\frac{\mathrm{Mbit}}{\mathrm{s}}$ and allows peaks of $2\,\frac{\mathrm{Mbit}}{\mathrm{s}}$ and bursts of $100\,\frac{\mathrm{kB}}{\mathrm{s}}$.

In the contained environment a local area network consisting of three computers is deployed; a server, proxy and a host. The server is intended to run the proof of concept data acquisition server. The host which will run the client and is intended to be infected with bot-malware will run Windows 7 SP1. The proxy and server will both run the Linux Debian Wheezy 6.1 stable.

Infecting the host requires a method of cleaning and resetting it, as it is a main component of the experiments. As it is chosen to use a bare-metal host and not a virtualized host, an external OS is needed to reset the host. On a USB-drive a Linux operating system is installed which can be used to save a image of the harddrive containing the Windows harddrive before it is infected. This image can later be used to remedy the harddrive. The use of a bare-metal host instead of a easier to deploy and maintain virtualized host is done as an attempt to avoid risking the bot-malware to detect it is being monitored. The containment system with all its components is illustrated in figure 7.1.

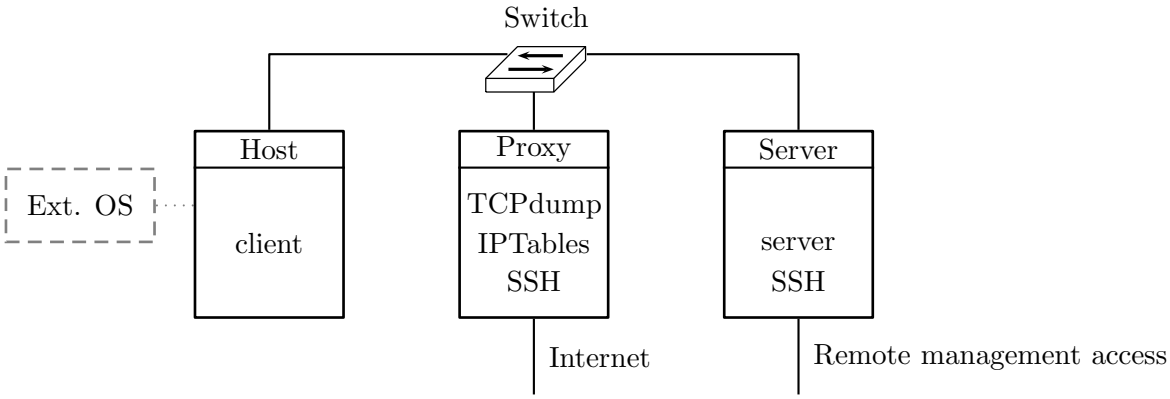

Figure 7.1: Containment environment.

7.1.1 Experiment procedure

A experiment procedure is needed to be defined in an attempt to provide each sample with a comparable foundation. An experiment includes the execution of a sample, either malicious or benign. Prior to experiments the contained environment is configured and it is ensured that the containment rules is properly applied. Below is listed the steps of the experiment procedure.

1. Boot the external OS and remedy the host harddrive. Disconnect the external OS drives prior to booting the host to avoid risking the bot-malware spreads into it.
2. Start the server.
3. Start tcpdump on the proxy.

4. Stop the client and set the "tag" of the experiment in `custom.cfg`.
5. Copy the sample to execute from a USB-pen to the host.
6. Start the client and execute the sample.
7. Wait 10 min.
8. Set the "push" flag in `custom.cfg`, telling the client to push its acquired data to the server.
9. Shutdown the host.

These steps has been automated with python scripts for starting and stopping scripts on the host, remedy the host on the external OS and to apply the containment rules on the proxy.

7.2 Labeling

It is important to perform an accurate labeling of the flows as the detector is in risk of being trained with incorrect labeled data. Incorrect labeled data used for training the detector could lead to a bad performing detector in the real world, with a lot of false positives or false negatives. To label the acquired flows it will be assumed that background flows will be present alongside the either benign or malicious flows. These background flows are created by other programs on the host such as parts of the operating system.

To identify background flows experiments with no sample is conducted. These background experiments will be conducted prior, in between and after a series of experiments in an attempt to catch flows that are appearing due to the change of time. This is likely to happen as time will be stuck on the host which is remedied for every experiment, but time is not static on the external systems on the Internet contacted by the processes on the host.

Flows captured in background experiments are all assumed to be background flows which needs to be identified and removed from the benign and malicious experiments. It is therefore needed to be able to reidentity background flows across experiments so they can be removed. The flows cannot be identified by usual five-tumble-key as this is a unique identifier and can therefore not be used to identify a flow which is the "same" in terms of purpose, but is unlikely to have the same five-tumble-key. In this project is it chosen to use a three tumble key consisting of the most likely items in the five-tumble-key to stay the same across time. The five tumble key consists of local and remote ports, local and remote IP and protocol. The remote IP could change due to load spreading and the port in the initiating end of the flow is most likely randomly chosen. A hash key is therefore generated for each background flow by using the remote IP, port of the receiving end and the protocol. The alternative to this three tumble key would be statistically matching, but this does not guarantee a perfect filtering either as the content of the flow might differ over time. A thought example of a flow which change content over time is a check for software updates. One day the response is no update available and the other day there is a update which is transmitted.

From 3 background experiments 22 unique flows were identified. Any flow from malicious or benign experiments that matches the three tumble key of one of the background flows three tumble keys will be

filtered and removed from the dataset prior to feature extraction.

7.3 Malicious and benign samples

Samples of malicious (bot-malware) and benign software is needed in order to make them generate data to be acquired by the data acquisition system. Bot-malware are named in a lot of different naming conventions. In order to find useful bot-malware samples, a top ten botnet threats from 2014 was used the found names as a basis to search for samples. 11 working bot-malware samples were eventually found. Their names and acquires flows are summarized in table 7.1. The chosen benign samples were portable software are summarized in table 7.2.

Bot-Malware sample name	Acquired flows	Flows left post labeling
mazben.3_2012	8	5
mazben.1_2012	6	5
dorkbot.IRC_Maximus_2012	172	171
dorkbot.AB_2012	78	74
waledac.3	11	7
Trojan-Spy.Win32.Zbot.aad	11	8
Trojan-Spy.Win32.Zbot.zzy	11	7
Trojan-Spy.Win32.Shiz.l	10	9
ngrbot-12	42	39
ngrbot-9	13	7
gamarue-6(andromeda))	11	7
Total	373	339

Table 7.1: Malicious samples and the acquired flows prior and posterior labeling.

Benign application name	Acquired flows	Flows left post labeling
Mozilla Firefox	78	77
Mozilla Thunderbird	31	30
uTorrent	448	444
eMule	11	8
CCleaner	24	23
Filezilla	11	5
GIMP	11	8
Total	614	596

Table 7.2: Benign applications and the acquired flows prior and posterior labeling.

7.4 Feature extractor

With the data acquired by running experiments for each sample it is needed to process the data into features which can be used by the detector. It is chosen to use the Weka toolbox [university of Waikato, 2014] to apply the detector. The input to Weka is the features in a comma separated value file format (`csv`) file with all features including the label row wise. The feature extractor will have to produce such that a file as its output. The feature extractor acquires the captured data associated with each flow tagged to be used in the training set.

The chosen feature set is listed below. The list of features contains flow statistical descriptors, RTT features and TCP flag usage features, as well as the process name and the DNS domain level depth, TTL and responses. There are not used any features which can be time evaded, except from the RTT.

- Total quantity of packets per flow.
- Total quantity of bytes per flow.
- Mean bytes per packet in the flow.
- Standard deviation of bytes per packet in the flow.
- Packet input / output ratio.
- Bytes input / output ratio.
- Total quantity of bytes sent in the flow.
- Total quantity of bytes received in the flow.
- Percentage of TCP [psh, syn, ack, psh+ack, rst] flags used in the flow.
- Mean RTT.
- Standard deviation of RTT
- DNS resolved domain TTL.
- DNS domain level depth.
- Hash of the process name.

Most of the flow features are straight forward to calculate, as they are ratios and simple statistics such as mean and standard deviation. However, calculating the RTT is not so straight forward as it needs to be estimated. The RTT is choseo to be estimated from the TCP handshake using a passive tcp RTT presented in [Jiang and Dovrolis, 2002] and during the flow with another passive RTT estimator which is presented in [But et al., 2005].

Some of the chosen features probability distribution for each dataset (malicious and benign) has been visualized in the histograms below. Figure 7.2 shows a histogram of the packet input and output ratio for the benign and malicious dataset. These distribution for each dataset are not too different and reveals that most packets are generally more likely to be transmitted than received on the host. Inspecting the byte input output ratio shown in figure 7.3, reveals that even though most packets are outgoing rather than incoming, the majority of data is incoming. An explanation could be that outgoing packets are mainly requests or acknowledgements and incoming packets contains data.

Examples of features that, in this dataset, does not provide good discriminate information for the detector are the total bytes per flow (figure 7.4) and the mean RTT (figure 7.5). Both have histograms which for benign and malicious datasets are very equal. There does however seem to be some difference in the usage of flags in the TCP flows. Figure 7.6 shows the histogram for usage of reset (RST) flags in TCP flows and figure 7.7 shows the usage of acknowledgement (ACK) flags. The RST flag seems to be more often used per TCP flow for malicious than benign. This is backed by the observation that the occurrence of short flows is greater for the malicious samples than the benign. The histogram of packets per flow is not shown, as it is very similar to the bytes per flow. The use of ACK flags in packets in TCP flows seems to be relatively similarly distributed in both malicious and benign flows.

Figure 7.8 shows the histogram of process name hashes. This seems to be a good discriminative feature for these datasets. There are however overlaps which could be `svchost` which is a generic process name used by system network processes on Windows and Internet explorer (`iexplorer.exe`). Both candidates as process names which malicious samples can use to hide their presence. It is observed that the majority of the benign samples use their application name in the process which creates the flows and that malicious samples often don't. This is therefore also a features which could lead to optimistic classification as the names overlaps or the classifier could be trained to detect `iexplorer.exe` as malicious.

Figure 7.9 shows the histogram of a feature which is calculated, but not used by the detector, which is the flow duration. This is a time dependent feature and shows a huge difference between benign and malicious flows flow duration. The benign flows are all shorter than the malicious flows. These histograms does probably not give a representative probability distribution of the flow duration of malicious and benign applications.

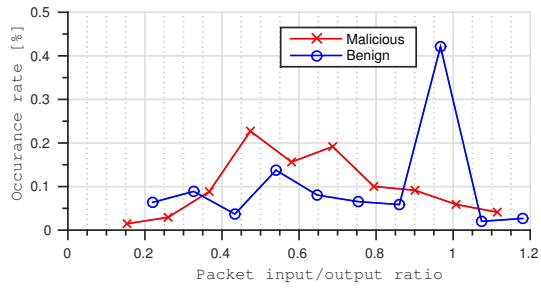

Figure 7.2: Histogram of packet input/output ratio.

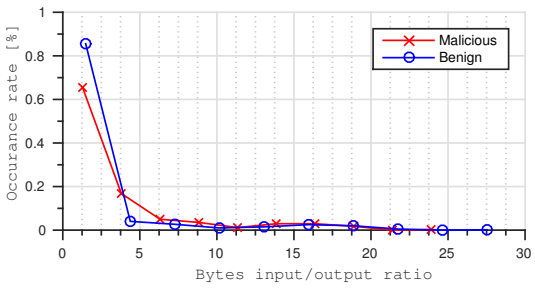

Figure 7.3: Histogram of byte input/output ratio.

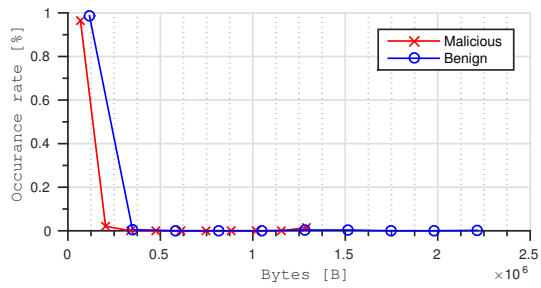

Figure 7.4: Histogram of total bytes per flow.

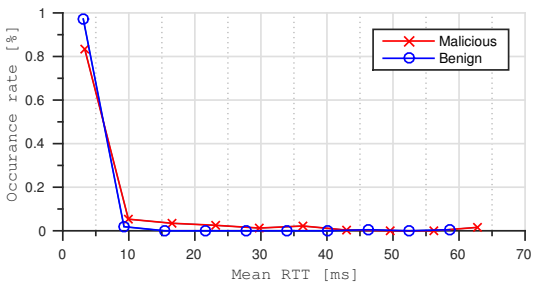

Figure 7.5: Histogram of mean RTT.

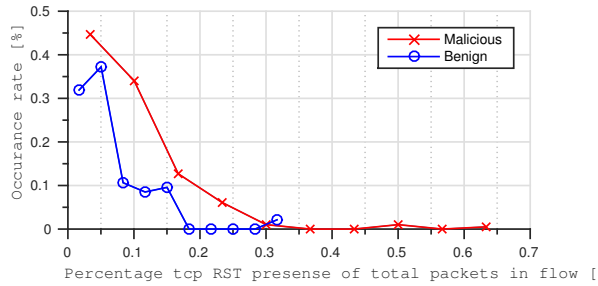

Figure 7.6: Histogram of TCP rst usage.

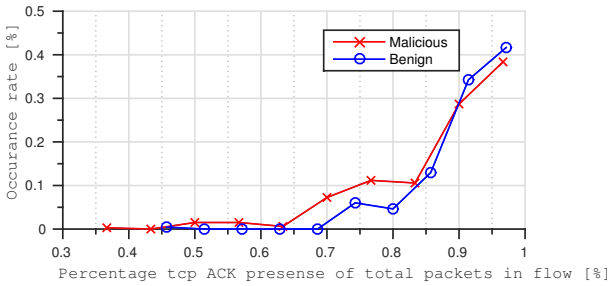

Figure 7.7: Histogram of TCP ack usage.

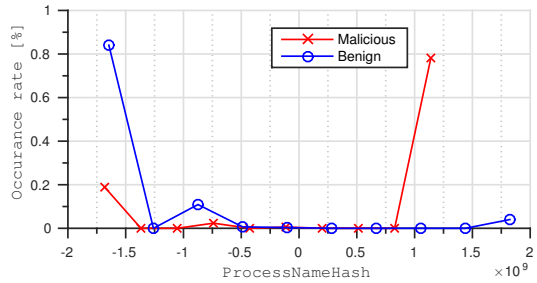

Figure 7.8: Histogram of processname hash.

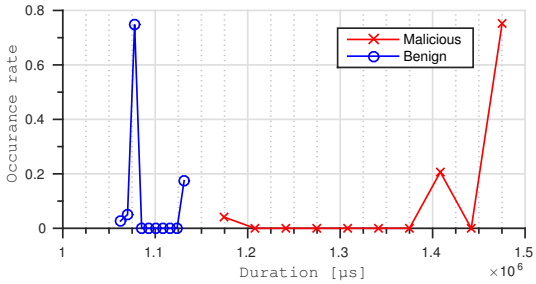

Figure 7.9: Histogram of flow duration.

7.5 Detector

The detector chosen is the C4.5 decision tree which has been used in several previous occasions such as the VBS classifier [Bujlow et al., 2011] and in [Stevanovic and Pedersen, 2014, Masud et al., 2008]. In Weka the C4.5 algorithm is implemented in Java and called J48 and is based on the work presented in [Quinlan, 1993].

The C4.5 detector builds its decision tree by concept of information entropy. The entropy can be considered as the impurity of the dataset. If the entropy is 1, the dataset is completely random splitted and if the entropy is 0 the dataset is perfectly splitted, meaning the classes are perfectly separated. The algorithm is greedy meaning that it, for every split needed to be made, will select the attribute which lowers the entropy of the dataset to be splitted the most. It will continue to add splits until the dataset is pure, meaning that all the classes are completely separated with the tree.

7.6 Results

In this section the results from testing and training the detector will be presented. The dataset is splitted by randomly chosen 66 % of the samples for training and another 34 % for testing. The results from the testing is presented as a confusion matrix which can be found in table 7.3 and a set of performance metrics which is the accuracy, recall and precision.

The confusion matrix shows the relation between the sample label and the detected label. This gives four groups which are:

- True Positivies (TP) which is correctly detected malicious samples.

	Detected as malicious	Detected as benign.
Malicious.	119 (TP)	5 (FN)
Benign.	14 (FP)	180 (TN)

Table 7.3: Confusion matrix from the trained and tested C4.5 classifier.

- True Negatives (TN) which is correctly detected benign samples.
- False Positivies (FP) which is benign samples detected as malicious or false alarm (type I error).
- False Negatives (FN) which is malicious samples detected as benign meaning a miss (type II error).

To further evaluate the performance of the detector, its accuracy, recall and precision is calculated. Accuracy is a measure of how well the classifier is able to correctly detect. Recall is a measure of how sensitive the detector is to correctly detect malicious samples and create type II errors. The precision is a measure of how often the detector detects malicious samples correctly from its total quantity of positive detected samples. The results are presented below.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = 0.94 \tag{7.1}$$

$$Sensitivity = \frac{TP}{TP + FN} = 0.96 \tag{7.2}$$

$$Precision = \frac{TP}{TP + FP} = 0.89 \tag{7.3}$$

These results suggest that the trained detector is a fairly good detector. However, the dataset is small and might not be representative. The results does prove that the data acquisition system can be used successfully together with a detection system and the result can be a good performing detection system.

Chapter 8

Discussion

In this chapter the project is discussed in aspects of how well the problems presented in the analysis are answered and how well requirements are fulfilled. Further, possible improvements and suggested future work is discussed.

Botnets are one of the most serious threats to Internet security today. In order to defeat them it is needed to detect them, which is done with detection systems. The detection system performance is dependent on its date acquisition system. The problem with present detection systems was found to be that that they are not both scalable and collaborative. The initiating problem was formulated as follows:

"How to improve data acquisition systems for bot-malware detection systems?"

Through investigations of the challenges caused by botnets and analysis of present data acquisition systems, it was found that a data acquisition, which fulfills the following functionalities and characteristics, would improve present data acquisition systems:

- Be scalable.
- Be collaborative meaning it can acquire data from multiple data domains and use multiple devices to collaborate on the detection of botnets.
- Avoid acquiring data which bot-malware can evade and manipulate.
- Acquire topology and protocol independent data.

This problem definition form the basis of the requirement chapter 3, which defines the system to be designed and implemented. In the requirements a proof of concept system was defined, aiming to prove that it is possible to improve present data acquisition systems. In the requirement chapter a set of requirements are defined for the system and its two components; client and server. Further, the requirement chapter contains the definition of how the requirements should be tested with a set of acceptance tests.

In the design chapter 4, the system is chosen to be designed with the system VBS as a foundation. This means that the design started with a description of the basic understanding of VBS, before it was derived which changes had to be made for the system to fulfill all the requirements. This included the design of a DNS monitor extension and adaptation of VBS to support the required data and listen to user input.

In chapter 5 the system was implemented. The intention of this chapter was to build the system, but also demonstrate how VBS is implemented. With the system designed and implemented it is ready for its acceptance tests which will show whether the system fulfills the requirements which is the needed in order to prove the concept.

The first acceptance test validated the system scalability requirements. The requirements stated that system should not break in the stress test and that the server should not loose more than 1 % of packages of acquired data sent from the clients, under the experiment with 3 worst case clients connected. The test results was a loss rate of 0 % and no signs of errors.

The second acceptance test is testing the system capability to acquire the defined data, function on a longer period of 14 d and the clients ability to acquire data on multiple platforms. Further, the clients ability to acquire data without having a connection to the server is also tested in this acceptance test. The results from this acceptance test show that the system works on multiple platforms and is capable of acquiring the defined data.

The last acceptance test is about the client reaction time. An experiment was therefore conducted 100 times where the client was instructed to transmit its acquired data. For each experiment the time from the client was instructed to it transmitted the file was calculated and the 90 % threshold was found to be 23 s well under the requirement of 30 s which means the requirements is fulfilled.

In the next section it will be discussed how well the problems are answered.

8.1 Problem fulfilling discussion

The verdict of the designed and implemented system is summarized in summarized in table 8.1.

Criteria	Verdict	Comment
Scalability	+	The system distributed design means it is scalable.
Collaborability	+	The system acquires data from multiple data domains and is from multiple hosts.
Vulnerability to evasion	(+)	The system does acquire time dependent data, but can be chosen not to be used by a detector.
Independence of protocol and topology	+	The system does not assume that the C&C channel uses any specific topology or protocol.

Table 8.1: Designed and implemented system verdict.

The verdict of the designed and implemented system compared with the other discussed present data acquisition systems are summarized in table 8.2. The designed and implemented system fulfills the four main characteristics of a data acquisition system, which was found to be needed to improve present data acquisition systems. The proven system is better than present data acquisition systems by being both scalable and collaborative as well as less vulnerable to evasion. The proven system also showed that it was fulfilling the desired functionalities and characteristics listed in the problem definition.

The system though leaves room for improvements. The following section will go through the future work on the project starting with potential improvements for the data acquisition system.

Criteria	Botminer	Correlating log files	Suspicious groups	EFFORT	System
Scalability	-	+	-	+	+
Collaborability	*	-	+	*	+
Vulnerability to evasion	-	-	*	-	(+)
Independence of protocol and topology	+	+	+	+	+

Table 8.2: The designed and implemented system compared to present data acquisition systems.

8.2 Future work

In this section the following potential improvements to the system will be discussed.

- Expanded platform support.
- Vulnerability to evasion.
- Improved collaboration.
- Improved validation.

Expanded platform support could be a part of the future work. It was found in the problem analysis that bot-malware is migrating to multiple platforms. This happens as the quantity of devices on the Internet is increasing. This could e.g. be mobile devices. The client could be expanded to support platforms as e.g Android or Mac OSX. The proof of concept system could be ported to these two platforms, as it is implemented in Java and the library, it uses to capture packets, JNetPcap is being developed to support these platforms.

The designed and implemented proof of concept data acquisition system is improving present data acquisition systems. It is though still a host based data acquisition system. This means that its presence can be detectable by bot-malware, if it knows what to look for. Bot-malware can potentially, if it is bot-malware with root access, attempt to remove the client or evade by time evasion, hide its own process name or change behavior, which would affect the data acquired by the data acquisition system. On the other hand, the client would also have root access, as it is required to capture packets, and could use it to hide from process lists etc. There cannot be a guarantee that a host based data acquisition will be immune to evasion techniques. The likelihood of the host based data acquisition system to be detected by bot-malware is related to:

- The popularity of the data acquisition system.
- The quantity of host specific data.
- The footprint left on the host.

The designed and implemented system is both scalable and collaborative. A key to this is its distributed structure where measuring probes are distributed. This could potentially also be achieved by distributing network based probes.

It is stated in several sources that novel bot-malware detectors are collaborative. This proof of concept is collaborative in case of acquiring data from multiple data sources, but also in case of acquiring data from multiple hosts. The proof of concept does however only acquire one host specific data which is the process name. The proof of concept system could be expanded to acquire further host specific data, but on the consequence of increasing the risk of being detected. Further research could therefore be done in analyzing which host specific data the system would benefit most of acquiring additionally.

The verification of the system is done mainly with acceptance tests. Acceptance tests are black box tests meaning the system is observed as a black box without knowledge to its content. Using mainly black box testing might result in missing vital internal errors. A concrete example could be lost packets in the system. To properly verify the system it would require a combination of white and black box testing of the system components as they are and gradually being assembled into larger components. This is also shown in the V-model. However, this procedure has been made more complicated due to the use of VBS as a foundation. The next iteration of the of the data acquisition system would involve deriving use cases to define the exact interaction with the system and its usage situations. Use cases and flow diagrams would form the skeleton and functionalities for the design. Having designed the entire system after the V-model would provide a better guarantee of the system performance.

8.3 Detection system discussion

In chapter 7 the proof of concept system was attempted use as part of a detection system. In this section the following will be discussed:

- Representative malicious datasets.
- Representative benign datasets.
- The dataset acquired.
- The results from the used detector.
- Risk of malicious samples to spread in the contained environment.
- Interaction with the Internet and labeling.

It is difficult to find usable malicious bot-malware samples. There are plenty available online, but due to the diversity of malware and the behavior of bot-malware they fall within different malware categories such as trojan, virus just to name a few. Further, the majority of the found samples where not executable on Windows 7 or seemed to be inactive upon execution. This could be due to evasion where the malicious sample is activated by user interaction or upon reboot. To take user interaction into the experiments would make repeatability increasingly difficult if it is not made automated. It was chosen to keep the experiments simple, assuming that a representative set of malicious samples could be assembled.

Representative benign data is another issue in bot-malware detection. Present methods used to acquire benign network datasets is to tab network edges, or local networks. This introduces questions of whether the users of these networks are representative and not infected. This system is a host based data acquisition system, meaning that to acquire benign data, the client would have to be installed on selected computers. How to choose the computers could depend on:

- Available computers.
- Representative by social standing, age etc.

It would not be possible to completely eliminate the risk of acquiring data from computers which are infected, if the computers are connected to the Internet. The risk can instead be minimized by eg forcing the use of antivirus software, apply a warning system to malicious websites, additional warnings before opening e-mail attachments or downloaded files. There is a trade-off between restrictions to minimize the risk of the computer to be infected and keeping the user of the computer still interested in using the system.

The dataset acquired in this project contained 339 malicious flows and 596 benign flows after labeling and filtering. The majority of the malicious flows was from two samples from the same family (dorkbot) for 72 % of all malicious flows in the dataset. In the benign dataset is also one application behind the majority, this is the torrent application uTorrent which is responsible for 74 % of all benign flows. This can potentially skrew the dataset.

The results from the trained and tested detector showed very good detection results. These should be taken lightly as the dataset presented clear signs that it might not be representative. This could be seen in features such as flow duration (which was however not used by the detector) which alone proved to be a perfect feature to classify malicious and benign samples. The results from the detector does however suggest that the data acquisition is capable of acquiring data which can be used to train and test a detector.

The setup of the containment environment leaves all three computers connected on the same network. This leaves the potential of a malicious sample to attack the proxy and the server computer. If a malicious sample did spread to these computers it would risk implicating other experiments. To minimize this risk all open services on the computers were protected by strong passwords. The data acquisition server uses three open sockets, one for file receiving, one for update of clients and one for registration of new clients. These services are not modified in this project, meaning they are as they where designed and implemented in VBS. VBS is not designed to cope with malicious attacks. A future redesign of the system should consider the security of the system.

Interaction from the Internet with the samples executed in the contained environment raises an issue with repeatability of the experiments. The host is remedied between each experiment, meaning that the host content is the same for every experiment. This means that the basis for each experiment is the same. The Internet cannot however be reset for each experiment. A concrete example of a application which changes over time is software updates such as Windows Update. To handle this in regard to labeling and filtering of the flows, experiments with no applications other than the operating system, was conducted regularly.

Chapter 9

Conclusion

In this chapter the conclusion of the project will be presented. The conclusion will start by presenting what problem this project investigates. Then the contribution of this project is presented.

Botnets are one of the most serious security threats to Internet security today. The distributed network, utilized by botnets, enables them to perform organized malicious activities which causes great Internet security concerns. Botnets are growing as an increasing quantity of devices gets online. Further, they are evolving to become increasingly complex and they are migrating to new platforms e.g. mobile devices.

The prerequisite to defeat botnets is to be able to detect them. Present detection systems are either host, network or hybrid based, meaning they are not both scalable and collaborative. Scalability and collaborability is found to be key characteristics needed to be fulfilled in order to improve present bot-malware data acquisition systems.

The contributions of the project are:

- Identification of present data acquisition systems main characteristics.
- Implementation and design of a proof of concept data acquisition system, which is capable of improving present data acquisition systems.
- Demonstration of the proof of concept data acquisition systems capability for detection systems.

Present data acquisition systems were compared on four main characteristics; scalability, collaborability, vulnerability to evasion and topology and protocol independence. None of the systems were both scalable and collaborative, and nearly all were found to be vulnerable to bot-malware evasion techniques.

The proven system is better than present data acquisition systems by being both scalable and collaborative as well as less vulnerable to evasion.

Demonstration proved, that the data acquisition system can be used successfully together with a detection system and the result can be a good performing detection system.

Chapter 10

Bibliography

[Bujlow et al., 2012] Bujlow, T., Balachandran, K., Hald, S., Riaz, M., and Pedersen, J. (2012). Volunteer-based system for research on the internet traffic. *TELFOR Journal*, 4(1):2–7.

[Bujlow et al., 2011] Bujlow, T., Balachandran, K., Riaz, M., and Pedersen, J. (2011). *Volunteer-Based System for classification of traffic in computer networks*, pages 210–213. IEEE Press.

[But et al., 2005] But, J., Keller, U., Kennedy, D., and Armitage, G. (2005). Passive tcp stream estimation of rtt and jitter parameters. In *Local Computer Networks, 2005. 30th Anniversary. The IEEE Conference on*, pages 8 pp.–441.

[Gu et al., 2008a] Gu, G., Perdisci, R., Zhang, J., and Lee, W. (2008a). Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 139–154, Berkeley, CA, USA. USENIX Association.

[Gu et al., 2007] Gu, G., Porras, P., Yegneswaran, V., Fong, M., and Lee, W. (2007). Bothunter: Detecting malware infection through ids-driven dialog correlation.

[Gu et al., 2008b] Gu, G., Zhang, J., and Lee, W. (2008b). BotSniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*.

[Jacobsen, 2013] Jacobsen, T. (2013). End-user qos indication by passive network measurement at isp core.

[Jacobson et al., 2013] Jacobson, V., Leres, C., and McCanne, S. (2013). Tcpdump.org.

[Jiang and Dovrolis, 2002] Jiang, H. and Dovrolis, C. (2002). Passive estimation of tcp round-trip times. *SIGCOMM Comput. Commun. Rev.*, 32(3):75–88.

[Masud et al., 2008] Masud, M., Al-Khateeb, T., Khan, L., Thuraisingham, B., and Hamlen, K. (2008). Flow-based identification of botnet traffic by mining multiple log files. In *Distributed Framework and Applications, 2008. DFmA 2008. First International Conference on*, pages 200–206.

[Matija Stevanovic, 2013] Matija Stevanovic, J. P. (2013). Machine learning for identifying botnet network traffic.

[Mockapetris, 1987] Mockapetris, P. (1987). Domain names - implementation and specification. STD 13, RFC Editor. `http://www.rfc-editor.org/rfc/rfc1035.txt`.

[Netforsk,] Netforsk, A. Volunteer-based system for research on the internet. Visited 15'th of May 2015.

[Quinlan, 1993] Quinlan, R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA.

[Rossow et al., 2011] Rossow, C., Dietrich, C. J., Bos, H., Cavallaro, L., van Steen, M., Freiling, F. C., and Pohlmann, N. (2011). Sandnet: Network traffic analysis of malicious software. In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, BADGERS '11, pages 78–88, New York, NY, USA. ACM.

[Shin et al., 2013] Shin, S., Xu, Z., and Gu, G. (2013). Effort: A new host–network cooperated framework for efficient and effective bot malware detection. *Computer Networks*, 57(13):2628 – 2642.

[SQLite,] SQLite. Sqlite homepage. Visited 15'th of May 2015.

[Stevanovic and Pedersen, 2014] Stevanovic, M. and Pedersen, J. (2014). *An efficient flow-based botnet detection using supervised machine learning*, pages 797–801. International Conference on Computing, Networking and Communications. IEEE.

[Stevanovic et al., 2012] Stevanovic, M., Revsbech, K., Pedersen, J., Sharp, R., and Damsgaard Jensen, C. (2012). *A collaborative approach to botnet protection*, volume 7465 of *Lecture Notes in Computer Science*, pages 624–638. Springer.

[Symantec, 2014] Symantec (2014). Threads to virtual environments. Visited at 15'th of Match 2015.

[Symantec, 2015] Symantec (2015). Internet security threat report. Visited at 19'th of May 2015.

[Technologiesy, 2014] Technologiesy, S. (2014). Jnetpcap. Visited at 25 may 2014.

[university of Waikato, 2014] university of Waikato, T. (2014). Weka 3: Data mining software in java. Visited 20'th of April 2015.

[Zeng et al., 2010] Zeng, Y., Hu, X., and Shin, K. (2010). Detection of botnets using combined host- and network-level information. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 291–300.

Part I

Appendices

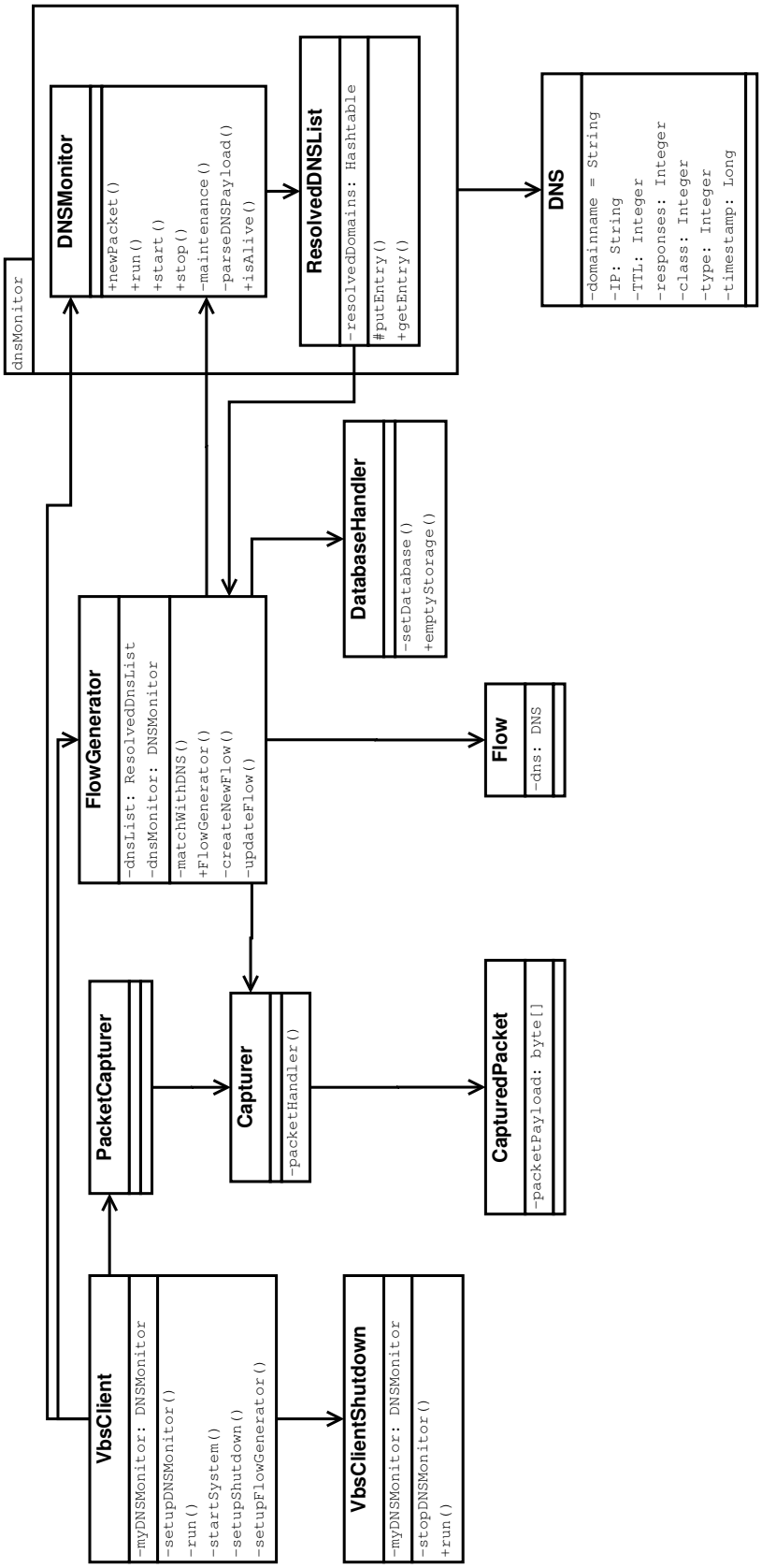

Figure 1: All changes involved in incorporating the `DNSMonitor` in a class diagram.

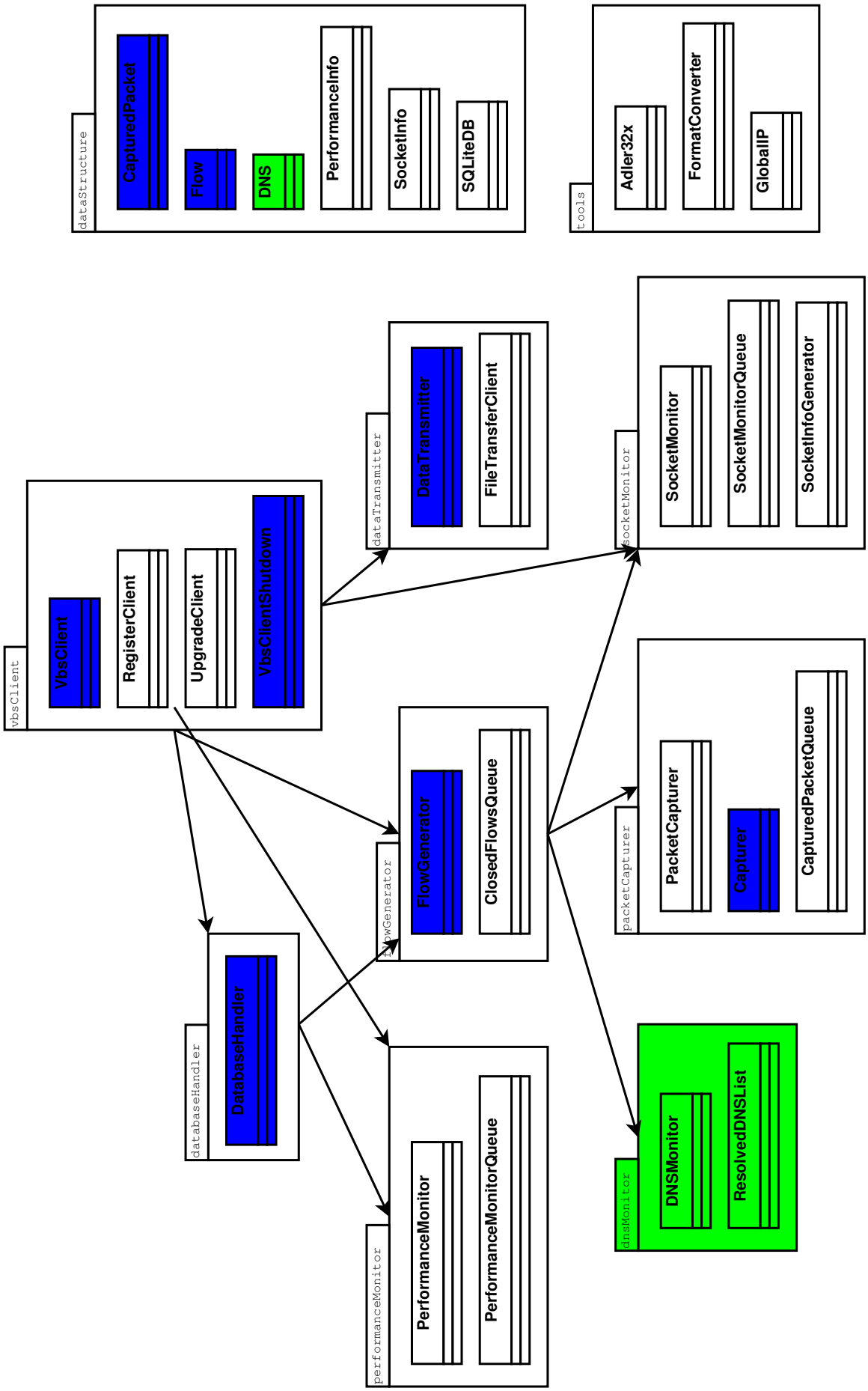

Figure 2: Client full class diagram. Blue means a VBS class which is changed. Green is a new class.

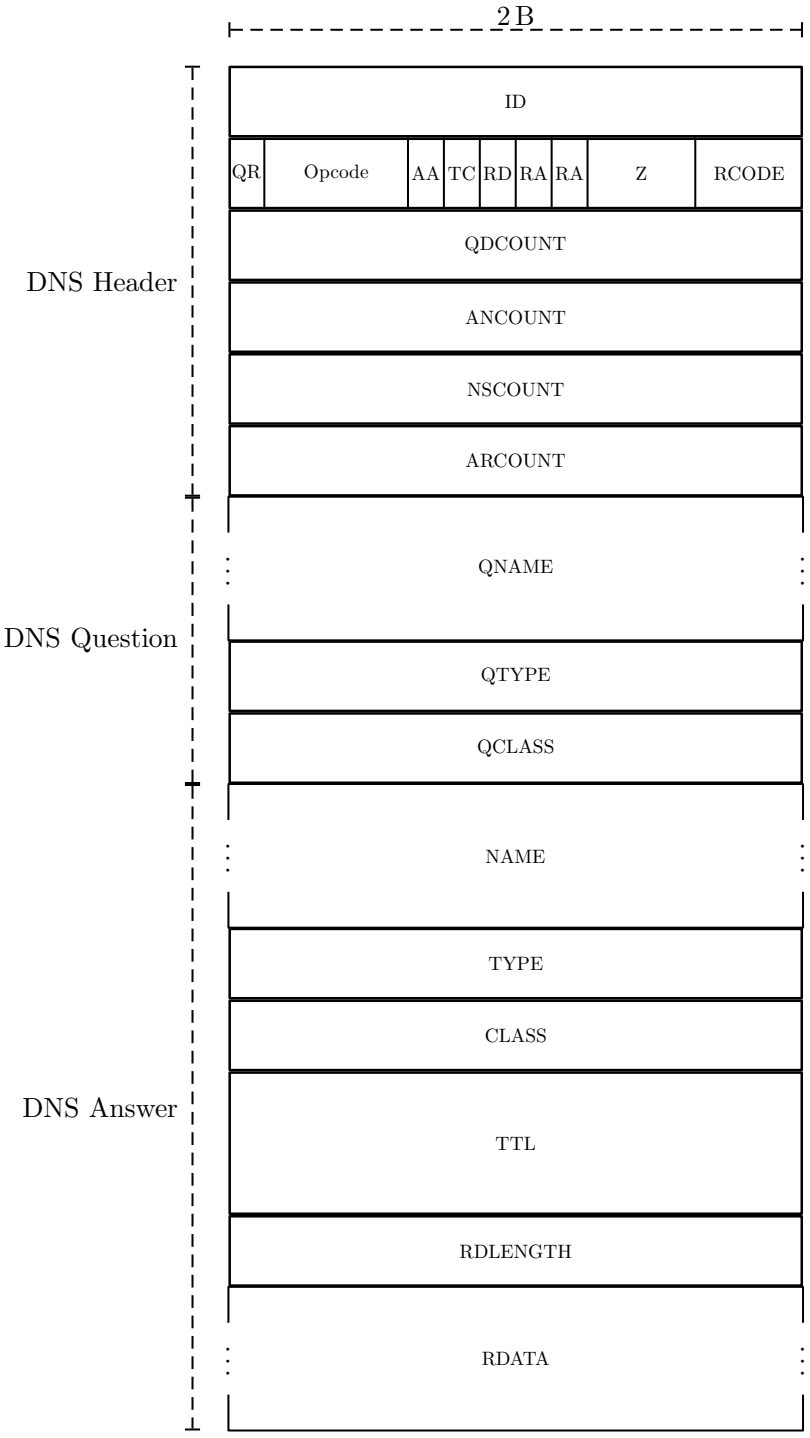

Figure 3: DNS response packet structure from [Mockapetris, 1987] with one question and response.