Sandnet++ - A framework for analysing and visualising network traffic from malware

Anthony Nelson

Technical Report

RHUL–ISG–2016–11

5 April 2016

Information Security Group
Royal Holloway University of London
Egham, Surrey, TW20 0EX
United Kingdom
Sandnet++ - A Framework for Analysing and Visualising Network Traffic from Malware

Author: Anthony Nelson

Email address: agn5@student.london.ac.uk

Date: March 2015

Supervisor: Lorenzo Cavallaro

Student number: 120274866

Submitted as part of the requirements for the award of the MSc in Information Security of the University of London.

ANTI-PLAGIARISM DECLARATION
You are reminded that all work submitted as part of the requirements for any examination of the University of London must be expressed in your own words and incorporate your own ideas and judgements.

Using another person’s thoughts or words as though they were your own work is considered to be copying or plagiarism. This is not allowed. You must clearly identify direct quotations from the published or unpublished work of another person by placing those quotes inside quotation marks, and also provide a full reference to their source.

Whether you use a series of short quotations from several different sources or a single long quotation from a single source, they must all be identified clearly. Equally, if you summarise another person’s ideas or judgements, you must refer to that person in your text, and include the work referred to in your bibliography. Failure to observe these rules may result in an allegation of cheating.

I declare that this dissertation is all my own work, and that I have acknowledged all Quotations from the published or unpublished works of other people.

I declare that I have also read the statement on plagiarism in the General Regulations for Awards at Graduate and Masters Levels for the MSc in Information Security and in accordance with it I submit this project report as my own work.

Please sign here to show that you have read the above:

_________________________________________ Date: __________________
## Contents

List of Figures iv  
List of Tables vi  
Abstract 1  

1 Introduction 2  
1.1 Report Objectives 3  
1.2 Motivation 4  
1.3 Scope 4  
1.4 Methods Used 4  

2 Malware Analysis 6  
2.1 Dynamic Malware Analysis 6  
2.1.1 Collection Method 6  
2.1.2 Existing Systems 7  
2.2 Clustering Malware 9  
2.3 Visualisation 10  
2.4 Network Intrusion Detection System Signatures 10  
2.5 Feature Extraction 11  
2.6 Networking Protocols and Programming Languages 12  

3 System Design 13  
3.1 Overall System Design 13  
3.2 Capture Framework Design 14  
3.2.1 Summary of Capture Framework Design Decisions 15  
3.3 Analysis Framework Design 16  
3.3.1 Feature Extraction 20  
3.3.2 Database Schema 23  
3.3.3 Summary of Analysis Framework Design Decisions 23  
3.4 Visualisation (Web Client) 24  

4 Clustering 26  
4.1 Malware Traffic Clustering 26  
4.1.1 Packet Clustering 27  
4.1.2 Session Clustering 28  
4.2 Traffic Capture Types and Clustering 31  
4.2.1 IDS Signature Generation for Malware Clusters 31  
4.3 Clustering Summary 32  

5 Implementation 33  
5.1 Capture Framework 33  
5.1.1 Choice of Sandbox 33  
5.1.2 Cuckoo Sandbox 33  
5.1.3 Configuration Options 35  
5.1.4 Source Code 35  
5.2 Analysis Framework 36  
5.2.1 Network Capture Format 36  
5.2.2 Database 36  
5.2.3 Choice of Packet Processing Code Library 37
List of Figures

3.1 Framework inputs and outputs ........................................... 13
3.2 Capture framework design .............................................. 15
3.3 Analysis framework design. Arrows are used to show the flow of data around the system. ........................................... 18
4.1 Visual separation of clusters achieved by highlighting alternate cluster rows ........................................... 27
5.1 Visualisation using D3, present in an early version of the web client. This bar chart shows the count of DNS packets that contain specific ‘answer count’ values. ........................................... 41
5.2 Visualisation using Highcharts. This area chart shows the distribution of response data lengths among sessions in a pcap ........................................... 42
6.1 Histogram results displayed in a table .................................. 45
6.2 Histogram results displayed as a bar chart ................................ 46
6.3 Histogram results displayed as a column chart ................................ 47
6.4 Histogram results displayed as a line chart ................................ 48
6.5 Histogram results displayed as an area chart ................................ 49
6.6 Histogram results displayed as a pie chart ................................ 50
6.7 Packet (transport) clustering results displayed in a table ........................................... 51
6.8 Packet (transport) clustering results displayed as a pie chart ........................................... 52
6.9 Session (HTTP request) clustering results displayed in a table ........................................... 53
6.10 Session (HTTP request) clustering results displayed as a pie chart ........................................... 54
7.1 6ec4ba6e32a5bb6006cdd06c518246ee2f551a_001146.pcap transport clustering ........................................... 61
7.2 c911088c90f85d9c1331e616034e28cba5d2cb52_019694.pcap packet destination IP addresses ........................................... 62
7.3 c911088c90f85d9c1331e616034e28cba5d2cb52_019694.pcap session HTTP request Host field values ........................................... 63
7.4 c911088c90f85d9c1331e616034e28cba5d2cb52_019694.pcap session HTTP request User-Agent field values ........................................... 64
7.5 c911088c90f85d9c1331e616034e28cba5d2cb52_019694.pcap HTTP request clusters using a threshold of 0.1 ........................................... 65
7.6 7a5749e38aaf62e2a05fe78af672ce46de6406_000966.pcap HTTP request clusters using a threshold of 3 ........................................... 66
7.7 ae7c22714babc3abcaada0aa7bc79b524952e9_021033.pcap packet protocols. Note, as multiple protocols can be identified in one packet the sum of the counts adds up to more than 654 (the number of the packets) ........................................... 67
7.8 ae7c22714babc3abcaada0aa7bc79b524952e9_021033.pcap session destination IP addresses ........................................... 68
7.9 ae7c22714babc3abcaada0aa7bc79b524952e9_021033.pcap session HTTP request Host field values ........................................... 69
7.10 ae7c22714babc3abcaada0aa7bc79b524952e9_021033.pcap HTTP request clusters with a k parameter value of 4 ........................................... 70
7.11 Gameover Zeus UDP port numbers histogram (sample size 100,000) [103] ........................................... 71
7.12 a6a7a73f7944a0a593e8f143cbce1d608782998_019263.pcap UDP destination ports. The shaded area highlights port numbers between 10000-30000. ........................................... 71
7.13 a6a7a73f7944a0a593e8f143cbce1d608782998_019263.pcap HTTP request clusters using a threshold of 0.1. Two of the requests in cluster 3 are not to randomised domain names. ........................................... 73
7.14 a6a7a73f7944a0a593e8f143cbbce1d608782998,019263.pcap HTTP request clusters with a k parameter value of 5. Cluster 5 contains only requests to randomised domain names. ................................................................. 74
7.15 Cross-family HTTP request clusters with a k parameter value of 4 .................. 75
7.16 Cross-family HTTP request clusters with a k parameter value of 6 .................. 76
7.17 The % of Cleaman family samples that contain each packet protocol .............. 77
7.18 Cleaman family IP packet TTLs and the percentage of pcap samples that contain them ................................................................. 78
7.19 The distribution of packet IP TTL values across all Cleaman samples ............. 78
7.20 Cleaman HTTP request clusters with a threshold parameter value of 7. Cluster 2 is entirely Cleaman traffic and has been correctly separated from the MALICIA capture framework traffic. ................................................................. 79
7.21 Cutwail family transport protocol clustering. 838 clusters are generated ........ 82
7.22 Cutwail family HTTP request clustering using a threshold of 7. Clusters 3, 4 and 5 contain the Cutwail traffic. ................................................................. 83
7.23 DNS query resource record type distribution across all MALICIA samples ....... 85
7.24 All HTTP request URI parameter names that appear in at least 1% of MALICIA network samples. Note, the top two parameters appear in traffic generated by the MALICIA capture framework itself. ................................................................. 87
7.25 HTTP request header names and the number of MALICIA traffic captures they appear in ................................................................. 88
7.26 HTTP Accept-Language header values present in the MALICIA dataset ......... 89
7.27 HTTP Accept header values present in the MALICIA dataset and the number of samples (out of 16967) they are present in ................................................................. 90
7.28 cdae6fee42288a8cf8b8de7ed683b6f325b4a71ef-zbot.pcap HTTP header names ... 93
7.29 cdae6fee42288a8cf8b8de7ed683b6f325b4a71ef-zbot.pcap HTTP POST body lengths 94
7.30 cdae6fee42288a8cf8b8de7ed683b6f325b4a71ef-zbot.pcap transport clustering .... 95
7.31 0865f1361e09c5f4bbaa333975b1962359d0f5-dridex.pcap TCP packet data length distribution ................................................................. 96
7.32 0865f1361e09c5f4bbaa333975b1962359d0f5-dridex.pcap HTTP request header distribution ................................................................. 97
7.33 0865f1361e09c5f4bbaa333975b1962359d0f5-dridex.pcap HTTP Hosts ............. 99
7.34 Dridex and Zeus HTTP request clusters with a k parameter value of 2 .......... 100
T.1 The main page of the web client ................................................................. 174
T.2 The upload file page of the web client. This uploads a file to the capture framework for analysis ................................................................. 175
T.3 The upload pcap page of the web client. This uploads a packet capture to the analysis framework ................................................................. 176
T.4 The view uploads page of the web client ................................................................. 177
T.5 The view multiple uploads page of the web client ................................................................. 178
T.6 The visualiser, when first loaded, and the visualisation options available ........ 179
# List of Tables

2.1 A comparison of static vs dynamic malware analysis. In practice, due to time and resource constraints, or security concerns, the benefits identified may not be fully realised. .......................................................... 7

3.1 Capture framework API endpoints ................................................. 15

3.2 Analysis framework API miscellaneous endpoints. Values enclosed in ⟨⟩ are variables that form part of the URI path. .......................................................... 18

3.3 Analysis framework API upload and metadata endpoints. Values enclosed in ⟨⟩ are variables that form part of the URI path. ......................................................... 19

3.4 Analysis framework API analysis endpoints. Values enclosed in ⟨⟩ are variables that form part of the URI path. .......................................................... 20

3.5 Analysis framework database schema field descriptions ........................... 23

3.6 Visualisations available in the web client ........................................... 25

4.1 Possible IDS signature coverage by malware variety. This is speculation based on observations made when conducting analysis for this report. .................. 32

5.1 Changes made to the default Cuckoo Sandbox configuration files .............. 35

5.2 Capture framework source code file listing ........................................ 36

5.3 Analysis framework source code file listing ........................................ 40

5.4 Lines of code used to create the framework, split by file type .................... 43

7.1 Histogram features analysed ....................................................... 59

7.2 Clustering algorithms analysis parameters. 0.1 is used as a threshold value to group identical HTTP requests. ......................................................... 59

7.3 MALICIA pcap inputs to be analysed individually, along with their family and traffic label as identified in the MALICIA dataset ................................... 60

7.4 URLs and the number of times they occur in Cridex traffic ....................... 81

7.5 Malware family and hashes of the two samples analysed using the capture framework ............................................................. 92

H.1 Visualisations and their options available in the web client ...................... 130

O.1 HTTP request header names, the number of MALICIA samples they appear in (out of 16967) and the count of occurrences across all MALICIA samples .......... 154

O.2 HTTP response header names, the number of MALICIA samples they appear in (out of 16967) and the count of occurrences across all MALICIA samples .......... 155

P.1 HTTP User-Agents, the number of MALICIA samples they appear in (out of 16967) and the count of occurrences across all MALICIA samples. The top two belong to the MALICIA capture framework itself. .............................. 157

Q.1 HTTP Server header values, the number of MALICIA samples they appear in (out of 16967) and the count of occurrences across all MALICIA samples. Several of the Servers appear to be used by the MALICIA capture framework itself (particularly those starting “SimpleHTTPWithUpload/0.1”) ................................................. 159

R.1 HTTP response Content-Type header values, the number of MALICIA samples they appear in (out of 16967) and the count of occurrences across all MALICIA samples 160
The field of information security faces many challenges. Perhaps one of the most important is the threat posed by malicious software (malware). Malware is harmful to the intended operation of a computer system and breaches at least one of the three tenets of information security: confidentiality, integrity and availability.

For example, malware may steal sensitive information from a computer system, such as a banking Trojan that steals credit card details and passwords. Computer viruses can infect files stored on a computer and malware may be able to spread to data backups, infecting those as well. Botnets have been used to launch Distributed Denial of Service (DDoS) attacks against unsuspecting victims, and other malware such as CryptoLocker, so called ‘ransomware’, is used to encrypt files on a victim machine to prevent the legitimate user accessing data until a ransom is paid.

One important step in combating malware is to understand how it communicates over a computer network, typically the internet. Most malware has to communicate remotely, whether to infect further victims, exfiltrate stolen information or receive instructions. Examining the network traffic generated by malware provides an opportunity to identify the unique features found only in malware traffic, and use these to distinguish it from benign traffic. Only if malware traffic is identifiable can it be blocked or otherwise disrupted. Detection of malware at a network level may also be used to identify and report infected hosts within an organisation.

The purpose of this report is to present a framework for analysing and visualising network traffic from malware. Background research is conducted into the relevant aspects of malware analysis, specifically dynamic malware analysis as a means to generate network traffic from malware. A system is then designed and implemented which consists of two components: the first is capable of running a malware sample and recording the network traffic; while the second can process the network traffic to extract multiple features. These features are used for the analysis and visualisation of malware traffic. Different types of visualisation are supported by the framework and the design allows new ones to be added.

Two traffic clustering algorithms are proposed and added to the framework developed. One proves particularly effective at grouping similar network communications and can distinguish between traffic from different malware families.

Experiments are conducted using the framework to analyse a published dataset of malware network traffic. Traffic from five malware samples is examined in detail before being aggregated and studied as a whole. Subsequently, three malware families are analysed and their network behaviour examined. The entire dataset is then studied as a whole and the benefits of the feature extraction process for malware traffic analysis are demonstrated.

The experiments are completed by executing two malware samples, capturing the network traffic and analysing the network behaviour. The report concludes with a discussion of future work that could be undertaken to further the framework that has been developed, and a reminder that the framework is extensible should others wish to use it for their own purposes.
Chapter 1

Introduction

Malicious software, most commonly known as malware, is an ever growing threat to computer users around the world and there are constant reports describing the latest discoveries in detail [15, 68, 128]. Multiple solutions [38, 106, 120] exist to detect network activity generated by malware, firewalls can be used to block malicious network traffic and anti-virus products can be installed on a computer to identify known malware. All of these require an understanding of how malware operates to accurately identify it. This report focuses on malicious network traffic and how it can be studied to inform how malware operates.

The focus on network activity is appropriate as the majority of malware requires some form of network communication throughout its lifecycle. This could be the initial infection vector, a beacon to command and control infrastructure to receive commands, or the exfiltration of stolen information.

The topic of network Intrusion Detection System (IDS) signatures is a recurrent theme throughout this report. IDS systems are used by organisations as one method to protect their network. To be able to detect malicious activity a network IDS system typically uses a collection of signatures, each of which describes network behaviour. That behaviour may be specific to one piece of malware, or may identify an entire family. Accurate signatures are required, otherwise legitimate network traffic may be mistakenly identified as malicious. Whilst network signatures are not an output of this project, their importance in current detection systems should not be overlooked. As such, features of malware traffic that could be used in network IDS signatures are identified throughout this report.

Before malware traffic can be studied it must first be collected. This project begins with a review of malware analysis, in particular dynamic malware analysis. Techniques for collecting network traffic from malware are discussed and existing systems reviewed, resulting in selection of Cuckoo Sandbox as the basis of a dynamic malware analysis system. Its use in this project is restricted to executing malware solely to obtain a network traffic capture (often referred to as pcap, or packet capture, throughout this report).

The topic of visualisation is touched upon to provide context for the visualisations required in this project. A definition of information visualisation is given and its benefits outlined. Examples of the type of visualisations created can be seen throughout the report.

The research phase finishes with an explanation of the core malware analysis approach taken by this project: protocol feature extraction. Many network IDS signatures use specific features of a protocol to identify malicious traffic. By extracting and studying these features, malware behaviour can be better understood, and ultimately detected if variations from legitimate traffic exist.

This report aims to expand on Sandnet [33, 112] by producing a framework to analyse and visualise network traffic from malware. To achieve this, a design is proposed that contains two main subsystems: a capture framework and an analysis framework. Each of these is presented in detail. The specific implementation of the design that has been created is described and its usage explained: a web client is used to upload malware samples to be executed; the protocol features are extracted from the resultant packet capture; feature data is converted to the desired visualisation
and displayed for analysis. Existing network captures can be uploaded directly to the analysis framework without the need to use the capture framework.

Automation is an important feature of the framework due to increasing numbers of new malware [67]. Having an Application Programming Interface (API) to allow automated access to functionality allows analysis to be conducted quickly and opens up the possibility for integration with other systems.

Another approach to malware analysis is to cluster network traffic to identify common behaviours. Techniques to cluster malware traffic are investigated in this report to understand how they can complement the analysis process. Two algorithms, modified from existing research [101, 107], are proposed and integrated into the developed framework.

The main body of work concludes with an analysis of malicious traffic. This serves as an evaluation of the framework and establishes its usefulness in analysing malware traffic. The clustering algorithms are tested and one of these is shown to accurately group similar communications and can distinguish between traffic from different malware families. The analysis is split into two sections: the majority of the work focuses on a published dataset of malicious traffic (the MALICIA dataset [55, 85, 86]); the remaining analysis uses two malware samples to test the entire end-to-end framework.

The report finishes with suggestions for future work, revisits the main results of the analysis undertaken and reiterates the contributions of this project to the field of malware analysis.

### 1.1 Report Objectives

The objectives for this report are split into core objectives, which MUST be achieved, and extension objectives, which MAY be achieved given sufficient time and are designed to expand on the core objectives.

**Core objectives:**

1. the design and implementation of a framework to extract information of interest from previously captured network traffic (whether from objective 6 or elsewhere, e.g. publicly available pcap files) and to produce visualisations to aid analysis (*analysis framework*)

2. the design and implementation of an API to allow automatic interaction with the analysis framework

3. a study of previously captured network traffic from malware using the analysis framework

**Extension objectives:**

4. an exploration of visualisation methods to help identify malicious network traffic

5. an exploration of methods to cluster malicious network traffic

6. the design and implementation of a framework to run malware over a specified time period and to capture the resulting network traffic (*capture framework*)

7. the design and implementation of an API to allow automatic interaction with the capture framework

8. a study of identified malware using the capture framework and the analysis framework and an analysis of the network traffic observed
Note - throughout this report when referring to the framework this should be taken to mean the capture framework and the analysis framework as a whole. The capture framework and analysis framework will be referred to individually when required for clarity.

1.2 Motivation

Having an interest in the computer security industry and having observed the wave of malware reporting released over the last few years, I decided that a project focused on malware capabilities would be of great interest and value. There are many aspects of malware that I could have focussed on, such as evasion techniques, encryption, or infection vectors. However, I decided to concentrate on the malicious network traffic. As I have a working knowledge of networking protocols and network forensics this suited me well. My background in software engineering is also important due to the highly practical nature of this project.

1.3 Scope

Only network traffic from known malware is within scope of this report, i.e. no attempt is made to determine if the original input (file or network traffic) is malicious or not. This is an important distinction to make and whilst the framework developed could be used to aid discovery of new malware samples, it is not the focus of this report.

The analysis framework is not limited to network traffic generated by any particular operating system. Technically, the capture framework can also run malware designed for a number of different operating systems, but for this report only Microsoft Windows XP (32 bit) was used. Thus, only malware targeted at Microsoft Windows x86 is considered within scope for the capture framework.

Any software libraries used in development of the framework are restricted to open source or freely available versions.

1.4 Methods Used

By having both core and extension objectives I ensure the bulk of the project has well defined goals with sufficient room for expansion in a variety of directions should this be achievable. The methods used to fulfill the objectives include:

- **literature search** - it is important to understand related work in this area, what solutions already exist (e.g. dynamic malware analysis systems) and how malware traffic can be analysed. Findings are presented in chapter 2.

- **technical and implementation research** - this covers research into topics such as networking protocols (e.g. Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Domain Name System (DNS) and Hypertext Transfer Protocol (HTTP), section 2.6), software to be used in this project (chapter 5 and Appendix C), design considerations (chapter 3 and chapter 4), identification of existing malware network traffic captures (section 7.1) and selection of suitable malware samples (section 7.5.1)

- **implementation** - the development of the framework designed in this project and resolution of any programming problems encountered. Along with the analysis, this was the most time consuming part of the project. Programming languages used include Python 2, Python
3. Bash shell, Bourne shell and JavaScript. HyperText Markup Language (HTML) and Cascading Style Sheets (CSS) are used for the user interface. The implementation is presented in chapter 5 and visualisations shown in chapter 6 and throughout chapter 7.

- **experiments** - use of the developed framework to capture and analyse malware network traffic, detailed in chapter 7
Chapter 2

Malware Analysis

This section presents research into the core topics covered by this report and highlights related work.

2.1 Dynamic Malware Analysis

Dynamic malware analysis is the process of analysing malware by executing it, often in a container or sandbox. The malware is usually allowed to perform its actions such as initial exploitation of a vulnerable computer program, installation then subsequent communication with command and control infrastructure. Such activities are monitored and recorded. Analysis can then be conducted on the data collected. Execution of the malware may be manual, such as an analyst launching an executable and interacting with any dialogues that present themselves, or it may be completely automated, whereby a software system is capable of running malware with no human-interaction.

Static malware analysis involves study of the source code, or assembly code if that is all that can be obtained, to identify what capabilities a piece of malware has and what these may be used for. For example, by converting an executable file to assembly code (a process known as disassembly and performed by a disassembler program) it is possible to view the instructions that will be executed by the CPU and any system calls made.

Some of the theoretical benefits and drawbacks of static and dynamic malware analysis are outlined in Table 2.1. Further information can be found in [28], including how dynamic malware analysis systems identify activities (e.g. files written, system calls made) on a host. However, the focus of this project is on network traffic so further discussion is centred on dynamic malware analysis as a means to capture network traffic. Static analysis, whilst capable of providing details of the network protocols that will be used, is unable to actually generate the network traffic required for this project so is not discussed further.

2.1.1 Collection Method

A variety of techniques have been proposed for collecting malware samples. For example, [85] and [108] download malware from known exploit server URLs using honeyclients and ‘milkers’. Honeypots inside virtual machines are used in [133], although the raw executable file is not saved for future use. In [57] malware is collected using a spam feed, extracting URLs from the emails which “point to malicious executables or drive-by downloads” [57] then visiting those URLs. Collaboration with security researchers and anti-virus companies is another method of obtaining malware samples. Public malware repositories such as Offensive Computing [17] and VirusTotal [132] can also be used to obtain malicious files, although the latter requires an account for its Intelligence platform. A research account has been obtained for the VirusTotal Intelligence platform for use in this project.

It is not the intention of this report to focus on how malware samples are collected. Instead, in
<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malware is not executed</td>
<td>Malware is executed</td>
</tr>
<tr>
<td>All execution paths possible can be examined</td>
<td>It may not be possible to fulfil all conditions required for certain functionality to execute (e.g. checks for specific dates/times or specific software installed)</td>
</tr>
<tr>
<td>Can be slow (e.g. an analyst interpreting dis-assembled code) or fast (automated checks for strings or bytes in a file)</td>
<td>Can be slow (e.g. manually running a sample or using an automated system to run a malware sample for days) or fast (running a malware sample for a few seconds)</td>
</tr>
<tr>
<td>No interaction with malicious infrastructure possible</td>
<td>Possible to communicate with malicious infrastructure and study any data returned</td>
</tr>
<tr>
<td>Can be fully automated (although manual analysis of the assembly code may be required)</td>
<td>Can be fully automated (although some malware may fail to run)</td>
</tr>
<tr>
<td>May require experience to get a malware sample to run, but some malware, e.g. in executable form, may be trivial to run</td>
<td>May require an experienced analyst, particularly for assembly code analysis. However, automated tools can help identify relevant information.</td>
</tr>
<tr>
<td>May be difficult if malware is obfuscated or packed</td>
<td>Malware can be allowed to unpack itself</td>
</tr>
<tr>
<td>Unlikely for malware to infect the analysis machine</td>
<td>Analysis machine likely to be infected by the malware</td>
</tr>
</tbody>
</table>

Table 2.1: A comparison of static vs dynamic malware analysis. In practice, due to time and resource constraints, or security concerns, the benefits identified may not be fully realised.

scope for the project is how malware can be executed to collect network traffic. As such, malware samples are limited to those that can be obtained from published malware datasets. Details of the datasets used in this project are provided in section 7.1.

### 2.1.2 Existing Systems

A number of different dynamic malware analysis systems exist although the approach they take to perform the analysis differs greatly. For example, Cuckoo Sandbox [34] performs analysis in a virtual machine such as VirtualBox [95] whereas TTAnealyse [5] uses QEMU [6] to emulate a PC and execute programs. CWSandbox [134], Cuckoo Sandbox and Joe Sandbox Complete (Joe Sandbox) [117] can even execute malware on a physical host, although a restore procedure is required to reset the system to a clean state and is likely to be more time consuming than a restore of a virtualised or emulated environment.

**Note:** *Emulation and virtualisation are subtly different. In a virtualised environment “much of the software for the simulated machine executes directly on the hardware without software interpretation” [47]. In an emulated environment the hardware is imitated using software, which “allows an analysis component to control every aspect of program execution” [28]. Both emulation and virtualisation can be used as a sandbox for malware execution.*

According to [117], Joe Sandbox “executes files and URLs in a controlled environment and monitors the behavior of applications and the operating system for suspicious activities. All activities are compiled into comprehensive and extensive analysis reports”. Joe Sandbox can use different virtualisation environments and supports a variety of target operating systems (most desktop versions of Microsoft Windows as well as the Android mobile operating system and Mac OS X). Unlike most other dynamic malware analysis systems, Joe Sandbox is capable of using “a mix of virtual and physical analysis machines for analysis. Physical devices are very helpful in order to
deal with evasive malware which may not run on virtual systems" [118]. A plugin for Joe Sandbox exists, called Joe Sandbox Class, which claims to use features from analysis reports to identify similar malware samples [116]. However, of particular note is that Joe Sandbox does not appear to make raw network traffic available for further analysis.

TTAnalyze focuses on analysing Microsoft Windows executables. It uses a modified version of the QEMU emulator for PC emulation, an RPC server to allow communication in and out of the virtual system, can detect which system calls are made and the function call arguments and is capable of producing reports containing information about file, process and network activity [5]. The network activity report is a “log that contains all network traffic sent or received by the test subject” [5]. Sample reports available at [66] indicate that full content of network communications is not available in the output of TTAnalyzer (or Anubis [4, 65], the successor to TTAnalyzer). TTAnalyzer claims to trade speed for emulation accuracy - “While the analysis is significantly slower compared to a virus scanner, the accuracy of the emulation is excellent. Since our focus is on the analysis of the behavior of the binary, this trade-off is acceptable” [5].

Cuckoo Sandbox runs one or more analysis virtual machines which communicate over a virtual network to a controller system. Much like TTAnalyzer, an agent is needed inside the virtual system. Cuckoo Sandbox uses an XMLRPC protocol to communicate with the agent software installed on the analysis machine. This is used to upload the malware sample and instruct the analysis machine to begin execution. Once the virtual machine has been running for a defined period of time a number of different processing modules are called, for example for memory dump analysis or for extracting information from the network traffic capture. The virtual machine is then shutdown. The raw packet capture is available after the analysis has completed. It is also possible to use Cuckoo Sandbox with physical hosts. Malwr.com [40] is an example of a publicly available instance of Cuckoo Sandbox to which anyone can submit files for analysis.

CWSandbox can be run in a number of different ways including inside a virtual machine. It is targeted at Microsoft Windows executables and uses a technique known as API hooking to intercept and identify system calls that are made inside the analysis environment, as well as DLL code injection to place its own code inside malicious executables and any processes spawned by the malware (including new executables downloaded by the original executable). CWSandbox generates an XML analysis report detailing activities such as “file system changes, registry modifications, mutex creation, or process-management actions” [134]. ThreatAnalyzer is the successor to CWSandbox and includes a pcap file of “All network activity generated by a sample during analysis” [119].

Sandnet [33, 112] takes a different approach to many of the dynamic malware analysis systems discussed and focuses on the network traffic generated by the malware, as opposed to the host based activities (file modifications, registry changes, processes spawned etc.). It uses VirtualBox [95] to host Windows XP Service Pack 3 virtual machines, termed ‘sandpuppets’. A controlling ‘sandherder’ hosts the virtual machines. Malware samples are executed for a defined time period, at least one hour, and network traffic is recorded. UDP and TCP flows (sessions) are reconstructed from a pcap file and application layer protocols identified.

Other dynamic malware analysis systems exist: The Platform for Architecture-Neutral Dynamic Analysis (PANDA) [25] is another system built on top of QEMU, Norman Sandbox, recently acquired by AVG Technologies [113], “simulates an entire computer and a connected network by reimplementing the core Windows system and executing the malware binary within the simulated environment” [134] and Ether [22] utilises hardware virtualisation extensions to enable it to sit outside of the virtual environment.

Regardless of which dynamic malware analysis environment is used for this project, the most important property is that it makes network traffic available for inspection so that relevant data can be identified and made available for analysis and visualisation. With the correct implementation, the specific dynamic malware analysis environment used can be always be replaced according to
need. For example, some environments, such as Ether, focus on evading detection by malware samples, so may be appropriate to use when running malware capable of detecting more ‘obvious’ virtualisation environments. However, detection of the analysis environment is not the focus of this project and modifications to hide it from malware samples are out of scope.

2.2 Clustering Malware

Much research has been conducted on how malware can be clustered [3, 14, 46, 50, 85, 100, 101, 107, 111, 123]. For instance, in FIRMA, “traffic in the network traces is first partitioned into traffic clusters using features that identify similar traffic, then signatures are created for each traffic cluster, and finally a sequence of steps merges similar signatures and groups signatures for the same family into signature clusters” [107]. This approach can cope with HTTP, Internet Relay Chat (IRC) and Simple Mail Transfer Protocol (SMTP) at the application layer, and TCP and UDP at the transport layer.

A different approach is taken in [101] - a three step clustering process is used. First coarse-grained clustering is performed on “malware samples based on simple statistical features extracted from their malicious HTTP traffic” [101]. Then fine-grained clustering is performed within each coarse-grained cluster. This measures “the structural similarity between the HTTP traffic generated by each sample in a cluster”, using features of the HTTP request line. The fine grained clusters are then merged if their HTTP traffic is similar enough. An advantage of this three step clustering method is that the more expensive fine-grained clustering (which requires each HTTP request to be compared) is only performed within each cluster, increasing the efficiency of the clustering. This approach is limited to HTTP only.

Simplifying [101], the techniques proposed in [100] focus on scalability of the clustering technique. The clustering algorithm used for coarse-grained clustering is changed to the BIRCH algorithm [136] and the third step (cluster merging) no longer exists. The authors demonstrate that with these changes that “scalability is achieved while retaining a good trade-off between detection rate and false positives for the signatures derived from the obtained malware clusters” [100]. This is limited to HTTP traffic.

Clustering exploit servers in [85] is based on five features: URL path and parameter values, DNS domain to IP address resolution, file hash, executable file icon and the malware family (if known). Two clustering algorithms are proposed: “partitioning around medoids (PAM) [20] and an aggressive clustering algorithm that groups any servers with some similarity” [85].

The authors of [3] demonstrate a system capable of clustering behavioural profiles. These profiles include network traffic from executing a malware sample, as well as host behaviour (e.g. writing to a file). Locality sensitive hashing and hierarchical clustering is used identify similar behavioural profiles, hence similar malware.

A detailed discussion of the clustering implemented in this project can be found in chapter 4. In contrast to some of the research discussed above, the focus is entirely on the network traffic generated by malware, not host level features such as files written to disk. The goal of the clustering is also subtly different from much of the above work. This project seeks to group similar network traffic, not similar malware samples (although that is possible if the traffic clustered comes from different malware). This leads to an important benefit - clustering of traffic generated by only one malware sample can be undertaken. This allows detailed analysis of the malware traffic to clearly understand the threat.

Once a specific piece of malware has been analysed further understanding can be gained by clustering network traffic obtained from the same malware family. This can highlight similarities across the variants of network communications within the family. If desired, clustering can then be ex-
tended across a much wider dataset, perhaps the entire dataset, to provide overall statistics and identify any common network communications used by unrelated malware. Using these different types of clustering allows trends to be drawn and a deeper understanding to be developed. A description of the traffic types clustered in this project can be found in section 4.2.

2.3 Visualisation

An in-depth study of visualisation is out of scope of this project, however, simply presenting raw textual (or binary) information may be inappropriate when trying to understand certain data. For example, when viewing the distribution of IP addresses a malware sample communicates with (and the number of communications), this can easily presented as text within a table. However, by using another visualisation method, such as a bar chart, or overlaying a heat map, properties of the data may stand out, for example the IP address that is contacted the most. Giving the viewer the chance to explore the data and discover properties within it can be considered an integral part of the network traffic analysis.

To bring focus to the visualisation in this project a suitable definition is required:

*The goal of information visualization is to transform abstract data into computer graphics that are easy to understand and to use to support decision making and reasoning* [18, p. 1]

Given this statement the visualisations used in this project should serve a purpose and provide benefits over simply viewing the raw data. These benefits are discussed in [75], and for clarity are listed below. A visualisation should:

- answer a question
- pose new questions
- explore and discover
- support decisions
- communicate information
- increase efficiency
- inspire

Visualisations can take many different forms, each with properties that make them more or less suitable for displaying certain data. A review of visualisation theory conducted in [62, Ch. 2] highlights how certain visualisation properties such as colour, motion, size and proximity can be used to draw a viewer’s attention to elements of the visualisation.

An outline of the visualisations used in the project, including screenshots of the implementation, can be found in chapter 6 and throughout chapter 7.

2.4 Network Intrusion Detection System Signatures

A network Intrusion Detection System (IDS), also known as a NIDS, evaluates network traffic to decide whether it is benign or malicious. Many network IDS systems such as Snort [120] and Suricata [38] use a database of signatures to make this decision. Network IDS signatures are easy
to deploy and update (e.g. on an IDS at an organisation’s gateway), can be used to detect malware activity before a successful infection (e.g. reconnaissance and scanning), and can be as vague or precise as needed (e.g. very specific signatures can be used to identify communications from a particular version of malware whereas a looser signature may cover an entire malware family). Network IDS signature packs may require a subscription to an organisation that produces them [16, 131], may be freely downloaded [130] or may be written by an individual familiar with the signature syntax.

An example network IDS signature, written using Snort syntax, is as follows:

\texttt{alert tcp any any -> any 80 (msg:"My signature"; content:"POST"; content:"\texttt{aabb45ff} ";)}

This will alert when a TCP packet with a destination port of 80, containing the string “POST” and the binary data “aabb45ff” (represented as hexadecimal) is examined by the network IDS system. Typically such systems are deployed on a network gateway or receive a mirror of traffic from a router that supports port mirroring. This enables a network IDS system to cover multiple computers/workstations without having to be installed on each one.

Network signatures do not feature heavily in this project, although their use to identify behaviour observed in chapter 7 is commented upon. There are two different ways in which network signatures can be used to complement the work undertaken. The first is as described in existing research that demonstrates techniques for automatic network IDS signature generation [91, 101, 107, 110]. For example, after clustering network traffic (see chapter 4) it is possible to generate signatures that identify traffic within a cluster. This point is revisited in sections 4.2.1 and 8.1.

The second way that network signatures can be used to complement this work is to use them as a method of feature extraction. Section 3.3.1 describes the feature extraction implemented for this project. In addition, it is possible to run all network traffic through a network IDS, such as Snort, and record which signatures hit on which traffic. The signature ID (or human-readable title) can then be used as a feature for that packet/session. This provides a convenient way of identifying known malware for which network signatures are already available. Questions such as “how much of the traffic for this malware sample has already been identified?” can then be answered. This was not implemented in this project due to time constraints but would make for relevant future work and complement what has been implemented.

2.5 Feature Extraction

Many network IDS signatures are written to look for matches in specific protocol fields, such as an HTTP method, Uniform Resource Indicator (URI) or User-Agent. For example, take the signatures presented in [107, p. 17]. The majority of these look for a specific value of at least one protocol field. Three of these are listed below:

- \texttt{“content:"POST"; http.method;”} - requires the HTTP method to be “POST”
- \texttt{“content:"/picture.php"; http.uri”} - requires an HTTP URI of “/picture.php”
- \texttt{“dsize:13;”; content:"\texttt{04000001050000000007000100}”} - requires the packet payload size to be 13 bytes and contain the binary data “04000001050000000007000100” (represented as hexadecimal). The payload size is itself a protocol field.

Thus, malware may be identified through properties of its network traffic. Malware reports issued by anti-virus companies [15, 68, 128] often include information that can be used to identify network traffic that belongs to malware. Even if infrastructure (such as domains and IP addresses) used by the malware is all that is reported, it still appears as a feature within the network traffic.
Note: this report uses the term feature to mean a protocol field, such as a TCP port number or an HTTP method, or another attribute of the network traffic, such as the packet length/size or the protocol name. Feature and field are often used interchangeably throughout this report. A feature value refers to the value of the particular feature. For example, if the feature is a destination IP address then a feature value might be ‘1.2.3.4’.

The approach taken in this project is to extract protocol features from network traffic and to make these available for analysis. The raw extracted data may be viewed directly or it may be further processed, for example by calculating the distribution of values for a particular feature or clustering traffic based on the different features it possesses.

This approach allows the behaviour of the traffic as a whole to be studied, whilst also enabling in depth examination of particular features that may be of interest to a malware analyst. Common features can be exposed, for example revealing the most common packet size or the most common User-Agent present in a network capture. Conversely, the least common values can also be identified, which may be of use if malicious traffic and non-malicious traffic is present in the same data. Trends in the data may be identified and isolated to one or more features, for example if different malware samples fail to properly implement a protocol and transmit network traffic that differs from ‘ordinary’ traffic. One such example is presented in section 7.4.4 where the tendency for malware to use the incorrect letter case for HTTP headers is identified.

To avoid pre-selecting the features available for analysis, and in the process missing some which may be important, as many features as possible are extracted by the framework developed. All of these are made available for analysis. The complete list of features extracted from the main dataset studied is available in Appendix S.

2.6 Networking Protocols and Programming Languages

Before commencing this project I already had a thorough technical understanding of a number of different networking protocols, including Internet Protocol (IP) version 4, TCP, UDP, DNS and HTTP. These protocols are prevalent in the data analysed during this project so I was able to draw upon my experience to interpret them. Where necessary the relevant protocol specifications and RFCs were studied, e.g. when implementing an HTTP parser to extract session protocol features RFC 2616 [29] and its successors [30, 31] were consulted.

I have experience in programming using a variety of different languages, including the ones used during this project, and so was able to use this when implementing the framework. I did, however, have to learn how to use specific software libraries and research the capabilities they provide, for example the visualisation libraries discussed in section 5.3.2.
Chapter 3

System Design

This section of the report describes the design of the framework and is complemented by chapter 5, which describes its implementation.

3.1 Overall System Design

The design of the system has been made as modular as possible. It is split into two main components: the capture framework (see section 3.2) and the analysis framework (see section 3.3).

It is possible to use the analysis framework without the capture framework if existing network captures are available. The capture framework takes as input a file (such as a Windows executable) and outputs a capture of any network traffic generated by the file, in pcap format. This pcap file is then used as input to the analysis framework, at which point statistics about the network traffic are made available and can be used to produce visualisations and cluster similar traffic. Figure 3.1 shows the data flow through the full system. This separation enforces a clear divide between the two components developed.

A web service is provided for the capture framework and a separate one for the analysis framework. These web services form the basis of the API. All data sent between a client and the API is in JSON [59] format, transmitted using HTTP.
3.2 Capture Framework Design

As stated in project objective 6, the purpose of the capture framework is “to run malware over a specified time period and to capture the resulting network traffic”.

This objective can be broken down into three constituent parts:

1. run malware
2. a specified time period
3. capture the resulting network traffic

Point 1 describes the input to the capture framework, which in this case is malware. This does not mean that non-malicious files cannot be run, but they are not the focus of this project.

Point 2 ensures that the time for which a given input file is run is configurable. This is necessary as some malware does not immediately produce network traffic or perform its malicious activity, instead it may wait for a defined period of time or use stalling code as explained in [61]. By allowing the capture framework to run for a longer period of time the chance of generating network activity should be increased. Note, however, that this is not an in depth attempt to specifically capture traffic from malware samples that delay their malicious behaviour. It is a useful feature to implement in its own right as it allows network traffic to be captured that may only ever happen after a period of time or after subsequent malware is downloaded and executed.

Point 3 ensures that any network traffic that is generated by an input file is recorded and it is this that will form the input to the analysis framework.

Figure 3.2 shows the capture framework design. All access to the capture framework occurs via its API and uses Representational State Transfer (REST) principles. The API is a RESTful web service. For a detailed description of REST refer to [109].

The design adheres to the objective discussed above and uses Cuckoo Sandbox [34, 37] as the main technology for generating and capturing any network traffic produced by the input file. In brief, Cuckoo Sandbox allows a file to be run inside a virtual machine and its activity monitored. In this case only the network activity is monitored, but Cuckoo Sandbox allows monitoring of a wide range of activities such as file system changes, processes spawned and memory used. For further discussion on Cuckoo Sandbox see section 5.1.1.

To submit a file to the capture framework the following is information is necessary:

- the input file base64 encoded
- the name of the input file, e.g. myexecutable.exe
- a human-readable description of the input file, for instance the name of the malware family
- a timeout (specified in seconds)

By supplying the input file base64 encoded, binary data can be handled with ease. The timeout specified controls how long the Cuckoo Sandbox virtual machine runs. It must be a minimum of 10 seconds to allow the virtual machine to start and adequate time for any network activity to take place.

Implementation specific details for Cuckoo Sandbox are specified in section 5.1.2.
Once an input file has been run within Cuckoo Sandbox, a check is performed to see if any network traffic was generated. If not, no further action is taken. However, if network traffic has been generated then it is retrieved from Cuckoo Sandbox using its REST API [35] and uploaded to the analysis framework.

The endpoints provided as part of the capture framework API are described in Table 3.1. All data sent between a client and the API is in JSON format, transmitted using HTTP. Thus, the API may be run on a server remotely from any clients.

<table>
<thead>
<tr>
<th>Endpoint URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/capture_api/help</td>
<td>Return a list of available endpoints</td>
</tr>
<tr>
<td>/capture_api/upload</td>
<td>Upload a file to the sandbox. HTTP POST required.</td>
</tr>
</tbody>
</table>

Table 3.1: Capture framework API endpoints

The design and implementation of the API fulfils project objective 7.

### 3.2.1 Summary of Capture Framework Design Decisions

This section briefly summarises and justifies the important design decisions made for the capture framework.

1. *a RESTful API is exposed to clients* - by utilising existing web technologies there are many clients, such as cURL [19], Wget [115] and any web browser, that already support interaction with the API. A client can be remote from the API.
2. *all external interaction occurs via the API* - this allows a central point of control and validation for all input and output.

3. *JSON is used as the data format to transmit data into and out of the capture framework* - it was chosen due to the high availability of libraries that can process the format. It also has the added advantage that it is easy to interpret by a human which can be useful when debugging problems. Binary data can be transmitted either as unicode (e.g. “\u000a”) or by using an encoding mechanism that converts binary data to a valid unicode string.

4. *the input file is base64 encoded* - this avoids any problems with transmitting binary data using HTTP or JSON. It does, however, involve a 33% expansion of the data, but as most input files (e.g. Windows executables) used in this project are only a few megabytes in size this is not a problem.

5. *a timeout is required for all submissions* - this ensures the minimum time necessary to load the virtual machine whilst allowing samples to be run for a long period of time if desired.

6. *the file is run inside a virtual machine* - using a virtual machine, rather than a physical machine, to run the malware samples has a number of advantages. Firstly, a virtual machine can be reset to a clean state after the sample has been run. If using a physical machine the operating system would likely have to be reinstalled unless every action the malware sample could take is fully understood; rootkits in particular can subvert an operating system kernel leading to loss of trust in the core operating system. Secondly, by using a virtual machine it is possible to run multiple samples at the same time using one physical machine. This wasn’t done during the course of the project but the design allows it by utilising Cuckoo Sandbox’s capabilities.

### 3.3 Analysis Framework Design

As stated in project objective 1, the purpose of the analysis framework is *“to extract information of interest from previously captured network traffic”* and *“to produce visualisations to aid analysis”*. This objective can be broken down into three constituent parts:

1. extract information of interest
2. previously captured network traffic
3. produce visualisations to aid analysis

Point 1 ensures that the information available from the analysis framework be of relevance to malware traffic analysis. Feature extraction is covered in section 3.3.1.

Point 2 describes the input to the analysis framework, which in this case is a network traffic capture. It does not matter how or where the network traffic was captured - this could be from the capture framework or elsewhere. Section 5.2.1 discusses the chosen capture format, pcap, in more detail.

Point 3 directly ties into this project’s title - analysis and visualisation. Visualisations are not directly produced by the API, instead the web client (see section 3.4) transforms the data received from the API into suitable visualisations. Thus, the web client can be considered integral to the overall analysis framework.

Figure 3.3 shows the analysis framework design. All access to the analysis framework occurs via its API and uses REST principles. The API is a RESTful web service.
The design adheres to the objective discussed above and consists of three main interaction types: uploading a network capture to the system, viewing metadata about the upload (such as number of packets or TCP sessions within the capture) and analysis (such as retrieving features from the upload(s) or clustering data). Any analysis operation that can be done on one upload can also be performed on multiple uploads. For example, a histogram on TCP destination ports can be performed across all packets within an upload, or across all packets within five uploads.

To submit a pcap file to the analysis framework the following information is necessary:

- the pcap file base64 encoded
- the name of the pcap, e.g. ‘ShylockMalwareCapture’
- a human-readable description of the traffic capture, perhaps providing the malware family name or brief details related to the traffic capture, e.g. ‘One hour snapshot of Shylock malware August 2014’
- the type of the capture format, currently only pcap is supported but this could be extended to include other formats such as pcapng [49]

By supplying the pcap base64 encoded, the binary data can be handled easily with only a 4/3 expansion of data. Base64 was chosen due to it being extremely common and well supported in most programming languages. The name and description of the pcap are there to distinguish between different uploads and inform an analyst, there are no requirements on the format they must take. However, they are required.

Once a pcap has been uploaded it is placed in a file store. Metadata about the pcap can be retrieved from the relevant API endpoint. The following information is available:

- id - the id of the upload within the database, a one up number
- name - the name of the pcap specified when uploaded
- description - the description of the pcap specified when uploaded
- upload type - the type of the uploaded network capture
- number of packets - the number of packets contained within the pcap, only available once the pcap has been processed
- number of sessions - the number of TCP sessions contained within the pcap, only available once the pcap has been processed
- have packets been processed - whether the pcap has been processed to extract packet features or not
- processed packets file path - the file store path to the processed packet features
- have sessions been processed - whether the pcap has been processed to extract TCP session features or not
- processed sessions file path - the file store path to the processed TCP session features

Before any analysis can take place, such as generating histograms or clustering, the uploaded pcap must be processed. The pcap is sent to two different processing engines, one to extract packet features and one to extract TCP session features (see section 3.3.1). These features are then placed in a file store and the path to the files is recorded in the database. The database entry for
the pcap is marked as having the packets and sessions processed. The API endpoints for analysis can then be called.

![Analysis framework design diagram](image)

Figure 3.3: Analysis framework design. Arrows are used to show the flow of data around the system.

The endpoints provided as part of the analysis framework API are described in Table 3.2, Table 3.3 and Table 3.4. All data sent between a client and the API is in JSON format, transmitted using HTTP. Thus, the API may be run on a server remotely from any clients.

<table>
<thead>
<tr>
<th>Endpoint URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/api/help</td>
<td>Return a list of available endpoints</td>
</tr>
<tr>
<td>/api/uploads/reset_all_processing</td>
<td>Reset all processing, forces all uploads to be reprocessed at next access</td>
</tr>
<tr>
<td>/api/uploads/⟨upload_ids⟩/force_process</td>
<td>Force processing of the upload id(s) if they haven’t yet been processed. ⟨upload_ids⟩ should be a comma separated list of upload ids that are desired, or a range separated by a hyphen. It may only be one id. It may also be the keyword “all”.</td>
</tr>
</tbody>
</table>

Table 3.2: Analysis framework API miscellaneous endpoints. Values enclosed in ⟨⟩ are variables that form part of the URI path.
<table>
<thead>
<tr>
<th>Endpoint URI</th>
<th>Description</th>
<th>URI query parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/api/upload</td>
<td>Upload a pcap file. This is the only endpoint which requires use of an HTTP POST rather than an HTTP GET.</td>
<td>N/A</td>
</tr>
<tr>
<td>/api/uploads/</td>
<td>Return the metadata of all uploaded pcaps</td>
<td>process - if set to “true” the pcap is processed using the feature extractors before metadata is returned</td>
</tr>
<tr>
<td>/api/uploads/⟨upload_ids⟩</td>
<td>Return the metadata of the specified uploads. ⟨upload_ids⟩ should be a comma separated list of upload ids that are desired. It may only be one id.</td>
<td>process - if set to “true” the pcap is processed using the feature extractors before metadata is returned</td>
</tr>
</tbody>
</table>

Table 3.3: Analysis framework API upload and metadata endpoints. Values enclosed in ⟨⟩ are variables that form part of the URI path.

<table>
<thead>
<tr>
<th>Endpoint URI</th>
<th>Description</th>
<th>URI query parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/api/uploads/⟨upload_ids_and_pkt_nums⟩</td>
<td>Get the details of the packets specified. ⟨upload_ids_and_pkt_nums⟩ is a semicolon delimited list of upload id to packet number, e.g. 1,2,3,1 means upload 1 packet 2 and upload 3 packet 1</td>
<td>N/A</td>
</tr>
<tr>
<td>/api/uploads/features/session_requests/⟨upload_ids_session_nums_request_nums⟩</td>
<td>Get the request details for the specified session requests. ⟨upload_ids_session_nums_request_nums⟩ is a semicolon delimited list of upload id to session number to request number, e.g. 1,2,1;3,1,1 means upload 1 session 2 request 1 and upload 3 session 1 request 1</td>
<td>N/A</td>
</tr>
<tr>
<td>/api/uploads/features/sessions/⟨upload_ids_and_session_nums⟩</td>
<td>Get the session details for the specified sessions. ⟨upload_ids_and_session_nums⟩ is a semicolon delimited list of upload id to session number, e.g. 1,2;3,1 means upload 1 session 2 and upload 3 session 1</td>
<td>N/A</td>
</tr>
<tr>
<td>/api/uploads/⟨upload_id⟩/features/</td>
<td>Get all session and packet features for the specified ⟨upload_id⟩</td>
<td>N/A</td>
</tr>
<tr>
<td>/api/uploads/⟨upload_id⟩/features/packets/</td>
<td>Get all packet features for the specified ⟨upload_id⟩</td>
<td>N/A</td>
</tr>
<tr>
<td>/api/uploads/⟨upload_id⟩/features/sessions/</td>
<td>Get all session features for the specified ⟨upload_id⟩</td>
<td>N/A</td>
</tr>
<tr>
<td>/api/uploads/⟨upload_ids⟩/fields/</td>
<td>Get a list of fields in the sessions and packets for the specified uploads. ⟨upload_ids⟩ should be a comma separated list of upload ids that are desired. It may only be one id.</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### Table 3.4: Analysis framework API analysis endpoints. Values enclosed in ⟨⟩ are variables that form part of the URI path.

<table>
<thead>
<tr>
<th>Endpoint URI</th>
<th>Description</th>
<th>URI query parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/api/uploads/⟨upload_ids⟩/histogram/⟨field_name⟩</code></td>
<td>Histogram the values for the specified field name across the specified upload id(s). <code>&lt;upload_ids&gt;</code> should be a comma separated list of upload ids that are desired. It may only be one id. The <code>&lt;field_name&gt;</code> should be one returned by a call to <code>/api/uploads/⟨upload_ids⟩/fields/</code></td>
<td><code>sortOrder</code> - one of “field_ascending”, “field_descending”, “count_ascending” or “count_descending”. <code>type</code> - one of “count”, “percent”, “num_uploads” or “percent_uploads”</td>
</tr>
<tr>
<td><code>/api/uploads/⟨upload_ids⟩/session_cluster</code></td>
<td>Cluster the session HTTP request protocol data in the upload or uploads. <code>&lt;upload_ids&gt;</code> should be a comma separated list of upload ids that are desired. It may only be one id.</td>
<td><code>clusteringType</code> - one of “K_CLUSTERS” or “DISTANCE_THRESHOLD” <code>k</code> - the number of clusters to partition by if using “K_CLUSTERS” <code>randomMultiLongestEdges</code> - if set to “true” and there are multiple longest edges, randomly choose an edge to cut (non-deterministic), otherwise always cut the first longest edge (deterministic) - only if using “K_CLUSTERS” <code>threshold</code> - A floating point number used with “DISTANCE_THRESHOLD” clustering. Represents the maximum allowed distance between session requests in the minimum spanning tree (inclusive).</td>
</tr>
<tr>
<td><code>/api/uploads/⟨upload_ids⟩/transport_cluster</code></td>
<td>Cluster the transport layer protocol data in the upload or uploads. <code>&lt;upload_ids&gt;</code> should be a comma separated list of upload ids that are desired. It may only be one id.</td>
<td>N/A</td>
</tr>
</tbody>
</table>

The design and implementation of the API fulfils project objective 2.

#### 3.3.1 Feature Extraction

There are two feature extraction engines within the analysis framework. One operates on packets, the other on TCP sessions.
Packets vs Sessions

TCP [105] is a session-oriented protocol. Information is transmitted as a stream of data that is divided into segments. Each segment is contained within an IP packet. TCP provides a reliable and ordered connection between two applications. TCP sessions need to be considered when designing any system that processes network captures, as packets alone will not give an accurate view of the traffic transmitted. For example, TCP segments may arrive out of order and need reordering. Alternatively, data may span more than one IP packet thus any code attempting to reassemble application layer protocols will need to operate on the reconstructed session spanning multiple packets. To ensure all TCP data within network captures is properly handled this framework utilises the concept of packets and sessions.

- **packet** - a protocol data unit at the network layer, typically an IP version 4 packet. Lower level information (such as Ethernet frames) may also be available.
- **session** - a reconstruction of one or more packets that form a TCP session, typically identified by a four tuple (source IP, source port, destination IP, destination port). A session can have a request and/or a response which represent the payload data contained within the TCP segments.

As UDP is a transaction oriented protocol [104] it is not possible to guarantee that all protocols layered on top of it use the concept of sessions. Therefore, UDP data is not reconstructed into sessions in the analysis framework. In specific cases, such as when working with the DNS protocol, information from the request is also present in the response, thus providing extra information that may be of use without having to reconstruct UDP sessions.

As the session feature extractor used in this project is passed a pcap file as input, it must first reconstruct all TCP sessions present in the network capture. Only then can protocols be identified and features extracted. See section 5.2.4 for a discussion on TCP session reassembly tools and the one selected for use in this project.

Malware communications typically need to reach across a network (such as the internet) rather than across local nodes (at the link layer). As such, the analysis framework drops any datagrams that do not contain IP version 4 before they reach the packet feature extractor. The session feature extractor does not do this although does require traffic to contain TCP segments for obvious reasons. No explicit customisation is made for IP version 6 but it can be processed by the analysis framework if the check for IP version 4 is changed to also include IP version 6. Link layer information (e.g. Ethernet source and destination MAC addresses) is available but is unlikely to be useful, e.g. if the network traffic was collected using the capture framework then the Ethernet MAC addresses would match those used by the capture environment.

Packet Feature Extraction

The following data is always extracted for each packet:

- **timestamp**
- **packet number within the pcap**
- **the protocols/layers within the packet. The order of the protocols is maintained (e.g. Ethernet, followed by IP, followed by TCP)**

Each protocol consists of four pieces of information:
• **name** - user-friendly name

• **short_name** - name all lowercase with no spaces, used internally by the framework for string matching

• **fields** - the features contained within the protocol. For example, the TCP protocol has a “seq” field to represent the TCP sequence number.

• **b64_fields** - any features contained within the protocol which cannot be natively represented in JSON format (i.e. any non-Unicode binary data) are base64 encoded so they can be transmitted in JSON format

A full example of the packet feature data extracted from a DNS packet is shown in Appendix E.

**Session Feature Extraction**

The following data is always extracted for each session:

• session number within the pcap

• source IP address

• source port number

• destination IP address

• destination port number

• request number of packets

• response number of packets

• length of the request payload (request data length)

• length of the response payload (response data length)

• length of the session (request data length added to response data length)

• whether the session was closed with a TCP FIN flag or not

• the request protocols contained within the session

• the response protocols contained within the session

Each session protocol consists of four pieces of information:

• **name** - user-friendly name

• **short_name** - name all lowercase with no spaces, used internally by the framework for string matching

• **data_len** - the application layer data length

• **fields** - the features specific to the protocol. For example, the HTTP Request protocol has a “method” field to represent the HTTP method in use.

A full example of the session feature data extracted from an HTTP packet is shown in Appendix F.
3.3.2 Database Schema

The database schema for the analysis framework is shown in Listing 3.1. This stores the minimum amount of information necessary within the database itself. Table 3.5 describes the meaning of each field.

The fields “packets_path” and “sessions_path” in the “uploads” table point to the JSON files storing packet and session features. The “id” field in the “uploads” table is used to name the uploaded pcap file which is stored in a configurable directory (see section 5.2.5).

<table>
<thead>
<tr>
<th>Table</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>upload_type</td>
<td>id</td>
<td>The table primary key</td>
</tr>
<tr>
<td>upload_type</td>
<td>name</td>
<td>The name of the upload type, e.g. pcap</td>
</tr>
<tr>
<td>uploads</td>
<td>id</td>
<td>The table primary key. Also used to name the uploaded pcap file which is stored in a configurable directory (see section 5.2.5).</td>
</tr>
<tr>
<td>uploads</td>
<td>name</td>
<td>The name given to the upload/pcap</td>
</tr>
<tr>
<td>uploads</td>
<td>description</td>
<td>The description given to the upload/pcap</td>
</tr>
<tr>
<td>uploads</td>
<td>upload_type_id</td>
<td>A foreign key referencing “id” in the “upload_type” table</td>
</tr>
<tr>
<td>uploads</td>
<td>processed_packets</td>
<td>0 if packet features have not yet been extracted from the upload, 1 if they have been extracted</td>
</tr>
<tr>
<td>uploads</td>
<td>packets_path</td>
<td>the path to the packet features in the file store</td>
</tr>
<tr>
<td>uploads</td>
<td>num_packets</td>
<td>the number of packets in the upload, only available once the packets have been processed</td>
</tr>
<tr>
<td>uploads</td>
<td>processed_sessions</td>
<td>0 if session features have not yet been extracted from the upload, 1 if they have been extracted</td>
</tr>
<tr>
<td>uploads</td>
<td>sessions_path</td>
<td>the path to the session features in the file store</td>
</tr>
<tr>
<td>uploads</td>
<td>num_sessions</td>
<td>the number of sessions in the upload, only available once the sessions have been processed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 3.1: Analysis framework database schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE TABLE upload_type (id INTEGER PRIMARY KEY ASC, name TEXT);</td>
</tr>
<tr>
<td>CREATE TABLE uploads (id INTEGER PRIMARY KEY, name TEXT, description TEXT, upload_type_id INTEGER, processed_packets NUMERIC, packets_path TEXT, num_packets INTEGER, processed_sessions NUMERIC, sessions_path TEXT, num_sessions INTEGER);</td>
</tr>
</tbody>
</table>

| Table 3.5: Analysis framework database schema field descriptions |

Table 3.5: Analysis framework database schema field descriptions

3.3.3 Summary of Analysis Framework Design Decisions

This section briefly summarises and justifies the important design decisions made for the analysis framework.

1. a RESTful API is exposed to clients - as described in section 3.2.1
2. all external interaction occurs via the API - as described in section 3.2.1. Other computer programs can use the API in an automated way to generate the data they require.
3. JSON is used as the data format to transmit data into and out of the analysis framework - as described in section 3.2.1
4. the input pcap is base64 encoded - this avoids any problems with transmitting binary data using HTTP or JSON. Additionally, as any protocol could be encountered, and the protocol...
could contain arbitrary binary data, fields extracted from the network traffic capture are encoded using base64 where necessary.

5. a database is used to store metadata about an uploaded pcap, and points to locations within a file store for the pcap file, packet features and session features - by utilising a file store the database does not need to be aware of particular protocols, each protocol can be stored in JSON format. An alternative is to store the JSON as a field within the database - this would likely have performance benefits if lots of concurrent access to the database was necessary.

6. features that can be used for analysis are extracted from packets and sessions separately - as a TCP session is composed of a collection of packets, extra processing is required before protocols can be identified within a session

7. histograms can be produced for features extracted from network traffic - features are key pieces of information contained within protocols (such as an IP source address or an HTTP response status code) that form the input to any histograms produced. How the histogram is visualised is determined by the (web) client.

8. clustering can be performed on transport protocols or on session HTTP requests extracted from network traffic - for a detailed discussion on the clustering available see chapter 4. How the clustering results are visualised is determined by the (web) client.

9. analysis can be conducted for one pcap individually, or for multiple pcaps together - by enabling analysis across pcap files it is possible to conduct extra types of analysis. For example, if a pcap file represents traffic from one malicious executable file, then it is possible to determine if there is any commonality between network traffic generated by two (or more) different pcaps, thus indicating if the executables are part of the same malware family.

10. visualisation is performed using a web client, which can be considered part of the analysis framework - a client is needed that can convert the JSON data into a visualisation suitable for a human to view

### 3.4 Visualisation (Web Client)

Project objective 1 requires visualisations to be produced. Clearly an API that accepts and returns JSON data does not fulfil this requirement. For the analysis framework to be considered complete, a web client is used to transform the data received from the analysis framework API into suitable visualisations. How the web client is positioned in relation to the API can be seen in Figure 3.3.

The web client calls the appropriate API endpoints in response to user interaction. For example, if a user wants to view a pie chart of the source IP addresses in pcap upload 12 the following endpoint is called:

```
/api/uploads/12/histogram/packet.ip.src?sort_order=field ascending&histogram_type=count
```

The web client then processes the JSON data that is returned from the API endpoint and displays a pie chart within the web page.

The intended flow of control through the web client is as follows:

1. load web client
2. choose to select only one upload/pcap or multiple uploads/pcaps
3. browse the available pcaps and select one or more for analysis
4. select the type of visualisation, e.g. a table, a histogram bar chart or a packet transport clusters pie chart

5. select any options for the chosen visualisation, e.g. histogram field, histogram type, label rotation (some visualisations become quite crowded and are easier to interpret if the axis labels are rotated, for example by 45 degrees) or clustering parameters

6. view visualisation

At each step the web client calls the appropriate API endpoint(s) to retrieve the necessary information.

In addition, the web client has a form to allow a pcap file to be uploaded. As for all other functions, this simply calls the relevant API endpoint (in this case, an HTTP POST to /api/upload).

Whilst not strictly part of the analysis framework, the ability to upload a file for submission to the capture framework is included within the web client. This avoids the need to design and implement a separate web client with the sole purpose of uploading a file to be run dynamically. When uploading a file, the web interface allows the choice of waiting for the capture framework processing to complete or to return immediately for further interaction. File submission is achieved by performing an HTTP POST to the capture framework API endpoint /capture_api/upload.

Table 3.6 shows the visualisations available for each type of data. Appendix H expands on this and lists the available options to configure what is displayed by the visualisation and how it is displayed.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Visualisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature histogram</td>
<td>Table</td>
</tr>
<tr>
<td></td>
<td>Bar chart</td>
</tr>
<tr>
<td></td>
<td>Column chart</td>
</tr>
<tr>
<td></td>
<td>Line chart</td>
</tr>
<tr>
<td></td>
<td>Area chart</td>
</tr>
<tr>
<td></td>
<td>Pie chart</td>
</tr>
<tr>
<td>Packet (transport layer) clustering</td>
<td>Table</td>
</tr>
<tr>
<td></td>
<td>Pie chart</td>
</tr>
<tr>
<td>Session (HTTP request) clustering</td>
<td>Table</td>
</tr>
<tr>
<td></td>
<td>Pie chart</td>
</tr>
</tbody>
</table>

Table 3.6: Visualisations available in the web client
Chapter 4

Clustering

This section of the report describes the clustering methods used to meet project objective 5.

4.1 Malware Traffic Clustering

The traffic clustering used in the analysis framework is split into two types: packet clustering and session clustering. This distinction maintains consistency throughout the framework so that packets and sessions can be treated appropriately.

The packet or session object itself is not used directly in clustering results. Instead, a feature vector (a set of features) is used to represent the packet or session. Enough information is included in the feature vector to retrieve the original packet or session if necessary.

A packet feature vector consists of:

\[
\langle \text{pkt\_num}, \text{upload\_id}, \text{proto\_name}, \text{src\_ip}, \text{dst\_ip}, \text{src\_port}, \text{dst\_port}, \text{data\_len} \rangle
\]

\text{pkt\_num} and \text{upload\_id} uniquely identify the packet and the upload (pcap) the packet belongs to. \text{proto\_name} is the transport layer protocol of the packet. This can be either UDP or TCP. \text{src\_ip} and \text{dst\_ip} are the source and destination IP addresses of the IP packet whilst \text{src\_port} and \text{dst\_port} are the source and destination port numbers used by the transport layer protocol. \text{data\_len} corresponds to the data length of the packet, i.e. UDP payload length or TCP payload length - the length of the headers is not included. For UDP packets this can be calculated from information available in the UDP header; for TCP packets, information from both the TCP and IP headers is necessary.

HTTP allows a TCP connection to be used for multiple HTTP requests (and responses). Thus, when a session contains the HTTP protocol, it is possible that the session request contains multiple HTTP requests. This may hold true for other application layer protocols as well. The protocol extraction code and clustering algorithms used in this project take this into account. The features used for a session request feature vector are:

\[
\langle \text{session\_num}, \text{request\_num}, \text{upload\_id}, \text{proto\_name}, \text{src\_ip}, \text{dst\_ip}, \text{src\_port}, \text{dst\_port}, \text{data\_len} \rangle
\]

\text{session\_num}, \text{request\_num} and \text{upload\_id} uniquely identify the specific session and request in an upload (pcap). By using these three values it is possible to uniquely identify the (HTTP) request in each cluster. The requests are allocated one-up numbers which correspond to the order in which they appear in the TCP session. All other features are as described for a packet feature vector.

If clustering on sessions (versus session requests only) is desired then the \text{request\_num} feature can be dropped from the session request feature vector to form a session feature vector. This was not implemented for this project as the only session clustering performed is on HTTP requests.

The analysis framework and the web client support ‘click-through’ to details. If clusters are shown as a pie chart then each segment of the pie (a cluster) can be selected which results in a call to
one of the following endpoints:

- `/api/uploads/features/packets/(upload_ids_and_pkt_nums)` - display the packet features for all packets in the cluster
- `/api/uploads/features/session_requests/(upload_ids_session_nums_request_nums)` - to display the (HTTP) requests

These endpoints were previously described in Table 3.4.

If clusters are shown in a table then clicking on the table shades the rows for every other cluster, separating the clusters visually. This can be seen in Figure 4.1.

![Table showing packet clustering results](image)

**Figure 4.1: Visual separation of clusters achieved by highlighting alternate cluster rows**

### 4.1.1 Packet Clustering

The design of the framework allows different types of input to be used to cluster packets. Only one type of packet clustering is implemented but the framework is sufficiently extensible and modular that adding more should be straightforward. Any features extracted could be included as part of a packet clustering algorithm, e.g. `packet.tcp.flags` and `packet.ip.ttl`.

The packet clustering implemented for this report is based on transport layer protocols. This takes as input all packets (in one or more pcaps) and groups them into clusters containing similar transport layer features. The selection of features is based on that described in [107, p. 9] with only slight modification (Wireshark is not used for protocol identification in this framework). Packets are grouped if they meet the following criteria:

- they have the same transport protocol (only UDP and TCP are considered)
- they have the same size (data length)
they have the same destination IP address
they have the same destination port

Packets which have these four features identical form a cluster. There is no upper limit to the number of clusters generated by the algorithm. Pseudocode illustrating how the clustering is performed is shown in Listing 4.1.

```python
all_clusters = []
for packet in packets:
    if packet.protocol is "UDP" or "TCP":
        if this packet should be placed in an existing cluster:
            add packet feature vector to existing cluster
        else:
            create a new cluster where cluster.protocol = packet.protocol and
            cluster.data_len = packet.data_len and cluster.dst_ip = packet.dst_ip
            and cluster.dst_port = packet.dst_port:
            add packet feature vector to new cluster
            add new cluster to all_clusters

return all_clusters
```

Listing 4.1: Pseudocode showing the packet clustering algorithm

4.1.2 Session Clustering

The design of the framework allows different types of input to be used to cluster sessions. Only one type of session clustering is implemented but the framework is sufficiently extensible and modular that adding more should be straightforward. As with packet clustering, any features extracted could be included as part of a session clustering algorithm, e.g. session.fin_closed.

The session clustering implemented for this report is based on session HTTP requests. This takes as input all HTTP requests (in one or more pcaps) and groups them into clusters containing similar HTTP request features. The selection of features is based on the fine-grained clustering described in [101] and consists of the following:

- HTTP request method
- HTTP URL path
- HTTP parameter names
- HTTP parameter values

For example, Listing 4.2 shows an example HTTP request line. The request method is “GET”, the URL path is “/path/to/file.html”, the parameter names are “name” and “id” and the parameter values are “test” and “42”.

```
GET /path/to/file.html?name=test&id=42
```

Listing 4.2: Example HTTP request line

To compare two HTTP requests a distance metric is used. The distance is calculated exactly as for fine-grained clustering in [101]. In summary:
the distance between two HTTP request methods is 0 if the same method is used, 1 otherwise

the distance between two HTTP URL paths is set to the normalised Levenshtein distance between the URL paths

the distance between two sets of HTTP parameter names is the Jaccard distance between those sets

the distance between two lists of HTTP parameter values is “the normalized Levenshtein distance between strings obtained by concatenating the parameter values” [101]

According to [101, p. 4] “The normalized Levenshtein distance between two strings $s_1$ and $s_2$ (also known as edit distance) is equal to the minimum number of character operations (insert, delete, or replace) needed to transform one string into the other, divided by $max(length(s1), length(s2))$” and the Jaccard distance “between two sets $A$ and $B$ is defined as $J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$.

A weighting is then applied to each of the four distances calculated, as in [101]. This allows different levels of importance to be attached to the four distances. More importance (a higher weighting) is given “to the distance between the request methods and pages, for example, and less weight to the distance between parameter values” [101]. The overall distance between two HTTP requests is then calculated by adding the four weighted distances together. The actual weight values used are the same as in [101]. The implementation does not allow these to be configured when clustering is performed, however this would be a trivial change to make. Pseudocode illustrating the distance calculation is given in Listing 4.3.

```
if request1 method == request2 method:
    method distance = 0
else:
    method distance = 1

url path levenshtein distance = calculate normalised levenshtein
distance(request1 url path, request2 url path)

url parameter names jaccard distance = calculate Jaccard distance(request1
parameter names set, request2 parameter names set)

url parameter values levenshtein distance = calculate normalised levenshtein
distance(request1 url parameter values concatenated, request2 url parameter
values concatenated)

distance = (method weight * method distance) +
(url path weight * url path levenshtein distance) +
(url parameter names weight * url parameter names jaccard distance) +
(url parameter values weight * url parameter values levenshtein distance)

return distance
```

Listing 4.3: Pseudocode showing the distance calculation for a pair of HTTP requests

Once the distance between all HTTP requests has been calculated the requests need to be placed into clusters. In [101] this is performed using single-linkage hierarchical clustering. In this project the clustering is performed in the following way:
calculate the minimum spanning tree between all pairs of HTTP requests. A spanning tree of an undirected graph connects all vertices of the graph. In this case, each HTTP request is a vertex. The distance between a pair of HTTP requests is used as the weight for the edge that connects the vertices. A minimum spanning tree is the spanning tree which has the smallest possible total weight. Kruskal’s algorithm [64] is used to calculate the minimum spanning tree.

cut the minimum spanning tree according to the clustering type:

- $k$ clusters - cluster into $k$ clusters, where $2 \leq k \leq n$ and $n$ is the total number of requests. To cluster a minimum spanning tree into $k$ clusters, the $k - 1$ longest edges are cut. Anything still connected forms a cluster. This method is deterministic if, when performing any given cut, there exists only one longest edge. Otherwise the algorithm can be made deterministic (by always cutting the first longest edge encountered) or non-deterministic (by choosing a random longest edge to cut).

- distance threshold - the number of clusters is not fixed. The minimum spanning tree is cut wherever any edge has a value greater than a threshold $t$. The threshold is configurable but by default is set to 7.0. Based on testing conducted this seems like a sensible value. This method is deterministic - the same threshold value will always result in the same clusters, given the same input. The lower the threshold value, the more similar requests must be to cluster together. If two HTTP requests are identical (with respect the features measured) then the distance between the two will be 0 (in the code this is actually 0.00001 for implementation reasons).

output the clusters based on the final minimum spanning tree. Any vertices still connected form a cluster. The clusters contain the session request feature vector for each request in the cluster.

Pseudocode illustrating the clustering algorithm is given in Listing 4.4.

```
1 return None if number of requests is 0 or 1

2 for each request:
3   for every other request not yet checked in outer for loop:
4     calculate distance between the two requests

5 calculate the minimum spanning tree for all request pair distances using
6   Kruskal’s algorithm

7 if using $k$ clusters:
8   if $k < 2$:
9     $k = 2$
10   else if $k >$ number of requests:
11     $k = \text{round}(\sqrt{\text{number of requests} / 2})$
12
13   remaining cuts = $k - 1$
14 while remaining cuts != 0:
15   get the longest edge in the minimum spanning tree
16   if longest edge == 0:
17     break while loop as no more edges can be cut
18   edge to cut = longest edge
19 if number of longest edges > 1 and non-deterministic:
```
edge to cut = random choice from the longest edges

cut the edge to cut in the minimum spanning tree

remaining_cuts -= 1

else if using distance threshold:
    edges to cut = get all edges in the minimum spanning tree where distance > t

for each edge to cut:
    cut the edge in the minimum spanning tree

using the final minimum spanning tree, create a cluster for each group of
vertices still connected. If a vertex is not connected to anything then it
forms its own cluster. The cluster contains the session request feature
vectors.

return the clusters

Listing 4.4: Pseudocode showing the session HTTP request clustering algorithm

4.2 Traffic Capture Types and Clustering

This project uses clustering to group similar network traffic together. This can be performed on
the following types of malware traffic captures:

- **within a pcap from one malware sample** - examination of network traffic from a specific piece
  of malware, useful to understand how it operates. This can potentially highlight features
  that can form network IDS signatures to identify the specific malware.

- **between pcaps from the same malware family** - allows examination of malware samples to
  see how communications differ between different instances of the malware. For example, this
  might highlight how URI parameters change between specific installations.

- **between pcaps from related malware families** - can highlight any similarities between families,
  possibly hinting at shared origins (e.g. a source code base that diverged but the networking
  functionality remained unchanged)

- **between pcaps from unrelated malware families** - allows identification of communication tech-
  niques that may be prevalent across a wide number of malware families

- **across the entire dataset** - provides overall statistics and trending for all data examined

4.2.1 IDS Signature Generation for Malware Clusters

As clustering moves from a specific malware sample to family wide samples, through to inter-
family samples and then eventually to all samples, false positives for the IDS signatures that
may be generated are likely to increase. With a varied enough collection of network traffic, the
commonalities shrink to only include features that occur in the majority of traffic, legitimate or
malicious. Based on observations made when conducting the analysis for this report, the types of
IDS signatures that can be generated are likely to be as speculated in Table 4.1. The wider the
variety of malware samples included in the clustering input, the less specific any common features are likely to become.

<table>
<thead>
<tr>
<th>Signature coverage</th>
<th>Malware variety</th>
<th>Example feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very specific</td>
<td>Individual malware sample</td>
<td>Domain name</td>
</tr>
<tr>
<td>Family wide</td>
<td>Multiple samples from the same malware family</td>
<td>HTTP URI</td>
</tr>
<tr>
<td>Related families</td>
<td>Multiple samples from related malware</td>
<td>Protocol used combined with other protocol features (e.g. HTTP URI parameter names)</td>
</tr>
<tr>
<td>Generic malware heuristics</td>
<td>Select unrelated malware samples</td>
<td>HTTP response containing obfuscated JavaScript</td>
</tr>
<tr>
<td>Nothing unique enough to signature</td>
<td>The entire dataset</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 4.1: Possible IDS signature coverage by malware variety. This is speculation based on observations made when conducting analysis for this report.

4.3 Clustering Summary

The clustering described in this chapter is available as part of the analysis framework. It does not attempt to classify network traffic as malicious or not, but to partition the data into related communications. By studying similar traffic, a malware analyst may be able to separate benign from malicious traffic, as well as identify malware samples that make requests to legitimate services (e.g. an internet connectivity check). Network IDS signatures may be written to identify traffic within individual clusters.

If clustering is performed across traffic captured from multiple malware families then it may be possible to cleanly separate the traffic from the different families, again aiding analysis. It is also possible that traffic from different malware may cluster together, indicating common ancestry or communication techniques used by multiple families.

Finally, the framework is designed in such a way that new clustering algorithms can easily be added alongside the existing ones. Clustering of packets or sessions is possible and all features extracted are available to be used by clustering algorithms.

The results of the implemented clustering algorithms can be seen in chapter 7, where traffic from a variety of malware is clustered and the findings reported.
Chapter 5

Implementation

This section details the implementation of the design described in chapter 3. It describes any implementation-specific choices made and why, explaining any effect this has on the framework where applicable.

5.1 Capture Framework

5.1.1 Choice of Sandbox

Cuckoo Sandbox is used as the system for capturing network traffic from malware samples. However, it is possible to replace Cuckoo Sandbox with other technologies as the capture framework API hides any implementation specific details. Thus, if an alternative dynamic malware analysis system became available that offered new capabilities (such as the ability to execute extra file types), or performance improvements, then the framework could be changed to use it and existing clients would not be aware a change had occurred.

An executable could even be run in multiple malware analysis systems (or multiple virtual machines within one system), leading to multiple network traffic captures. This could be useful to detect malware which behaves differently depending upon the environment (e.g. on different operating systems). However, this is not within the scope of the project and so is not investigated further.

5.1.2 Cuckoo Sandbox

Cuckoo Sandbox has been configured for specific use with this project. The changes made are detailed in Table 5.1 and mostly revolve around removing processing that Cuckoo Sandbox is capable of, but which is not necessary for the purpose of integrating it into the capture framework solely to generate a network capture from an input file. Additionally, the sniffer module was modified to ignore Cuckoo Sandbox’s own traffic. This prevents it affecting any follow on analysis or clustering.

A virtual machine running the Windows XP (32 bit) operating system, Service Pack 3, was set up and configured to be used by Cuckoo Sandbox. Cuckoo Sandbox requires an agent installed in the virtual machine so that it can receive files to be run, as well as control other processing. The Python interpreter [39] was installed, followed by the Cuckoo Sandbox agent.

VirtualBox [95] is used to run the virtual machine. It is possible to run different virtual machines simultaneously, although this was not implemented in this project. The design does, however, make this possible should a higher malware sample throughput be desired.

The host system that runs Cuckoo Sandbox requires virtual networking to be set up to allow the guest system to access the internet. This is described in detail in Appendix D.
<table>
<thead>
<tr>
<th>File</th>
<th>Change</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuckoo/conf/auxiliary.conf</td>
<td>bpf = not arp and not (udp src port 137 and udp dst port 137)</td>
<td>Prevent ARP and NetBIOS Name Service traffic from being recorded</td>
</tr>
<tr>
<td>cuckoo/conf/cuckoo.conf</td>
<td>version_check = off</td>
<td>Prevent Cuckoo Sandbox from checking for new versions at startup</td>
</tr>
<tr>
<td></td>
<td>connection = sqlite:///database.db</td>
<td>Set the database used by Cuckoo Sandbox to be an SQLite file</td>
</tr>
<tr>
<td>cuckoo/conf/processing.conf</td>
<td>[analysisinfo] enabled = no</td>
<td>Disable unnecessary processing modules</td>
</tr>
<tr>
<td></td>
<td>[behavior] enabled = no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[debug] enabled = no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[dropped] enabled = no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[memory] enabled = no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[network] enabled = no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[static] enabled = no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[strings] enabled = no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[targetinfo] enabled = no</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[virustotal] enabled = no</td>
<td></td>
</tr>
<tr>
<td>cuckoo/conf/reporting.conf</td>
<td>[jsondump] enabled = no</td>
<td>Disable unnecessary reporting modules</td>
</tr>
<tr>
<td></td>
<td>[reporthtml] enabled = no</td>
<td></td>
</tr>
<tr>
<td>cuckoo/conf/virtualbox.conf</td>
<td>mode = headless, path = /usr/bin/vboxmanage</td>
<td>Run the VirtualBox virtual machine with no visible interface (to save system resources) and fix an incorrect default path to the vboxmanage executable. Note, while conducting the analysis for this report the virtualbox mode was set to “gui” for debugging purposes.</td>
</tr>
</tbody>
</table>
File Change Purpose
---
cuckoo/
modules/
auxiliary/
sniffer.py

Replace lines 57 and 58 with:

```
pargs.extend([“and”,
“not”,“(”,“dst”,“host”,
host,“and”,“dst”,“port”,
str(CUCKOO_GUEST_PORT),
“)”,“and”,“not”,“(“,“src”,
“host”,host,“and”,“src”,“port”,
str(CUCKOO_GUEST_PORT),
“)”])
```

Fix the Cuckoo Sandbox network capture so that its own XMLRPC agent traffic is not captured

<table>
<thead>
<tr>
<th>File Change Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fix the Cuckoo Sandbox network capture so that its own XMLRPC agent traffic is not captured</td>
</tr>
</tbody>
</table>

Table 5.1: Changes made to the default Cuckoo Sandbox configuration files

### 5.1.3 Configuration Options

The following options are configurable for the capture framework:

- **DEBUG** - toggles debug mode. All work was done with this set to “True” as it provides a built in web server. If set to false an external web server is required to host the Python code.

- **COMPRESS_DEBUG** - allows gzip compression in debug mode if set to “True”

- **FILE_SUBMISSION_SCRIPT** - the location of the script that manages file submissions to Cuckoo Sandbox and then submits the generated pcap to the analysis framework

- **TMP_DIR** - a temporary directory to write the uploaded file to

Table 5.2 explains how the configuration can be changed using an environment variable and external file. Alternatively, the default configuration file can be edited directly.

### 5.1.4 Source Code

Code written for the capture framework uses the following programming languages:

- Python 3
- Bourne shell
- Bash shell

Table 5.2 describes the purpose of each source code file.

Refer to Appendix C for a breakdown of other software used.
5.2 Analysis Framework

5.2.1 Network Capture Format

There are a variety of packet capture formats capable of storing network traffic [13, 41, 49, 126]. This project uses the libpcap format due to its popularity and integration into a large number of programming libraries and utilities. However, the design of the framework is such that another capture format can easily be used - whenever a packet capture is uploaded to the analysis framework the format of the file is specified as part of the upload. Support for a format other than libpcap can be achieved by adding an extra row to the database table “upload_type” (see section 3.3.2) and integrating new programming code to read the capture format into the analysis framework.

5.2.2 Database

SQLite [122] is used as the database engine for the analysis framework. Situations when SQLite might be an appropriate database engine to use are presented in [121], and this project suits some of those situations, in particular to “Stand-in for an enterprise database during demos or testing”. Installing the software and getting a working database running takes very little time and this is the primary reason for selection of SQLite. Adding in support for an alternative database engine is possible; most of the changes required are to the “database.py” Python module.

When the analysis framework is run and the web interface root URI (“/”) accessed, it creates a connection to the database. If the database does not yet exist it is created along with the relevant tables and initial data. The SQL for this is shown in Listing 5.1.

<table>
<thead>
<tr>
<th>File</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>/sandnet2/analyse.bash</td>
<td>Submits a file to Cuckoo Sandbox, waits until processing is complete then checks to see if there is a pcap file for the sample. If there is the script submits it to the analysis framework API.</td>
</tr>
<tr>
<td>/sandnet2/capture_framework/ runserver.py</td>
<td>Loads the configuration for the capture framework and runs a web server on localhost, i.e. starts the capture framework API</td>
</tr>
<tr>
<td>/sandnet2/capture_framework/ runserver.sh</td>
<td>Runs runserver.py with the appropriate PYTHONPATH Linux environment variable</td>
</tr>
<tr>
<td>/sandnet2/capture_framework/ src/sandnet/cf/api/api.py</td>
<td>The capture framework API</td>
</tr>
<tr>
<td>/sandnet2/capture_framework/ src/sandnet/cf/api/config.py</td>
<td>The capture framework configuration options. These settings are overridden if the environment variable “SANDNET_CAPTURE_API_CONFIG” points to a python file containing a custom configuration.</td>
</tr>
</tbody>
</table>

Table 5.2: Capture framework source code file listing

---

CREATE TABLE upload_type (id INTEGER PRIMARY KEY ASC, name TEXT);
INSERT INTO upload_type VALUES(1,'pcap');
CREATE TABLE uploads (id INTEGER PRIMARY KEY, name TEXT, description TEXT, upload_type_id INTEGER, processed_packets NUMERIC, packets_path TEXT, num_packets INTEGER, processed_sessions NUMERIC, sessions_path TEXT, num_sessions INTEGER);

Listing 5.1: Analysis framework initial database SQL

Page 36 of 179
5.2.3 Choice of Packet Processing Code Library

The packet feature extractor (scapy_packet_reader.py) is the only piece of code that is written in the Python 2 programming language. The Scapy library [9] is only available for Python 2. A Python 3 port does exist [102], however the project was only started on 29/01/2015, by which point scapy_packet_reader.py was already complete.

Scapy is a very powerful program with lots of functionality. According to [9], it can “forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more”. This makes Scapy an ideal choice for use in this project; any protocols that Scapy supports are also supported by the analysis framework. As Scapy continues to be developed (at time of writing the latest commit to the Scapy source code repository is 02/03/2015) improved protocol support in Scapy will result in improved protocol support for this framework.

Other Python packet processing libraries are available [26, 70, 71]. The selection of Scapy over other libraries was based on support for as many protocols as possible, ease of use and support/tutorial availability.

5.2.4 Choice of TCP Session Reassembly Tool

A number of different programs capable of reassembling TCP sessions were evaluated for use in this project. These were:

- tcpick [127]
- Bro IDS [99, 106]
- pynids [94]
- libntoh [44]
- tcpflow [45]
- flowgrep [87]
- justniffer [92]
- tcptrace [96]
- tshark [42]

Each tool was installed and then evaluated (using a test pcap file) based on its output format, ease of use and features provided. Notable observations for each tool are now discussed.

The first, tcpick, performs well and produces human-friendly output to a console. The output between sessions can be delimited by a custom separator allowing further processing to extract sessions. Key information is colour coded in the output, but this is mostly irrelevant as the output will not be directly shown to a human. Data can be written to a file or to the standard output stream (stdout).

Bro IDS offers a lot of functionality, one aspect of which is the ability to reconstruct TCP sessions. Output from Bro is written to a number of different files (typically tab delimited) and represents fields or statistics from the sessions (e.g. source IP address or the number of packets). It can also extract information from individual packets such as DNS query names. Overall Bro provides much more information than is necessary for the analysis framework.
During testing, pynids did not cope well with TCP sessions that were not closed properly (e.g. with a TCP FIN or RST flag) and sometimes failed to output data from sessions that were closed. It was quickly ruled out.

A Python wrapper exists for libntoh or it can be called using the native executable. During testing, it did not appear to be able to output the information required (such as the raw traffic from client to server and the number of packets in a session) without custom code being written.

The raw data for requests and responses is written to different files by tcpflow. The IP addresses and port numbers of the TCP session form part of the file names. An XML report detailing the processing conducted is generated after a pcap file has been processed. The number of packets in a session is reported but this is not broken down into request and response packets.

As flowgrep depends upon pynids, and pynids did not handle unclosed TCP sessions well, flowgrep was not evaluated.

The justniffer tool is capable of outputting a number of different fields from reconstructed TCP sessions, including HTTP specific fields such as User-Agent as well as specified custom headers. Regular expressions can be applied to request or response headers to extract more complex examples. The raw request and response bytes can be output to stdout.

When using tcptrace, all requests in a session (e.g. multiple HTTP requests) are grouped together into an output file which is saved to disk. This contains the raw bytes from the network traffic. A similar file is created for the session response. Using the command line option “-b” brief statistics are output to stdout. If the “-I” option is used then detailed statistics are produced. Example output from tcptrace can be seen in Appendix I.

The final tool evaluated was tshark. Using the correct command line arguments it is possible to have tshark output selected fields in a delimited format. A summary of available fields is available at [97]. The syntax to produce this type of output can be seen at [129, p. 11]. However, the reconstructed session cannot be output in raw format.

The key feature required of a TCP session reconstruction tool for this project is the ability to present a reconstructed session to calling code. This allows any further processing to be integrated into the rest of the framework code. The final choice is tcptrace as it meets this criterion, is simple to use and provides a wealth of information which can be subsumed into the analysis framework if necessary.

### 5.2.5 Configuration Options

The following options are configurable for the analysis framework:

- **DEBUG** - toggles debug mode. All work was done with this set to “True” as it provides a built in web server. If set to false an external web server is required to host the Python code.
- **COMPRESS_DEBUG** - allows gzip compression in debug mode if set to “True”
- **BASE_DIRECTORY** - the top level directory used by the analysis framework
- **PCAP_UPLOAD_DIR** - the location of the pcap file store, typically beneath the `BASE_DIRECTORY`
- **PROCESSED_PACKETS_DIR** - the location of the packet feature file store, typically beneath the `BASE_DIRECTORY`
- **PROCESSED_SESSIONS_DIR** - the location of the session feature file store, typically beneath the `BASE_DIRECTORY`
• **TCPTRACE\_OUTPUT\_DIR\_BASE** - the location of the working directory used when reconstructing TCP sessions, typically beneath the **BASE\_DIRECTORY**

• **SQLITE\_DATABASE** - the location of the SQLite database file, typically beneath the **BASE\_DIRECTORY**. Can be set to “:memory:” in which case the database will be created in computer memory.

• **SCAPY\_PACKET\_READER** - the location of the scapy\_packet\_reader script which extracts packet features. The script uses Python 2 (not Python 3) so is called as an external program.

Table 5.3 explains how the configuration can be changed using an environment variable and external file. Alternatively, the default configuration file can be edited directly.

### 5.2.6 Source Code

Code written for the analysis framework uses the following programming languages:

- Python 2
- Python 3
- Bourne shell
- Bash shell

Table 5.3 describes the purpose of each source code file.

<table>
<thead>
<tr>
<th>File</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>/sandnet2/scapy_packet_reader_src/sandnet/af/fe/</td>
<td>Takes a pcap file as input and identifies packet features within it. Outputs the packet features as JSON. Uses Python 2 so is separate from /sandnet2/analysis_framework. Uses Scapy [9] for protocol support.</td>
</tr>
<tr>
<td>packetextractors/scapy_packet_reader.py</td>
<td></td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/runserver.py</td>
<td>Loads the configuration for the analysis framework and runs a web server on localhost, i.e. starts the analysis framework API</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/runserver.sh</td>
<td>Runs runserver.py with the appropriate PYTHONPATH Linux environment variable</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/api/api.py</td>
<td>The analysis framework API</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/api/config.py</td>
<td>The analysis framework configuration options. These settings are overridden if the environment variable “SANDNET_API_CONFIG” points to a python file containing a custom configuration.</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/api/database.py</td>
<td>Manages all interaction with the database and file stores. When pcap uploads are accessed, calls the packet and session feature extraction code, if necessary.</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/api/json_encoder.py</td>
<td>Encodes Python objects to JSON</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/api/upload.py</td>
<td>Simple Python object to represent an upload</td>
</tr>
<tr>
<td>File</td>
<td>Purpose</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/clustering/feature_vector.py</td>
<td>Feature vectors represent the key information for either a packet or session request when clustering. They are used as the output of the clustering algorithms. See Appendix B for an example of the feature vectors grouped into clusters.</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/clustering/session.py</td>
<td>Session (HTTP request) clustering algorithms</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/clustering/transport.py</td>
<td>Packet (transport layer) clustering algorithms</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/fd/applicationlayer/http.py</td>
<td>Simple Python object to represent the HTTP (request and response) protocol features</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/fd/protocol.py</td>
<td>Simple Python object to represent any protocol, an extension of Python's dict object. Used for packet protocols.</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/fd/session.py</td>
<td>Simple Python object to represent a TCP session</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/fe/packets/scapy_packet_extractor.py</td>
<td>Parses the output of scapy_packet_reader.py enabling it to be used in Python 3 source code. Validates packet protocols and checks for corrupt data.</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/fe/sessions/applicationlayer/http_feature_extractor.py</td>
<td>Parses TCP sessions and extracts any HTTP protocols present in order to identify features</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/fe/sessions/tcptrace/sessioniser.py</td>
<td>Calls tcptrace to extract TCP sessions from a pcap, then calls http_feature_extractor.py</td>
</tr>
<tr>
<td>/sandnet2/analysis_framework/src/sandnet/af/web/web_interface.py</td>
<td>Responds to the initial web request to load the web interface</td>
</tr>
</tbody>
</table>

Table 5.3: Analysis framework source code file listing

Refer to Appendix C for a breakdown of other software used.

### 5.3 Web Client

#### 5.3.1 User Interface Code Libraries

There are a huge number of software libraries that can be used to simplify web page development, or to add extra functionality to web pages. The libraries used to implement the web client include:

- **Bootstrap** [10] - used to apply consistent styling to the interface
• **Jasny Bootstrap** [20] - contains useful extensions for Bootstrap such as hiding menus off canvas and table row selection

• **bootstrap-select** [84] - an extension to Bootstrap, used to provide dropdown boxes that are searchable

• **Bootstrap Table** [124] - an extension to Bootstrap, used for all tables within the web client. Provides table pagination and searching.

• **jQuery** [58] - used for its plethora of helpful JavaScript functions

• **Knockout** [114] - used to map data models to the web interface

### 5.3.2 Choice of Visualisation Library

There are a number of different visualisation options that could be used to fulfil project objectives. Dashboards can present summary views from a number of different information sources, all on one page. A couple of dashboard frameworks were considered for use within this project [23, 125]. They were not used due to lack of support for detailed visualisations but the framework could be extended to use them if a dashboard was required.

According to [11], “D3.js is a JavaScript library for manipulating documents based on data”. Research into the capabilities offered by D3 identified the almost limitless variety of different visualisations that can be created. The principle D3 author’s collection of visualisations [12] provides a huge number of examples of what can be achieved using D3 to generate custom visualisations. For this reason, D3 was used as the first visualisation engine within the web client. A screenshot of this early work can be seen in Figure 5.1. Custom code was written to enable fluid transitions between different visualisation states, e.g. when sorting a bar chart from ascending to descending. A video showing this is available in the project repository (see section 5.6).

![Sandnet Visualiser](image)

**Figure 5.1:** Visualisation using D3, present in an early version of the web client. This bar chart shows the count of DNS packets that contain specific ‘answer count’ values.

However, as D3 offers completely customisable visualisations, this makes it complex to use. For ‘standard’ visualisations such as bar charts and pie charts a lot of coding is required. This led to the use of Highcharts [2] as the visualisation library. Highcharts provides a large number of pre-made visualisations, such as line charts, bar charts and pie charts, that can be configured easily. The ability to print the visualisation or export it to an image file is also supported. An example of an area chart generated using Highcharts is shown in Figure 5.2. Interaction with the visualisations is supported. For example, hovering over a data point in a line chart (or a bar in a bar chart, segment of a pie chart etc.) produces a hover over animation that displays the name of the data point and its value.
5.3.3 Source Code

The web client is built using the JavaScript programming language to implement the logic. This consists of one file ("my_code.js") that is capable of interacting with the framework API and makes use of the Highcharts library to generate visualisations. The code also takes care of manipulating the URL so that a URL can be entered directly to access particular aspects of the web client. For example, “http://localhost:5000/#visualiser=1,3” will display the visualisation page for pcap files one and three, whereas “http://localhost:5000/#view-uploads” will display the view uploads page.

HTML and CSS are used to define the layout of the web page. The requisite HTML is loaded only when needed, with the flow of control managed by the JavaScript code and dependent upon user interaction.

The web client has been tested using Firefox version 36 and Chromium version 41. Any web browsers supported by the JavaScript libraries used (see section 5.3.1) and the visualisation library (see section 5.3.2) should be capable of using the web client.

Refer to Appendix C for a breakdown of other software used.

5.4 Implementation Statistics

The majority of the framework is written using Python 3, JavaScript and HTML. The number of lines of code is shown in Table 5.4. This includes all files that are part of the framework. Empty lines and comment lines are included in the totals. Here, the term ‘lines of code’ is also used for non-programming languages such as HTML and CSS. The grand total across all files is 5971 lines of code.
<table>
<thead>
<tr>
<th>Python (2 and 3)</th>
<th>Bash</th>
<th>Bourne</th>
<th>JavaScript</th>
<th>HTML</th>
<th>CSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>3537</td>
<td>136</td>
<td>18</td>
<td>1845</td>
<td>390</td>
<td>45</td>
</tr>
</tbody>
</table>

Table 5.4: Lines of code used to create the framework, split by file type

5.5 System Security

As is discussed in section 5.1, virtual machines are used when running malware samples. From a security point of view, the use of virtual machines greatly reduces the risk of infection of the host system and the virtual hard disk can be restored to a clean snapshot after the sample has been run.

To further complement this, the host system that the framework was installed on to conduct the project, was itself a virtual machine. Thus, even if a malware sample could break out of the environment it was running in, it would still be inside another virtual machine.

To obscure the originating IP address from malware command and control servers, a Virtual Private Network (VPN) service was purchased. The VPN was activated (on the host machine) before any malware samples were run within the capture framework, thus the IP address that communicated with malware command and control servers was the IP address of the VPN provider.

5.6 Availability

Access to the source code for the framework is restricted. It is stored in a private repository [88]. For access please contact the author or supervisor. The repository also contains all the raw data from the analysis (see chapter 7), but not the MALICIA dataset itself, which can be obtained from [55].

5.7 Hardware Specification

A laptop with a 64 bit 4-core 2.8GHz Intel i5 processor and 8GB RAM was used throughout this project. Linux Mint version 17 was installed on the laptop and used as the host operating system. The framework was installed inside a VirtualBox virtual machine (running Linux Mint version 17) allocated 4GB RAM and two processor cores. Hardware virtualisation extensions were enabled.
Chapter 6

Visualisation and the Web Client

This chapter describes the types of visualisations the framework is capable of and provides some screenshots of the web client to illustrate its capabilities. The screenshots in this chapter include the full web page view. Screenshots in other chapters are cropped to just the pertinent information, typically by removing the web page title and borders.

The principle of responsive design was used when designing the web client so that it can be used on different sized screens and information remain well presented and organised. A screen size of 1600x900 was used throughout development and the virtual machine used to conduct the analysis and create screenshots for this report had a screen resolution of 1600x796.

6.1 Histograms

The histograms are all calculated by the analysis framework (web server) API and raw JSON data returned to the web client for visualisation. The histogram may be calculated in four different ways:

1. the absolute number of each feature value present (named ‘count’ in the web client)
2. the percentage of the feature that each feature value represents (named ‘percent’ in the web client)
3. the number of uploads/pcaps each feature value is present in (named ‘number of uploads’ in the web client)
4. the percentage of uploads/pcaps each feature value is present in (named ‘percentage of uploads’ in the web client)

For example, if a histogram is calculated for the ‘packet.ip.dst’ feature then the features values are the destination IP addresses present in each packet of the network capture(s) being analysed. To illustrate, presume three network captures are under analysis:

1. the ‘count’ histogram will return the number of times each destination IP address is present in the captures
2. the ‘percent’ histogram will return the percentage that the ‘count’ of each destination IP address represents of the sum of all ‘counts’ for each destination IP addresses in the three pcap files
3. the ‘number of uploads’ histogram will return the number of pcaps (between one and three in this example) that each destination IP address appears in
4. the ‘percentage of uploads’ histogram will return the percentage of uploads (rounded to three decimal places) that each destination IP address appears in. In this example the percentage for each destination IP address could be 33.333%, 66.667% or 100%.
Each histogram may be sorted in one of four ways. The sorting is performed by the analysis framework (web server) API based on input parameters passed by the (web) client.

1. ‘field ascending’ - sort by the feature value name, ascending
2. ‘field descending’ - sort by the feature value name, descending
3. ‘count ascending’ - sort by the ‘count’ or ‘percent’, ascending
4. ‘count descending’ - sort by the ‘count’ or ‘percent’, descending

### 6.1.1 Histogram Screenshots

#### Table

Figure 6.1 shows the results of a call to the histogram endpoint displayed in a table. The table displays the ‘count’ of each ‘packet.ip.proto’ field value, sorted in descending order by count. Clicking on the table headers alternates sorting the table in ascending/descending order.

![Sandnet Visualiser](image)

Figure 6.1: Histogram results displayed in a table
Bar Chart

Figure 6.2 displays a bar chart generated on the session request data lengths present in one pcap. The ‘percent’ that each request data length represents out of all requests is displayed in descending order of data length. Thus the largest session requests appear at the top of the bar chart.

Sandnet Visualiser

Figure 6.2: Histogram results displayed as a bar chart
Column Chart

Figure 6.3 displays a column chart for the HTTP request headers present in four network captures. The ‘number of uploads’ that each header name appears in is displayed. Header names are sorted in ascending order and the text is rotated by -45 degrees to enable the names to be easily read. Hovering over one of the columns results in a hover effect which displays the column name (Content-Type) and the value (2).

Figure 6.3: Histogram results displayed as a column chart
Figure 6.4 displays a line chart showing the distribution of UDP payload data lengths. The ‘count’ of each data length value is plotted in ascending order of data length. This enables immediate identification of the most common data length (50).

Figure 6.4: Histogram results displayed as a line chart
Area Chart

Figure 6.3 displays an area chart for the HTTP URI parameter names present in seven network captures. The ‘percentage of uploads’ that each parameter name appears in is displayed. Parameter names are sorted in ascending order and as with a line chart, the peaks of the chart allow easy identification of the most common values.

Figure 6.5: Histogram results displayed as an area chart
Figure 6.6 displays a pie chart of the distribution of TTL values for DNS resource A records in three pcap files. The ‘count’ of each TTL determines the size of each segment of the pie. Hovering over a segment results in a hover effect displaying the TTL value, the count and the percentage of the total it represents.

6.2 Clustering

Visualisations for clustering results are a subset of those available for histograms. As the features used to cluster a packet or session represent a many-dimensional object, they are not trivial to visualise. For instance, the session (HTTP request) clustering implemented uses four features of an HTTP request to define the distance between one request and another so cannot simply be represented using a scatter plot.

Two types of visualisation are available: a table to provide the raw details and a pie chart to more easily visualise the clusters and allow ‘click-through’ to the data contained in a cluster. Due to time constraints, further visualisation methods are not implemented, but could form the basis of future work.
6.2.1 Clustering Screenshots

Packet Clustering - Table

Figure 6.7 displays a table with the results of clustering packets using the transport protocol clustering algorithm. Clicking on the table headers alternates sorting the table in ascending/descending order. Clicking on the table shades alternate cluster rows to aid visibility, as can be seen in the screenshot. The bottom of the table has been omitted from the screenshot for brevity.

Figure 6.7: Packet (transport) clustering results displayed in a table
Packet Clustering - Pie Chart

Figure 6.8 displays a pie chart with the results of clustering packets using the transport protocol clustering algorithm. Hovering over a segment results in a hover effect displaying the cluster number, number of elements in the cluster and the percentage of the total it represents. Clicking on a segment retrieves the raw JSON data describing the features of packets in the cluster, an example of which can be seen in Appendix G.
Session Clustering - Table

Figure 6.9 displays a table with the results of clustering sessions using the HTTP request clustering algorithm. The clustering type is set to ‘distance threshold’ and the threshold parameter is set to three. Clicking on the table headers alternates sorting the table in ascending/descending order. Clicking on the table shades alternate cluster rows to aid visibility, as can be seen in the screenshot.

Figure 6.9: Session (HTTP request) clustering results displayed in a table
Session Clustering - Pie Chart

Figure 6.10 displays a pie chart with the results of clustering sessions using the HTTP request clustering algorithm. The clustering type is set to ‘distance threshold’ and the threshold parameter is set to three. Hovering over a segment results in a hover effect displaying the cluster number, number of elements in the cluster and the percentage of the total it represents. Clicking on a segment retrieves the raw JSON data describing the features of sessions in the cluster, an example of which can be seen in Appendix J.

6.3 Other Web Client Functionality

Further screenshots illustrating the web client are available in Appendix T. These show the layout of the application and functionality which is not described in more detail in the body of this report.
Chapter 7

Analysis

This section of the report summarises the analysis conducted using the capture framework and the analysis framework.

To start with, the analysis framework is used with one dataset. Select pcap data is first analysed individually to develop an understanding of specific malware network behaviour. Then, the individually analysed pcap data is grouped together and the same analysis performed on the aggregated, cross-family, data. Next, three different malware families are analysed - all pcaps within the family are analysed collectively to identify any trends within that family. Section 7.4 is then wrapped up with a study of the entire dataset. As described in section 2.2 and section 4.2, analysis and clustering across different types of data enables multiple benefits to be realised: both in depth understanding of a specific threat and identification of trends within or between malware families.

To test the entire end-to-end framework, two malware samples are executed in the capture framework and pcap collected and uploaded to the analysis framework. These are then studied in the same manner as previous individually analysed pcaps. Finally, the HTTP request clustering algorithm is tested against the two aggregated traffic captures.

7.1 Available Datasets

7.1.1 MALICIA

The main dataset used for this analysis was the MALICIA dataset, which is described in [85, 86]. It consists of a collection of malware binaries obtained by visiting exploit servers URLs. The malware has been executed in a sandboxed environment and the network traffic recorded. It is this network traffic that forms the dataset used in this analysis. Each sample is labelled with the malware family where known.

According to [85], malware is executed inside a virtual machine running Windows XP Service Pack 3. Most network traffic is not allowed to escape the sandbox, except for some DNS and HTTP to specific sites.

I spent time looking in detail at the pcap files, using tools such as Wireshark [43] as well as the analysis framework. From this preliminary examination I observed that some of the network traffic was not generated by the malware executables and was instead generated by the capturing system used to run the malware samples. Examples of this include the string “User-Agent: JeppoWeb” which occurs in 14526 of the MALICIA packet captures and the string “GET /?start=1&sha1=” which appears in 12950 of the MALICIA packet captures.

I did not remove these artefacts from the packet captures. Even if I had I could not have been sure that I identified all traces of the capturing system. Thus, artefacts from the capturing system used to generate the MALICIA dataset may feature in my analysis. However, the features that I am confident belong to the MALICIA capture system, are disregarded during analysis where possible.
7.1.2 Other Datasets

A number of other sources were considered for use in this project. Publicly available pcaps are reasonably common but finding pcap files of malicious traffic is more difficult. Several repositories of malicious pcap files were identified [24, 27, 32, 69, 90, 98]. Whilst not used directly in this report, a number of the pcap files were processed to test the framework against a wide variety of malware generated traffic.

VirusTotal [132] was used to collect samples to be run in the capture framework. These were used for the analysis conducted in section 7.5.

7.2 Security Concerns

When executing a malware sample there are certain security measures that need to be put in place to avoid compromise of the system. Firewall rules were used to block traffic between the machine running the malware and the rest of the network. The system as designed in section 3.2 is itself run inside a virtual machine on the laptop used for this project. Thus, the malware sample is executed inside a virtual machine running the Windows XP operating system, running inside a virtual machine using the Linux Mint operating system, running on a physical laptop.

A Virtual Private Network (VPN) service was purchased for use when running malware samples. This was used to obscure the true IP address of my internet connection when communicating with malware infrastructure. A large pool of IP addresses was available when using the VPN service, which helped to ensure that a suspicious malware author could not block my IP address if they became aware their malware was being analysed in an automated environment.

Existing research [63] has focused on using containment policies to contain malware samples that are run dynamically. In [112] certain services are redirected to local sink holes and bandwidth limited to reduce the chance of Denial of Service (DoS) attacks to others. The approach taken in this project with the capture framework is to limit the upload bandwidth available to the capture system, whilst allowing all network activity so that it can be recorded and the malware capabilities and actions understood. This ensures that responses from any malware infrastructure will be received and available for study alongside outbound traffic. However, there is no reason why some of the techniques implemented in [63, 112] couldn’t be applied to the capture framework to limit the type of outbound traffic allowed to leave the sandbox.

7.3 Methodology

The approach used for the analysis is detailed below. The modular design of the system allows optional use of the capture framework, the deciding factor is whether the input is a malware sample (such as an executable) or a pcap network capture file. The input for all analysis in section 7.4 is a packet capture file so use of the capture framework is not required. Analysis using the capture framework is covered in section 7.5.

1. upload the file to the analysis framework
2. use the analysis framework API to collect the raw data for the histograms of the network features identified in Table 7.1
3. use the web client to analyse the histograms of the network features identified in Table 7.1 and record any ‘interesting’ features manually
4. cluster the packet and session data using the available clustering algorithms, as specified in Table 7.2, and identify any ‘interesting’ results.

If multiple pcap files are being analysed together (sections 7.4.2, 7.4.3 and 7.4.4) then items 2, 3 and 4 are conducted on the data from multiple pcaps. The analysis framework supports this by allowing multiple upload/pcap IDs to be specified. Certain types of analysis can only take place (or at least only make sense) when analysing multiple pcaps, e.g. querying the percentage of pcaps that have a particular feature - this would always be 100% if only one pcap is selected.

The most time consuming step (for the system) is usually either the feature extraction step (see section 3.3.1) or the session clustering (see section 4.1.2). The feature extraction typically occurs when an upload is first accessed by the analysis framework (e.g. viewing a histogram). To avoid having to wait for this, the force_process endpoint (see Table 3.2) was called for every pcap uploaded and the processing left running overnight. This took approximately six hours.

The individual inputs that were analysed are listed in Table 7.3. Once analysis on each individual input was complete, various combinations of input were analysed as a collection of network traffic. This included:

- one collection containing all individually analysed pcaps
- for the MALICIA dataset, a collection for each malware family (for three families), as identified by the family and/or traffic label fields provided as part of the MALICIA dataset
- the entire MALICIA dataset

The methodology used for analysis of the collections was exactly the same as has been already been described. The only exception is for the entire MALICIA dataset where saving a histogram for every feature and performing the clustering would have been extremely time consuming. Instead, the histograms for features of interest were generated using the web client and no clustering was performed.

*Note: throughout this analysis the term ‘public IP address’ is used to refer to a non-private IP version 4 address. The term ‘private IP address’ is used to refer to the private IP address space consisting of 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16.*

<table>
<thead>
<tr>
<th>Feature Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>packet.protocols</td>
<td>Protocols contained within the packet, e.g. IP, TCP or DNS</td>
</tr>
<tr>
<td>packet.ip.proto</td>
<td>IP next protocol number</td>
</tr>
<tr>
<td>packet.ip.dst</td>
<td>Destination IP address</td>
</tr>
<tr>
<td>packet.ip.flags</td>
<td>IP flags</td>
</tr>
<tr>
<td>packet.ip.frag</td>
<td>IP fragment offset, 0 if not fragmented</td>
</tr>
<tr>
<td>packet.ip.ttl</td>
<td>Time-To-Live of the IP packet</td>
</tr>
<tr>
<td>packet.udp.sport</td>
<td>Source port of a UDP packet</td>
</tr>
<tr>
<td>packet.udp.dport</td>
<td>Destination port of a UDP packet</td>
</tr>
<tr>
<td>packet.udp.data_len</td>
<td>UDP payload data length</td>
</tr>
<tr>
<td>packet.tcp.sport</td>
<td>Source port of a TCP packet</td>
</tr>
<tr>
<td>packet.tcp.dport</td>
<td>Destination port of a TCP packet</td>
</tr>
<tr>
<td>packet.tcp.data_len</td>
<td>TCP payload data length</td>
</tr>
<tr>
<td>packet.tcp.reserved</td>
<td>Reserved bits within the TCP header that should not be set</td>
</tr>
<tr>
<td>Feature Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>packet.tcp.flags</td>
<td>Decimal value of the TCP flags, such as SYN, ACK or FIN and combinations of flags (e.g. a SYN/ACK combination is 18)</td>
</tr>
<tr>
<td>packet.tcp.urgptrer</td>
<td>TCP Urgent Point, or 0 if URG flag not set</td>
</tr>
<tr>
<td>packet.udp.dns_server</td>
<td>IP address of any DNS servers, identified by UDP communications to port 53</td>
</tr>
<tr>
<td>packet.dns.rd</td>
<td>DNS recursion desired flag (0 not desired, 1 desired)</td>
</tr>
<tr>
<td>packet.dns.an-r_a_record_ttl</td>
<td>DNS TTLs for all type A resource records</td>
</tr>
<tr>
<td>packet.dns.an-rttl</td>
<td>DNS TTLs for all resource records, including A records types</td>
</tr>
<tr>
<td>packet.dns.an-rtype</td>
<td>Decimal value for DNS resource record types, e.g. 1 (A), 12 (PTR) or 5 (CNAME)</td>
</tr>
<tr>
<td>packet.dns.an-rclass</td>
<td>Decimal value for the DNS resource record class (almost always set to 1 for IN)</td>
</tr>
<tr>
<td>packet.dns.an-rllen</td>
<td>Length of the DNS resource record data field</td>
</tr>
<tr>
<td>packet.dns.an-rrname</td>
<td>Name of the DNS resource record, e.g. for an A record type this is the domain name</td>
</tr>
<tr>
<td>packet.dns.an-rdata</td>
<td>Data in the DNS resource record, e.g. for an A record type this is the IP address of the domain</td>
</tr>
<tr>
<td>packet.dns.qd-qtype</td>
<td>Decimal value for the type of the DNS query, e.g. 1 (A) or 12 (PTR)</td>
</tr>
<tr>
<td>packet.dns.qd-qclass</td>
<td>Decimal value of the class of DNS query, (almost always set to 1 for IN)</td>
</tr>
<tr>
<td>packet.dns.qd-qname</td>
<td>Name of the DNS query, e.g. for a query type of A this is the domain name</td>
</tr>
<tr>
<td>session.protocols</td>
<td>Protocols contained within a session, e.g. “Http Request” and “Http Response”</td>
</tr>
<tr>
<td>session.fin_closed</td>
<td>Whether the session was terminated with a TCP FIN flag, set to true or false</td>
</tr>
<tr>
<td>session.session_num</td>
<td>The session number within a pcap. For this report, only the largest value is extracted, and used to identify how many sessions are contained within the pcap.</td>
</tr>
<tr>
<td>session.dst_ip</td>
<td>Destination IP address of the session</td>
</tr>
<tr>
<td>session.src_port</td>
<td>(TCP) source port of the session</td>
</tr>
<tr>
<td>session.dst_port</td>
<td>(TCP) destination port of the session</td>
</tr>
<tr>
<td>session.request_num_packets</td>
<td>Number of packets comprising the session request</td>
</tr>
<tr>
<td>session.request_data_len</td>
<td>Total data length of the session request</td>
</tr>
<tr>
<td>session.response_num_packets</td>
<td>Number of packets comprising the session response</td>
</tr>
<tr>
<td>session.response_data_len</td>
<td>Total data length of the session response</td>
</tr>
<tr>
<td>session.request.http_request.headers</td>
<td>Header names in an HTTP request</td>
</tr>
<tr>
<td>session.request.http_request.Host</td>
<td>HTTP request Host header value</td>
</tr>
<tr>
<td>session.request.http_requestREFERER</td>
<td>HTTP request Referer header value</td>
</tr>
<tr>
<td>session.request.http_request.User-Agent</td>
<td>HTTP request User-Agent header value</td>
</tr>
<tr>
<td>session.request.http_request.Accept-Language</td>
<td>HTTP request Accept-Language header value</td>
</tr>
</tbody>
</table>
### Feature Name Description

<table>
<thead>
<tr>
<th>Feature Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>session.request.http.request.Accept-Charset</td>
<td>HTTP request Accept-Charset header value</td>
</tr>
<tr>
<td>session.request.http.request.version</td>
<td>HTTP request version, e.g. 1.0 or 1.1</td>
</tr>
<tr>
<td>session.request.http.request.method</td>
<td>HTTP request method, e.g. GET, POST or PUT</td>
</tr>
<tr>
<td>session.request.http.request.uri.param_names</td>
<td>HTTP request URI parameter names</td>
</tr>
<tr>
<td>session.request.http.request.uri.param_values</td>
<td>HTTP request URI parameter values</td>
</tr>
<tr>
<td>session.request.http.request.uri.path</td>
<td>HTTP request URI path</td>
</tr>
<tr>
<td>session.request.http.request.body.len</td>
<td>HTTP request body length (typically only POST requests have bodies)</td>
</tr>
<tr>
<td>session.response.http.response.headers</td>
<td>Header names in an HTTP response</td>
</tr>
<tr>
<td>session.response.http.response.Content-Type</td>
<td>HTTP response Content-Type header value</td>
</tr>
<tr>
<td>session.response.http.response.Server</td>
<td>HTTP response Server header value</td>
</tr>
<tr>
<td>session.response.http.response.Cache-Control</td>
<td>HTTP response Cache-Control header value</td>
</tr>
<tr>
<td>session.response.http.response.Pragma</td>
<td>HTTP response Pragma header value, typically used to indicate a no-caching policy in HTTP/1.0 clients</td>
</tr>
<tr>
<td>session.response.http.response.Transfer-Encoding</td>
<td>HTTP response Transfer-Encoding header value</td>
</tr>
<tr>
<td>session.response.http.response.Content-Encoding</td>
<td>HTTP response Content-Encoding header value</td>
</tr>
<tr>
<td>session.response.http.response.version</td>
<td>HTTP response version, e.g. 1.0 or 1.1</td>
</tr>
<tr>
<td>session.response.http.response.status_code</td>
<td>HTTP response status code, e.g. 200 or 404</td>
</tr>
<tr>
<td>session.response.http.response.reason_phrase</td>
<td>HTTP response reason phrase, e.g. OK</td>
</tr>
<tr>
<td>session.response.http.response.body.len</td>
<td>HTTP response body length</td>
</tr>
</tbody>
</table>

#### Table 7.1: Histogram features analysed

<table>
<thead>
<tr>
<th>Input</th>
<th>Type</th>
<th>k Parameter Values</th>
<th>threshold Parameter Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>packets - transport layer</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>sessions - HTTP request</td>
<td>k clusters</td>
<td>2, 3, 4, 5, 7 and 10</td>
<td>N/A</td>
</tr>
<tr>
<td>sessions - HTTP request</td>
<td>distance threshold</td>
<td>N/A</td>
<td>0.1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 and 12</td>
</tr>
</tbody>
</table>

Table 7.2: Clustering algorithms analysis parameters. 0.1 is used as a threshold value to group identical HTTP requests.

### 7.4 MALICIA Dataset Analysis

The MALICIA dataset I obtained contained 17017 pcap files. Of these, 50 were empty files. The remaining 16967 files were uploaded to the analysis framework and the packets and sessions were
processed without error. However, a number of the pcaps appear to stop mid-flow. This could be due to the way they were captured, e.g. using a time limit, with network traffic being transmitted at the moment of shutdown. This often exhibited itself as packets not containing the required information (e.g. an IP source address) or sessions not being complete (e.g. an HTTP request specifying a content length larger than the length of the request). When this was detected the corresponding packet or session was ignored from all further processing and doesn’t feature in any analysis.

In total, there are 11,363,268 packets in the MALICIA dataset that were processed by the analysis framework, and 159,464 TCP sessions identified and processed.

Of the 16967 files uploaded to the analysis framework, five were selected for individual analysis. These are listed in Table 7.3 and cover four different malware families. The Zeus/Zbot family appears twice as two variants of Zeus are analysed. The specific pcap files selected for analysis cover different attributes of the MALICIA dataset: one is from a packed executable, four have icons associated with the executables, one sample is ‘milked’ (retrieved) by the MALICIA framework 387 times and so on. The intention is to cover reasonably well known malware.

<table>
<thead>
<tr>
<th>Input</th>
<th>Family</th>
<th>Traffic Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>6e4b6e32a5bb6006cedd06518246ec2ef551a_001146.pcap</td>
<td>cleaman</td>
<td>cleaman</td>
</tr>
<tr>
<td>c911088c90f85d9c1331e616034e28c6ba5d2eb52_019694.pcap</td>
<td>cridex</td>
<td>cridex</td>
</tr>
<tr>
<td>78c5749e38af062c2a05ec78ba672ce46de64026_000966.pcap</td>
<td>cutwail</td>
<td>cutwail</td>
</tr>
<tr>
<td>ae7e22714bab3babcaada0aa7fbc79b5249b2e9_021033.pcap</td>
<td>zbot</td>
<td>zbot</td>
</tr>
<tr>
<td>a6a737944a0a59d3e8f143ebced1d608782998_019263.pcap</td>
<td>zbot-gameover</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3: MALICIA pcap inputs to be analysed individually, along with their family and traffic label as identified in the MALICIA dataset

### 7.4.1 Individual Sample Analysis

This section presents an analysis of five network traffic captures, each from a different malware family. Generation of the histogram data for all five selected pcaps using a Python 3 script to call the analysis framework API took 2.1 seconds, averaged over five attempts. This included writing one JSON file to disk per histogram as well as creating one CSV file per pcap containing the histogram data.

Generation of the clustering data for all five selected pcaps using a Python 3 script to call the analysis framework API took 9.5 seconds, averaged over five attempts. This included writing one JSON file to disk per pcap for the transport clustering, writing six JSON files per pcap for the k clusters session clustering as well as creating 13 JSON files per pcap for the distance threshold session clustering.

The data gathered from the API is available in the project repository (see section 5.6). Analysis was then conducted using the web client to visualise the raw data, histograms and clustering results.
Cleaman - 6ec4ba6e32a5bb6006cdd06c518246cece2ef551a_001146.pcap

This pcap contains traffic from the Cleaman malware family which is typically used to “redirect Bing, Google, and Yahoo search results to bogus webpages that serve advertisements, adware programs, and malware” [79].

Over 45% of the packets are for the Simple Network Management Protocol (SNMP). This includes SNMP inside of Internet Control Message Protocol (ICMP) data. ICMP features heavily in this pcap. ICMP and UDP (probably for the SNMP traffic) account for over 80% of the IP traffic. Over 60% of the UDP data is to port 161, used for SNMP. A number of public IP addresses are contacted.

Only a local DNS server is used (10.3.1.3). No notable HTTP traffic was observed and as such the HTTP request clustering only contains data from the MALICIA capture framework.

The transport clustering confirms the observations mentioned above. Two clusters clearly stand out and can be seen in Figure 7.1. The separated segment (Cluster 3) is the cluster containing all the SNMP traffic. Upon inspection the SNMP data is inbound, i.e. the malware appears to be receiving these connections from public IP addresses.

Cridex - c911088e90f85d9c1331e616034e28cba5d2cb52_019694.pcap

Cridex is a worm that focuses on stealing banking details (e.g. usernames and passwords) as well as logon details for social networking web sites. It can also be used to spread other malware [77, 80].
Nearly all the traffic within this pcap is TCP (99.4%). Ignoring the virtual machine IP address (192.168.0.2), the destination IP addresses are fairly evenly distributed across different public IP addresses - the malware communicates with a variety of different command and control servers, as shown in Figure 7.2.

![Packet destination IP addresses](image)

**Figure 7.2:** c911088c90f85d9c1331e616034e28cba5d2cb52_019694.pcap packet destination IP addresses

Nearly all of the HTTP requests are to public IP addresses. All of these are direct to IP addresses, no domain is used and no DNS lookup is performed. The port number is always specified as “8080”. This can be seen in Figure 7.3, and matches the destination IP addresses previously described. There is a DNS request to look up the IP address of “www.google.com”. This appears to be a connectivity check by the malware as there is one HTTP request with “www.google.com” in the Host header.
Interestingly there are two HTTP User-Agents that can be observed in this traffic. One of them only occurs once (upon manual inspection this is associated with the request to “www.google.com”) and so might be missed if manually trawling through the raw data. However, because all User-Agents can be listed it is possible to identify that there is a second one used by the malware. This is shown in Figure 7.4 (note, the “JeppoWeb” and “Python-urllib/2.6” User-Agents are generated by the MALICIA capture system). The URI “/zb/v01/b/in/” is seen for nearly all of the HTTP requests. Most HTTP requests are POSTs and have a Content-Length (POST body length) of 225.
Figure 7.4: c911088e90f85d9c1331e616034e28cba5d2cb52 019694.pcap session HTTP request User-Agent field values

As there are so many similar HTTP requests the session clustering produces very few clusters. Figure 7.5 shows that 359 of the 362 HTTP requests are identical with respect to the features measured when clustering (using a distance threshold) and so are grouped together.

If network IDS signatures were desired (see section 2.4 and section 4.2.1) to identify this Cridex sample, then the HTTP features previously discussed would make good candidates to be included in a signature as they never change.
The Cutwail botnet, also known as Pushdo, is typically used to send spam emails. It is capable of stealing FTP passwords, can spread other malware and hides itself using a rootkit [21, 81]. In January 2009 it was the second largest botnet as measured by the average spam per day [56].

This network capture has more TCP than UDP packets (278 vs 200) as well as some ICMP. There is no SMTP, which is not expected for a spamming botnet. A number of different public IP addresses are contacted. Multiple domains are looked up using DNS queries. Each is queried twice except for one (“gweb.4octets.co.uk”) for which there are eight DNS queries.

12 HTTP requests and nine HTTP responses are identified in the pcap. Only five different HTTP request headers are observed (“Accept-Language”, “Connection”, “Content-Type”, “Host” and “User-Agent”). All User-Agent values are set to “Shareaza” (a peer-to-peer (P2P) file sharing tool) which indicates the malware is pretending to be another piece of software or is using a software library which sets that User-Agent header. The Host header fields are a subset of the domain names queried using DNS. It is not clear exactly why all domain names queried are not used in the HTTP requests. It is possible that the malware queries a number of domain names, some of which are to be used as backup in case the primary domains cannot be reached, or the domains could be used for a protocol other than HTTP.
HTTP URI parameter names of “client”, “get” and “net” are used in eight HTTP requests (for example “/g2/bazooka.php?net=gnutella2&get=1&client=RAZA2.5.0.0”). Some of the URI values (such as “gnutella2”) fit the P2P file sharing theme, which suggests the malware may use P2P for communications. This led me to revisit the TCP destination port numbers. Six packets have a destination port of 6346 which is commonly used by file sharing software using the Gnutella P2P network. I confirmed this by checking the packet features with a call to the analysis framework API endpoint /api/uploads/7869/features/packets/ - all the packets involved the public IP address “86.76.0.12”.

The different types of HTTP requests cluster well when using a distance threshold value of three. This results in seven clusters and can be seen in Figure 7.6. Clusters five, six and seven are the traffic generated by the malware, clusters 1-4 are traffic from the MALICIA capture framework. Those clusters containing the malware traffic have enough structure in the HTTP requests that creating network IDS signatures for them is possible.

![Session HTTP Request Clusters](image)

Figure 7.6: 78c5749e38af062c2a05ec78ba672ce46de64026.000966.pcap HTTP request clusters using a threshold of 3

By checking a number of different features in the analysis framework I have shown it is possible to discern the different protocols used by the malware i.e. HTTP and P2P. Much of the command and control infrastructure can also be enumerated, e.g. by viewing a histogram of the DNS queries (although this won’t identify requests directly to IP addresses). The HTTP request clustering also proved capable of accurately splitting the different types of traffic.
Zeus (zbot) - ae7e22714babe3babcaada0aa7fbc79b5249b2e9_021033.pcap

Zeus is a crimeware toolkit, specifically a “set of programs which have been designed to setup a botnet over a high-scaled networked infrastructure” [8]. It is typically used to steal personal information and financial details and may be used to download other malware [82]. There are different types of Zeus malware in the MALICIA dataset, this sample is labelled as “zbot”.

As can be seen in Figure 7.7, the majority of the packets are TCP in IP in Ethernet. UDP appears to be used for by the malware to make DNS queries - six DNS A record queries were made, two each for the domains “bebexarila.com.pt”, “brilhanteservice.com.br” and “risparmioassicurativo.net”. These all used the locally configured DNS resolver (192.168.0.1).

![Figure 7.7: ae7e22714babe3babcaada0aa7fbc79b5249b2e9_021033.pcap packet protocols. Note, as multiple protocols can be identified in one packet the sum of the counts adds up to more than 654 (the number of the packets)](image)

Studying the TCP sessions, the majority (almost 80%) have a size of 820 bytes (request data length and response data length). These are composed of requests of 458 bytes and responses of 362 bytes. This suggests that the same request and response is used in multiple sessions. The content of the communication has not had to be examined to hypothesise this. There are 49 HTTP requests and 48 HTTP responses in the 54 sessions. Five sessions may contain protocols that the analysis framework does not have a session protocol parser for.
In Figure 7.8 it is clear that two public IP addresses account for the majority (over 80%) of the outbound session communications. The visualisation helps to show this by sorting the data into ascending order, producing a steep slope between the third and fourth IP addresses.

![Histogram of session destination IP addresses](image.png)

Listing the HTTP response versions there are two HTTP responses with a version number of “1.0”. HTTP 1.0 [7] is an older version of the HTTP specification and I did not expect to see much of it. Investigating in more detail revealed that the HTTP responses sent by the MALICIA capture framework use HTTP 1.0 and so this could be a useful way to identify that traffic (in the raw pcap). In contrast, when looking at the session.request.http_request.version feature, the HTTP requests made by the MALICIA capture framework appear to be version 1.1 and the Zeus traffic appears to use version 1.0.

Of the HTTP request headers, the User-Agent one proves useful for identifying the malware traffic. A User-Agent of “Mozilla/4.0 (compatible; MSIE 5.0; Windows 98)” is seen in 47 HTTP requests. The MALICIA capture framework uses Windows XP so this User-Agent claiming to be Windows 98 is obviously spoofed and is a good candidate to include in a network IDS signature. Also of note is the Content-Encoding HTTP request header which only ever appears with a value of “binary” (in 44 requests) and does not appear to be a standard value as defined in relevant RFCs [29, 31]. In Figure 7.9 the HTTP request Host header fields are shown and match the domains in the DNS queries. The domains are hosted on IP address “203.74.181.4” (as determined by looking...
at the feature packet.dns.an-rdata). This means that the top two IP addresses listed in Figure 7.9
(“91.121.84.204” and “200.72.183.54”) are placed directly in the Host header and DNS requests
are not used by the malware to determine all of its infrastructure, i.e. the IP addresses are likely
hardcoded into the malware.

There are three different URIs to download executable files (for example “/rQuTTn7k/mkE.exe”).
It is possible that the malware is attempting to fetch other modules for itself or other malware to
execute. There are 44 HTTP requests with a URI of “/pony/gate.php”, thus it is reasonable to
postulate that these requests serve a different purpose to a request to download an executable file.
Examination of the HTTP request clustering data reveals that these two types of HTTP requests
do form separate clusters, for instance performing distance threshold clustering with a threshold of
7 or clustering into k clusters where k is set to 4. This is shown in Figure 7.10. Cluster 3 contains
the requests to “/pony/gate.php”. Cluster 4 contains all three requests for executable files, for
reference the JSON data returned when clicking on the cluster (to view more information in the
web client) is provided in Appendix J.

Figure 7.9: ae7e22714babe3babcaada0aa7fbc79b5249b2e9_021033.pcap session HTTP request Host
field values
Zeus (zbot-gameover) - a6a7a73f7944a0a593e8f143cbbce1d608782998_019263.pcap

Gameover Zeus is an evolution of earlier versions of Zeus. Instead of employing centralised command and control servers, Gameover Zeus uses P2P communications, whilst using a domain generation algorithm as a fallback to a centralised communications channel. A thorough dissection of Gameover Zeus can be found in [1]. There are different types of Zeus malware in the MALICIA dataset, this sample is labelled as “zbot-gameover”.

Given the information available about Gameover Zeus it is reasonable to assume the following two points will be confirmed during analysis of the traffic:

- P2P traffic will be present
- random domain names will be observed

Thus, when approaching the analysis of this traffic capture I was particularly interested to see if I could find these types of traffic and confirm the Gameover Zeus behaviour.
According to reporting [1, 103] the UDP protocol is used for Gameover Zeus P2P communications. Figure 7.11 shows the distribution of UDP port numbers used for the P2P communications. As such, UDP port numbers 10000-30000 are a sensible feature to look for within this pcap.

![Figure 7.11: Gameover Zeus UDP port numbers histogram (sample size 100,000) [103]](image)

Figure 7.11 shows the distribution of UDP destination ports in the sample. The shaded area highlights the 20 different UDP destination ports within the 10000-30000 range, each of which is used only once. Interestingly, there are exactly 20 packets that use a UDP source port of 11403, indicating that a fixed source port is used when trying to communicate out to a P2P node. Whilst it is not possible to be 100% sure that the UDP packets are the Gameover Zeus P2P protocol, it does match known behaviour. A possible new method of identification has been revealed - checking for multiple UDP connections with the same source port and a destination port in the 10000-30000 range.

Using other features such as the size of the UDP packet (packet.udp.data.len) are unlikely to help identification of the protocol - “the junkSize field is generated randomly and that number of random bytes is added at the end of each packet. It is probably a feature intended to cripple the ability to automatically detect the suspicious traffic using network signatures” [103]. The next step is to check for the existence of the domain name generation algorithm.

![Figure 7.12: a6a7a73f7944a0a593ce8f143ebce1d608782998_019263.pcap UDP destination ports. The shaded area highlights port numbers between 10000-30000.](image)
According to a histogram of the packet.dns.qd-qname feature there are 33 domain names queried, 30 of which appear to be randomly generated (e.g. “mwpexc59hznsmnynygysf22lw118xes031.info” and “j56bza27a57l1f52p12psh14ism39j9c4958l.x.ru”). The top level domains present are “biz”, “net”, “ru”, “com”, “org” and “info”, exactly as described in previous studies of the domain generation algorithm - “The DGA uses top-level domains taken from the set {biz, com, info, net, org, ru}” [1]. What does appear different, however, is the length of the domain names - “The generation of a domain name starts by taking the MD5 hash over the concatenation of (transformations of) the year, month, day, and domain index. The MD5 hash is then used to generate a domain name of at most 32 lower case alphabetic characters. Finally, the domain is completed by selecting one of the six top-level domains and concatenating it to the domain name” [1]. The domains present in this pcap are more than 32 characters long. Ignoring the top level domain, some are 36 characters long, others 37 or even 40 characters in length. This may represent an evolution in the algorithm used to generate the domain or perhaps the malware sample used to generate this pcap was a different version of Gameover Zeus to the aforementioned reporting.

Of the 30 domains queried using DNS, 29 are used in the Host field for HTTP requests. The domain “g33e31k27kzhovgyv66hbxktweshzh24eqa67.org” is not used. Initially I lacked an explanation for this. After viewing the complete list of packet features (using the analysis framework API endpoint /api/uploads/10688/features/packets/) the answer was clear; the packet containing the DNS response was the fifth to last in the pcap, hence the traffic capturing was stopped before the malware could make the HTTP request to the domain.

Running the transport clustering algorithm resulted in 159 clusters. Three of these contained more than 50% of the packets (60.3%). Two of the three clusters contain the packets used by the MALICIA capture system to upload the executable file to the virtual machine, the third contains the TCP packets to IP address “59.176.200.37” on port 80. This represents all the empty (data length of 0, typically TCP packets with the ACK or ACK/FIN flags set) HTTP request packets as all domains queried resolve to the same IP address. Other, smaller, clusters contain groups of packets with the same data length to (or from) this IP address. Thus, the transport clustering algorithm is capable of grouping this related traffic.

The results of the HTTP request clustering with a threshold value of 0.1 can be seen in Figure 7.13. Cluster 3 contains all the HTTP traffic generated by the malware, as well as two requests to “www.google.com”. These may be two connectivity checks by the malware or may be part of the MALICIA capture framework. They have been grouped with the requests to the randomised domain names because the four HTTP features described in section 4.1.2 are exactly the same for the traffic to Google as well as for the randomised domains, i.e. a request method of “GET”, a URL path of “/” and no parameter names or values. The two requests to Google appear in separate clusters when clustering into k clusters where k is set to 5. This works when the HTTP requests are identical (with respect to the features measured) because the first requests encountered are separated into their own clusters, and the two requests to Google occur before any of the requests to randomised domain names. This is shown in Figure 7.14 where the “Random when multiple longest edges” parameter is set to false to ensure the first requests encountered are separated.
Figure 7.13: a6a7a73f7944a0a593e8143cbbce1d608782998_019263.pcap HTTP request clusters using a threshold of 0.1. Two of the requests in cluster 3 are not to randomised domain names.
7.4.2 Cross-Family Analysis

To validate the accuracy of the clustering, cross-family analysis is performed on the five samples analysed individually. All five traffic captures shown in Table 7.3 are selected and clustering performed across the aggregated packet/session features. Unfortunately the transport clustering is of little benefit as there are no TCP or UDP packets destined for the same public IP address that appear in different families. Had this been the case they would have been clustered together.

As there are five different families it would seem reasonable to perform the HTTP request clustering, splitting into $k$ clusters, with a $k$ value of 5. However, the Cleaman traffic capture contains no HTTP traffic from the malware, so $k$ is set to 4. The hypothesis here is that the traffic for each family will be sufficiently different to traffic from other families and so a cluster will form for each family. This is shown in Figure 7.15.

Cluster 1 contains all MALICIA capture framework traffic. Cluster 2 contains all the HTTP POST requests from the Zeus (zbot) malware traffic. Cluster 3 contains some MALICIA capture framework traffic and HTTP GET requests from Cutwail (the traffic with the User-Agent “Shareaza”), Gameover Zeus (the traffic to the randomised domain names) and Zeus (zbot) (the requests to
download executable files). Cluster 4 contains only Cridex HTTP requests - these are all the HTTP POSTs to URIs such as "/zb/v_01_b/in/".

Clustering into four clusters has given a nice split of the traffic but could be further improved if the Cutwail, Zeus and Gameover Zeus traffic that grouped together can be split into separate clusters. The results of using a k parameter value of 6 can be seen in Figure 7.16. Clusters 1 and 3 are MALICIA capture framework traffic. Cluster 2 contains some of the Cutwail traffic with User-Agent “Shareaza”. Cluster 4 contains the same 44 Zeus POST requests as cluster 2 when k is 4. Cluster 5 contains some MALICIA capture framework traffic, the remains of the Cutwail traffic with User-Agent “Shareaza”, as well as the Gameover Zeus randomised domain traffic and the Zeus (zbot) requests to download executable files. Cluster 6 contains the Cridex HTTP POST requests. It appears that the extra two clusters introduced have separated out some of the MALICIA capture framework traffic and some of the Cutwail traffic. The remaining Cutwail, Gameover Zeus and Zeus (zbot) traffic is more similar than the Cutwail traffic that has been split off. This is due to the strong weighting applied to the URL path in the clustering algorithm.

Figure 7.15: Cross-family HTTP request clusters with a k parameter value of 4
Accurate splitting of the Cutwail, Gameover Zeus and Zeus (zbot) traffic required testing a few different clustering combinations. Using distance threshold clustering with a threshold parameter value of 5 eventually resulted in correct splitting of the traffic. This resulted in 13 clusters.

7.4.3 Malware Family Analysis

Five individual network traffic samples have been analysed, each from a different malware family. They have been analysed individually (section 7.4.1) and as a collection (section 7.4.2). Analysis now switches to family-wide traffic. The malware families analysed are those of three of the individual samples already studied in section 7.4.1.

Cleaman Family

There are 63 pcap files labelled with the Cleaman family. These contain 25967 packets and 336 TCP sessions. The percentage of the pcaps that contain each packet protocol is shown in Figure 7.17.
SNMP traffic was found in the individual Cleaman sample analysed - just under half of the Cleaman family samples contain SNMP traffic which indicates that there may be different versions of the family or that there is a condition under which SNMP traffic is sent/received. As might be expected, protocols such as TCP, DNS and ICMP feature in every sample.

347 unique public IP addresses were contacted by the Cleaman family suggesting a large amount of infrastructure is used for communications to/from the malware.

The Time to Live (TTL) of an IP packet can hint at the operating system that generated it. Microsoft Windows versions typically use a TTL of 128 whereas a Linux system sets a TTL of 64 [89]. However, it is only one of many features that can be examined to passively identify an operating system [74] and as such should not be considered definitive. Figure 7.18 shows the percentage of Cleaman samples that contain each TTL value. There are clearly four TTL values (1, 64, 128 and 255) that occur in the majority of traffic samples. As they are either the lowest (1) or highest values (255), or a value indicating they are on the same subnet as the host (64 and 128), it is likely that these packets are generated by the MALICIA capture framework, or are the packets sent by the malware (captured before the first network hop).

The values below 64 (but not 1) could be packets that are generated by a Linux system (i.e. the authors of the malware run a Linux operating system on command and control servers, or if the malware is attempting to attack another system then those systems are running Linux). Those
between 65 and 127 could have been generated by a computer running a version of Microsoft Windows. Anything above 128 could be from another operating system or may indicate use of a customised TTL.

A technique that malware could use to check if it was running within a sandbox would be to check for the presence of IP packets with a TTL of 1, 64, 128 or 255 as these are unlikely to occur between nodes communicating across the internet.

Figure 7.18: Cleaman family IP packet TTLs and the percentage of pcap samples that contain them

In contrast to Figure 7.18, the visualisation in Figure 7.19 shows that although there are many samples that contain packets with a TTL of 112, for example, they don’t actually account for that much of the traffic (just 1094 from 25967 packets).

Figure 7.19: The distribution of packet IP TTL values across all Cleaman samples

Having all the traffic samples for a particular malware family provides some opportunities for enumeration of domain names the malware might communicate with. There are two simple ways
to extract this information: histogram the DNS queries made and histogram the HTTP Host header field values. The DNS qnames queried are listed in Appendix K. There are only two HTTP Hosts that are not private IP addresses or “www.google.com” - “190.9.35.198” and “190.9.35.199”. Thus, any of these domains or IP addresses could be investigated in more detail (e.g. to check if they are non-malicious domains the malware resolves to conduct an internet connectivity check) if they were to be used in a network IDS signature.

When clustering the HTTP requests using distance threshold clustering with a threshold of 7, four clusters form. Three of these contain the MALICIA capture framework traffic and one the Cleaman HTTP traffic. This is cluster 2 in Figure 7.20. Using the web client, clicking on the cluster calls the appropriate analysis framework API endpoint (/api/uploads/features/session_requests/<upload_ids_session_nums_request_nums>) and returns all the HTTP requests in cluster 2.

Figure 7.20: Cleaman HTTP request clusters with a threshold parameter value of 7. Cluster 2 is entirely Cleaman traffic and has been correctly separated from the MALICIA capture framework traffic.

The HTTP requests in cluster 2 can be sub-clustered into two groups (which would also happen if a lower threshold parameter such as 6.8 was used): those that have a URI of “/post.php?rnd=787751” and a Host header of “190.9.35.198”, and those with a URI of “/view.php?rnd=787716” with a Host header of “190.9.35.199”. Both of these IP addresses do not appear in any DNS traffic.
so are hardcoded into the malware. “190.9.35.198” appears in 6 Cleaman traffic samples and “190.9.35.199” appears in 43. Each is seen only once per sample. As reported in [107, p. 17], the Cleaman HTTP traffic does not have User-Agent or Accept HTTP headers. In fact, all the traffic has only one header, Host, which contains an IP address. The absence of other HTTP headers and the fact that an IP address is used in the Host field is likely to be something that can be used in a network IDS signature to identify Cleaman traffic across different samples.

Cridex Family

There are 101 pcap files labelled with the Cridex family. These contain 104743 packets and 7109 TCP sessions. As might be expected, packet protocols such as TCP, UDP, DNS and ICMP feature in every sample. 6945 HTTP requests are identified with each pcap containing at least one HTTP request. 6862 HTTP responses are identified but only 96 of the 101 pcaps contain an HTTP response.

There are communications with a large number of IP addresses (268). Enumeration of the domain names queried reveals that some are randomly generated (e.g. “mvkrxumvbedbouiyfh.ru”) whilst others are composed of words (e.g. “scanforsecurityholes.ru”, although security is misspelt). The individual Cridex sample previously analysed did not exhibit this behaviour. Thus, by analysing traffic from multiple samples it is possible to identify behaviour that the malware is capable of but may not always use. It is now clear that Cridex has the ability to randomly generate domains. Reporting [77] confirms use of a domain generation algorithm. The DNS qnames queried are listed in Appendix L.

Enumeration of the HTTP hosts contacted is recorded in Appendix L. There are 166 unique public IP addresses or domain names used. As with other traffic samples analysed, these could be used as in network IDS signatures.

Viewing the histogram of User-Agents used by Cridex reveals two that appear to be Java related: “JNLP/6.0 javaws/1.6.0_14 (b08) Java/1.6.0_14” and “jupdate”. Both appear in nine different pcap files. Indeed there is an HTTP header “UA-Java-Version” present in nine traffic captures, all of which have the value “1.6.0_14”. This may indicate that Cridex uses Java exploits to spread or that part of it is written using the Java programming language.

Listing the names of the HTTP headers may be another useful way of finding data that can be used to write network IDS signatures for the traffic. Of note, there is one HTTP header called “Accept-Encoding” that appears in 94 samples, yet in nine samples it appears as “accept-encoding”. Note the case sensitivity. Whilst HTTP headers are not strictly case sensitive, they typically appear in a standard form, such as “Accept-Encoding”. Deviations from this may be the result of a coding error by the malware author.

Transport clustering results in 2418 clusters. These were not examined in detail due to time constraints. Clustering of the 6945 HTTP requests was performed using distance clustering, with a threshold of 5 and 7. 17 clusters form when the threshold parameter is 5, with the Cridex traffic split among them. With the threshold at 7 two clusters form. One only contains MALICIA capture framework traffic (142 requests), the other 6803 requests (which does include 94 requests from the MALICIA capture framework with the User-Agent “JeppoWeb”) are in the second cluster.

Those HTTP requests in the second cluster that are traffic from the Cridex malware have a URI such as “/mx5/B/in/”. All URIs and occurrences are listed in Table 7.4.

If an analyst were to write a network IDS signature to detect the URI seen in the individual Cridex sample analysed (“/zb/v_01_b/in/”), it would be reasonable to use the regular expression:

`/[a-z]{2}/v_[0-9]{2}_[a-z]/[a-z]{2}/`
Table 7.4: URIs and the number of times they occur in Cridex traffic

However, if that regular expression is tested against the URIs in Table 7.4 it only matches three of them, one of which is the original URI analysed. Thus, by analysing multiple samples from a malware family it is possible to discover new network behaviour which in turn can be used to create network IDS signatures that identify the greater variety in behaviour observed. I have also shown how the HTTP request clustering algorithm implemented in the analysis framework accurately groups the Cridex traffic together, allowing focus on the most relevant traffic.

Cutwail Family

There are two pcap files labelled with the Cutwail family. These contain 4813 packets and 365 TCP sessions. As might be expected, packet protocols such as TCP, UDP, DNS and ICMP feature in both samples. 92.4% of the IP packets have a next protocol number of 6, i.e. contain a TCP payload.

24.1% of the TCP packets are destined for port 80 whilst 23% are destined for port 443. Use of port 443 suggests the malware may be configured to use secure Transport Layer Security (SSL or TLS) for its web connections. This behaviour was not observed in the individual Cutwail sample studied, indicating that the two pcap files in this family could be from different versions of the malware, one of which attempts to secure it’s communications from network snooping.

185 HTTP requests are contained within the family traffic samples. Somewhat surprisingly there are 353 HTTP responses. Understanding how it is possible to have more responses than requests requires examination of the session features (by calling endpoints /api/uploads/7869/features/sessions/ and /api/uploads/8494/features/sessions/). When these are studied it becomes clear that the HTTP responses are being received when requests are made to TCP port 443. In every case, a request is made to TCP port 443 containing a 13 byte payload. This traffic accounts for 47.1% of all the session requests. The same HTTP response is received for each request. This suggests that the MALICIA capture environment is set up to respond to TCP port 443 requests with a default HTTP response (as the same response is seen in a variety of other MALICIA pcaps). As such the responses do not tell us much about the malware, but the requests are extremely useful.

Typical TLS traffic starts with a handshake protocol: a Client Hello message followed by a Server Hello message, certificate exchange to authenticate one or more parties, then a secure key exchange. Only then can encrypted data be transmitted between the two parties. The 13 bytes seen in the requests to port 443 are not long enough to be a Client Hello message. Manual inspection reveals the same 13 bytes are used in every transmission - the hex value “04000001050000000007000100”. Research into this value uncovered that [107, p. 17] contains a network signature containing
the exact same byte string. In [107] it was identified via clustering techniques, here it has been identified by investigations into unbalanced session protocols, the odd use of TCP port 443 and repeated use of sessions with a request length of 13.

As with analysis of the other malware families, the domains queried are enumerated and the results are available in Appendix M. HTTP Host names requested are also shown in Appendix M. To cover every possible domain the malware may use, features such as packet.dns.an-rdata should also be checked (e.g. for CNAME domains). In the Cutwail example this reveals domains such as “apache2-heavy.hamer.dreamhost.com” and “apache2-vat.robelli.dreamhost.com” which are not present in any DNS query names or HTTP Host headers.

The visualisation of the transport clusters can be seen in Figure 7.21. 838 clusters form, of note are clusters 249 and 261. These contain the TCP traffic to port 443 previously discussed. The reason they appear in separate clusters is because some of the packets have a 0 (rather than 13) data length - these are the TCP packets with the ACK/FIN flags set (cluster 249) and as such contain no payload data. A useful modification to the transport clustering algorithm would be one that takes this into account when clustering packets, or performing the transport clustering on sessions instead of packets. As there are so many clusters, the visualisation shown in Figure 7.21 is difficult to interpret and would be better suited in another form, perhaps one that uses a combination of colour coding, size and spacing to convey information.

![Figure 7.21: Cutwail family transport protocol clustering. 838 clusters are generated.](image-url)

The session clustering works reasonably well on the Cutwail HTTP requests. The different behaviour is split into three clusters (clusters 3, 4 and 5 in Figure 7.22). These differ primarily in the URI they use (e.g. “/gwc/skulls.php?net=gnutella2&get=1&client=RAZA2.5.0.0” versus...
“/?net=gnutella2&get=1&client=RAZA2.5.0.0”). If network IDS signatures were desired then they could be written separately for each cluster or one signature could be written to cover all three clusters (e.g. using the URI parameters). The approach taken would be dependent upon how specific a signature was desired.

7.4.4 Feature Extraction

To demonstrate the effectiveness of the analysis framework design and the utility of the feature extraction approach, analysis is conducted across the entire MALICIA dataset to give an overview of the networking protocols used by all the malware in the dataset. Specific protocols are examined in more detail, in line with those discussed in the Sandnet paper [112]. A full list of all features extracted is available in Appendix S.

Network Protocols

Of the 16967 pcap files in the MALICIA dataset that contained data, there are 11,363,268 packets and 159,464 TCP sessions identified and processed. The most common packet protocols (excluding Ethernet and IP version 4) are TCP (56.8% of packets), UDP (20.2% of packets) and ICMP (20.2%...
of packets). For the TCP sessions, 44772 (28.1%) are empty (i.e. had no TCP payload). 20994 of these contain requests that are empty, the remaining TCP sessions have a request but no response. Within TCP sessions, the framework currently only supports identification of HTTP requests and responses. There are 118,035 HTTP requests and 112,034 HTTP responses. This means 74% of the TCP sessions contained HTTP requests and 70.3% of sessions contain the HTTP response protocol. In [112, pp. 2–3] it is stated that 58.6% of samples generate HTTP traffic; in the MALICIA dataset 98.5% of samples generate HTTP requests. This truly marks HTTP as a popular protocol used by malware, even allowing for some samples only containing HTTP traffic from the MALICIA capture framework.

In total 449 different TCP destination ports (for TCP sessions) are used by malware in the MALICIA dataset. The most popular TCP ports for the TCP sessions are 8000 (42164 sessions, 26.4% of sessions, 87.2% of pcaps), 80 (37770 sessions, 23.7% of sessions, 62.4% of pcaps) and 8080 (35788 sessions, 22.4% of sessions, 6.8% of pcaps). Port 443 is used in only 8393 sessions (5.3%) which suggests that most of the malware samples executed have not switched to use of SSL/TLS to protect their communications. This could be because of the difficulty of obtaining a signed and trusted certificate in an untraceable manner. Even if a self-signed certificate is used that opens up the possibility of identifying the network traffic by use of the self-signed certificate, or of identifying a binary file which contains a fingerprint of a fixed certificate to verify (that may form a fixed string in the binary which can be identified by anti-virus products).

DNS

The Sandnet paper [112] finds that some malware uses hardcoded DNS resolvers, i.e. they do not use the local DNS server. In the entire MALICIA dataset there are 45 different DNS servers contacted, nine of which are public IP addresses and not the local DNS server. The most popular of these nine is Google’s DNS server (“8.8.8.8”) which is present in 3774 network captures (22.2%), followed by a German IP address (“83.133.123.20”) present in 1613 captures (9.5%). The complete list of DNS servers can be found in Appendix N. Other than the Google DNS server, none appear to be from popular legitimate DNS services.

There are 1119 different DNS TTL values used in DNS answer resource A records in the MALICIA data set. The most common is 3600 (one hour) which occurs 10061 times (48.3% of all the DNS resource A record TTLs). This is followed by 60 seconds which occurs in 2008 (9.6% of all the DNS answer resource A record TTLs) DNS answer resource A records. There are 20 occurrences of DNS answer resource A records with a TTL of 0. The largest TTL observed is 345600 seconds (4 days). As the DNS TTL is a four byte field the maximum possible value is $2^{32} - 1$. A TTL of 0 can be used to prevent caching of the answer, whereas low TTLs can be used to provide flexibility as to the IP addresses a domain points to. This can be used by malware employing fast-flux techniques - using “rapid-changing DNS entries to build a hosting infrastructure with increased resilience” [53]. However, as [112] notes, there is no guarantee that the domains won’t belong to content distribution networks or other legitimate, popular, web sites. Every TTL between one and 60 appears in the dataset and would make good targets for further investigations into malware with fast-flux behaviour. A clustering algorithm on the DNS data could be used to group domains which exhibit fast-flux like behaviour, although this has not been implemented in this project due to time constraints.

16663 (98.2%) network captures contain a DNS query with type 12 (a PTR query) - typically a reverse DNS lookup (IP address to domain). This is the most popular of all DNS query types, beating A record queries (8306 pcaps (49%)). These findings differ to those in [112], in which the samples analysed all contain A record lookups and very few PTR resource record types are seen. One explanation for this is that the malware samples in the MALICIA dataset may make increased usage of hardcoded infrastructure (IP addresses) instead of using DNS queries to determine the IP
addresses of the servers they need to communicate with.

Only one traffic sample contains an MX lookup. 92 (0.5% of) samples contain DNS queries for AAAA records, used to resolve IP version 6 addresses for a domain name. This shows that in comparison to IP version 4, version 6 is rarely used by malware authors to host infrastructure (or victims acting as infrastructure/bots have an IP version 4 address). This matches the general state of the internet where IP version 6 usage is low - one publication [48] shows approximately 6% of connectivity using IP version 6 (as of 21 March 2015).

Figure 7.23 shows the distribution of resource record types in DNS queries amongst all the MALICIA samples. The huge usage of PTR queries (12) and A record queries (1) can clearly be seen.

Figure 7.23: DNS query resource record type distribution across all MALICIA samples

HTTP Requests

86.7% of network traffic captures utilise HTTP GET requests, whilst 86.8% use HTTP POSTs. GET requests are typically used to retrieve resources (e.g. further malware to run) and POSTs to send information to a server (e.g. stolen information about the infected computer) although there is no reason why a GET cannot be used to send information and a POST retrieve it. The most
The distribution of HTTP URI parameter values is much greater. There are 13429 different parameter values used across the dataset, many of which appear to be randomly generated (e.g. “39b257ef”). This is to be expected, especially if malware uses the parameter to indicate a status, unique victim ID, timestamp or some other data with high variability. If a piece of malware is sufficiently well understood (perhaps through static malware analysis of the assembly code) and the meaning of the parameter is known (e.g. volume label of a victim computer’s hard drive) then interpretation of family-wide traffic may be possible. However, the structure of a parameter value can be exceedingly useful. For instance, some values appear to be hashes 40 characters in length. Creating a regular expression for these is possible and can be used to identify similar traffic. Additionally some URI parameters appear obfuscated or URL encoded which indicates that some malware attempts to restrict access to its data (without employing encryption such as TLS).
There is a wealth of information that can be mined from the HTTP headers present in the MALICIA dataset. In total there are 24 different HTTP request headers - Figure 7.25 shows the distribution in the MALICIA dataset. Sorting by name reveals the letter case errors in a number of different headers: “accept-encoding” instead of “Accept-Encoding”, “Content-type” instead of “Content-Type” and “Content-length” instead of “Content-Length”. A recent study [73] discusses a variety of techniques to detect HTTP malware communications by exploiting the inaccurate implementation of HTTP, such as those that include extra whitespace at the end of header values or misspell header names. A specific example given in [73] is the header name “Proxy-Connection” misspelt as “Proxy-Connnetion”. The letter case examples I have identified above are subtly different to this - they are spelt exactly as the header should be, it is the case of the letter that distinguishes them. As such, case sensitivity of HTTP header names can be used as a detection technique for malicious network traffic in some instances.
The full list of HTTP request and response header names can be viewed in Appendix O.

The User-Agent HTTP request header is extremely common in the MALICIA traffic, as can be expected. This header is typically used to identify the software, and perhaps the version of software, that makes an HTTP request. In total 33 different User-Agents appear in the MALICIA traffic.

A fresh Windows XP install will use Internet Explorer as the default web browser. Several of the User-Agents claimed to be other browsers (“Opera/9 (Windows NT 5.1; ; x86)”), get the Internet Explorer User-Agent header wrong completely (“Internet Explorer”, no Mozilla version or other information) or appear to just leak information (“WINXPSIP3.7875768F8B00CED2”). Malware usage of HTTP User-Agents has been widely studied [60, 73, 112] - it is not the intention of this project to further advance this, but to highlight how the framework that has been developed can be used to study the User-Agents and quickly hone in on the values that might be of interest for further study, or that can be used to create a network IDS signature. A full list of User-Agents present in the MALICIA dataset is given in Appendix P. Many of these appear unique enough to be used standalone in a network IDS signature.

In Figure 7.26 three different Accept-Language headers are shown. One of these, “en-us” is present.

---
Figure 7.25: HTTP request header names and the number of MALICIA traffic captures they appear in.
in 526 samples, the others (“en” and “en-US”) occur in only two. As such they are a good identification of the malware traffic. A direct comparison to [112] is not possible as I do not know the language settings used when capturing the MALICIA traffic (although I presume en-us). However, it is reasonable to assume that some of the “en-us” header values are hardcoded into malware. As such, a dynamic malware analysis system, such as the capture framework designed for this project, could benefit if its localisation settings were set to a less popular value than English. This would make such hardcoded header values clear in any network captures.

Every other HTTP header can be mined for information which will help to identify malicious traffic and understand the behaviour of malware. For example, Figure 7.27 shows four Accept header values that do not appear to be very common. One in particular is very long and consists of unicode strings making it extremely unique. HTTP request Content-Type headers are populated by timestamped headers of the form “multipart/form-data; boundary=127.0.1.1.0.10003.1350610610.490.3”. This results in 14530 unique request Content-Types, of which 14526 are the timestamped headers. The most popular HTTP request Content-Type header is “application/octet-stream” occurring in 1413 different capture, and indicates binary data is being transmitted to a server.

Further HTTP request headers are not studied, but, as with the examples which have been presented, the analysis framework is capable of displaying details about the HTTP protocol, including

---

### Figure 7.26: HTTP Accept-Language header values present in the MALICIA dataset

![HTTP Accept-Language header values present in the MALICIA dataset](image)
the header names and values. This aids the identification of unique header names and values, and can help malware analysts to understand the behaviour and capabilities of malware, and of malware families.

Figure 7.27: HTTP Accept header values present in the MALICIA dataset and the number of samples (out of 16967) they are present in

HTTP Responses

Some of the 112,034 MALICIA HTTP responses originate from the MALICIA framework itself, rather than the server the malware is attempting to contact. Nevertheless, to allow some comparison to the Sandnet paper [112] the HTTP response Server values are listed in Appendix Q. The Server header is set by a web server and used to indicate what version of software is running. In total, 33 different Server header values are present. It is even possible to see how the authors of the MALICIA capture framework (probably) upgraded their version of Python (from 2.7.2 to 2.7.3 to 2.7.4) as they ran different samples over time.

Studying the Server header values can also aid understanding of how an attacker operates. For instance, all the malware samples analysed are targeted at the Windows operating system, yet many of the servers are running variants of a Linux operating system (Debian, Fedora etc.). This may indicate an attacker that is comfortable with multiple operating systems, one who wishes to diversify their implant from command and control infrastructure, or simply the pop-
ularity of operating systems for public web server hosting. Specific library versions are also available (e.g., “Apache/2.2.16 (Debian) PHP/5.3.3-7+squeeze3 with Suhosin-Patch mod_ssl/2.2.16 OpenSSL/0.9.8o mod_perl/2.0.4 Perl/v5.10.1”). As with other HTTP request features discussed, any HTTP response header name or value, if unique enough, may be used in a network IDS signature to identify the traffic as malware communications. As with other data, the values of these HTTP headers may be faked by malware or the command and control infrastructure.

By far the most prevalent HTTP response Content-Type is “text/html” as most responses are HTML. A variety of other values are present and are shown in Appendix R. If the Content-Types are accurate, there are not many samples which download binary data for further use. Of course this could be an artefact of how the MALICIA data is captured.

Due to the way the MALICIA traffic is captured (the responses often appear to be returned by the MALICIA framework), I expect the response headers to be of less use in identifying malware. However, there are 37 different HTTP response headers present, some of which suffer the same letter case problems as with the HTTP requests (e.g. “Content-length” appears in 3 traffic captures, “P3P” instead of “P3P” appears in 54) indicating that at least some of the responses may be from the real web servers. Three response header values appear in only one sample: “Microsoft-OfficeWebServer”, “X-Pad” and “X-Remote-IP”. Others appear in only a few samples. As such, the names of the headers may make good network IDS signatures capable of accurately detecting specific malware traffic with a low false positive rate. Certainly they can be used to differentiate between malware traffic from families that do not include those headers.

Interestingly, there is one header named “X-Sinkholed-Domain” which appears to indicate that a domain known to be associated with malware has been requested, but that an ISP or hosting company has sinkholed the request, i.e. redirected it to a server other than the intended one, presumably in an effort to protect customers. All values of this header are “You are infected with malware Feodo.” - Feodo is an alternative name for the Cridex malware. All six captures that contain this header are tagged as Cridex in the MALICIA dataset.

The list of HTTP response headers can be found in Appendix O.

7.4.5 MALICIA Analysis Summary

In section 7.4 an evaluation of the analysis framework has been conducted using the MALICIA dataset. In depth analysis of five individual traffic samples (section 7.4.1) is used to show the level of detail that the framework can provide, to help understanding of a specific threat. Cross-family analysis is then conducted (section 7.4.2) using the five samples and is used to validate the accuracy of the clustering algorithms. A number of experiments are conducted, each with different parameter values, to arrive at a correct clustering of the HTTP requests, grouped by malware family.

The focus then shifts to family-wide analysis (section 7.4.3) to demonstrate how different, but related, malware exhibits similarities and that these can be identified using the HTTP request clustering algorithms implemented. Some unique features of the malware families are identified to illustrate how they can be used in network IDS signatures to identify traffic. Finally, a comprehensive review of the entire MALICIA dataset is conducted (section 7.4.4) to demonstrate the utility of the feature extraction approach used by this framework. Statistics are given for a number of different network protocols along with an in depth study of DNS and HTTP traffic.
7.5  Full Framework Analysis

This section presents an experiment conducted using the entire framework. The methodology used is similar to that described in section 7.3. An additional first step is required as specified below:

1. submit the file (executable) to the capture framework with a timeout set to 600 seconds (10 minutes). The output of this step will be a pcap uploaded to the analysis framework, containing any network traffic observed while running the malware sample.

Analysis then proceeds exactly as conducted for the individual pcap samples presented in section 7.4.1, although the only collection of pcaps analysed are the ones generated by the capture framework (i.e. the MALICIA dataset is not used at all). 600 seconds is used as a timeout value to allow the malware time to execute and perform any initial activity, such as beaconing to its command and control infrastructure. A longer timeout could have been used (for example one hour as in [112]) and may have advantages in certain situations, e.g. if a malware executable requires interaction from the attacker then a longer timeout, such as several days, may be one way of maximising the chance that the actor is present and able to control their malware.

7.5.1  Malware Executed

Approximately 30 samples obtained from VirusTotal were run in the capture framework but most either failed to execute or generated little/no network traffic, and so were discarded. Manual inspection of the pcaps that were generated revealed that a lot of samples were unable to resolve the domain names they queried so stopped executing. Possible reasons for this include the DNS provider used (OpenDNS) blocking known malware domains and the use of random domain names which have not been registered and so do not resolve to an IP address.

Two malware samples did execute and generate a reasonable amount of traffic that could be studied. These are listed in Table 7.5 and both are 32 bit Windows executable (Portable Executable) files. Note, due to the lack of consensus for the anti-virus results present in VirusTotal it is not possible to be 100% sure that the malware does belong to the family claimed.

<table>
<thead>
<tr>
<th>Malware family</th>
<th>MD5</th>
<th>SHA1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zeus/zbot variant</td>
<td>3d6046e1218b525805e55d86dc605361</td>
<td>cdae6fee42288a8cbf8de7ed683b6f325b4a71ef</td>
</tr>
<tr>
<td>Dridex</td>
<td>ab32064691e52c89b7ac2086ed5dc934</td>
<td>0865f13f61cc009c55b4baa333975b1962359d0f5</td>
</tr>
</tbody>
</table>

Table 7.5: Malware family and hashes of the two samples analysed using the capture framework

The pcap files generated are available in the project repository (see section 5.6).

Zeus/zbot Variant - cdae6fee42288a8cbf8de7ed683b6f325b4a71ef-zbot.pcap

As explained in section 7.4.1 during analysis of the Zeus (zbot) sample from the MALICIA dataset, Zeus is a crimeware toolkit. In May 2011 the source code for Zeus was leaked and became available for other malware authors to build upon [83]. It is possible that this sample is such a variant as the first submission of it to VirusTotal was in 2013.

In total there are 458 packets and 50 TCP sessions in this pcap. The vast majority of packets contain TCP. There are only eight UDP packets. Other than the pre-configured DNS resolver
the only public IP address present is “204.11.56.48”. Two DNS queries are made for the domain “vivaspace2013.com” which resolves to this IP address and has a TTL of 300. The top TCP destination port is 80, which represents all the outbound TCP traffic. As expected this is web traffic.

There are 50 web requests made in total, all have identical header names as can be seen in Figure 7.28. There is one HTTP response.

Every request is a POST and the User-Agent is always “Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)”. The Host field is always “vivaspace2013.com” which matches the domain requested in the DNS query. The URI is always “/C270suqdh/file.php”. The histogram of the POST body lengths can be seen in Figure 7.29. Whilst every HTTP POST does appear to be the same, the body of the POST is slightly different in some of the requests. This may be a beaconing attempt by the malware, trying to open a communications channel with its infrastructure. It may also be trying to send some information about the victim operating system back to the server in the POST body. 122 bytes of data is enough to include information such as the user account name and operating system version. However, it is not possible from the information available to determine the purpose of the HTTP POST.

Figure 7.28: cdae6fee42288a8cbf8de7ed683b6f325b4a71ef-zbot.pcap HTTP header names
Interestingly, the one and only HTTP response is a ‘404’ which means the resource requested from the server (“/C270suqdh/file.php”) has not been found. One explanation as to why the malware repeatedly performs the same HTTP request is that it is badly coded and doesn’t take into account that the resource it wants might not be available.

The transport clustering results can be seen in Figure 7.30. The largest cluster contains many of the packets destined for port 80 (all the empty packets with SYN or SYN/ACK flags for example).
As all 50 HTTP requests are the same (with respect to the features measured), they appear in the same cluster when a distance threshold of 0.1 is used. Thus, any network IDS signatures used to uniquely identify this malware’s communications could use the HTTP features previously discussed and be sure they had coverage of all the (known) traffic.

Dridex - 0865f13f61c009c5fb4baa333975b1962359d05-dridex.pcap

Dridex is an evolution of Cridex malware, previously analysed in section 7.4.1. It is a relatively new threat having first been observed in November 2014 and it appears to be used to steal personal and financial information [78]. The sample I obtained was first submitted to VirusTotal in February 2015.

There are 60 packets and eight TCP sessions in this pcap. The majority of the packets are TCP, with four UDP packets used for “NBT Datagram Packet” - NetBIOS over TCP. No DNS is present. Four public IP addresses are contacted: “202.44.54.5”, “5.196.241.196”, “66.110.179.66” and “92.63.87.13”. As no DNS lookups are performed these IP addresses must be hardcoded into the malware. The only two TCP destination ports present are 8080 (24 packets) and 80 (12 packets). The distribution of TCP packet data lengths is shown in Figure 7.31. Other than empty TCP packets (typically SYN, SYN/ACK packets) 263 bytes is the most favoured data length, possibly indicating that the same TCP request (or response) is used repeatedly. However, this is not at the expense of other sized TCP packets, the largest of which has a 310 byte payload.
Examining the session data reveals that there are four HTTP requests and four HTTP responses. These are evenly distributed between the four public IP addresses and two TCP ports previously mentioned. The URI for every request is "/", i.e. there are no parameter names or values. The histogram of the HTTP request headers shown in Figure 7.32 reveals an interesting characteristic of the traffic. The same HTTP request headers are present in all four of the requests, except for one request which has no User-Agent header. If a network IDS signature was based on the User-Agent value it would miss the session that contained that request.
Inspection of the User-Agent values reveals two are in use: “Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/36.0.1985.143 Safari/537.36” and “Mozilla/5.0 (Windows NT 6.3; WOW64; rv:34.0) Gecko/20100101 Firefox/34.0”. This appears to be an attempt by the malware to camouflage itself as either the Chrome or Firefox web browser.

With the exception of the Content-Length header, the remaining headers prove extremely interesting to study. The Accept header has a different value in every request (“application/*”, “image/png,image/*/q=0.8,*/*;q=0.5”, “text/*” or “text/html, application/xhtml+xml, */*”). The Connection header is either “Close” or “Keep-Alive”. The Content-Type is one of four values: “application/pdf”, “image/png”, “model/mesh” or “video/mpeg”. It appears that the malware is selecting values for headers from a preconfigured list. Further evidence for this appears in the Host and Referer headers. Other malware has been known to spoof HTTP header information [72].

**Note:** Referer is a misspelling of Referrer in the HTTP specification.

An HTTP request uses a Referer header to specify the previous resource/web page that linked to the newly requested one. A user browsing different web sites would typically have the Referer header set by their web browser upon clicking a hyperlink to open a new web page. This is available to the new web server and can be used for purposes such as checking the search terms that a user entered to find a web site (if clicking through from a search engine result) or identifying affiliate sites that send a lot of traffic to a partner web site.
The Referer headers in this pcap are “https://aol.com/” and “https://facebook.com/”. There are two problems with this. RFC 7231 states “If the target URI was obtained from a source that does not have its own URI (e.g., input from the user keyboard, or an entry within the user’s bookmarks/favorites), the user agent MUST either exclude the Referer field or send it with a value of “about:blank”” [31]. Yet in this pcap the malware is setting a Referer even though there is none to legitimately set. As with other headers, the values are probably selected from a preconfigured list.

The second problem is that the HTTPS protocol is specified in the Referer. RFC 7231 states “A user agent MUST NOT send a Referer header field in an unsecured HTTP request if the referring page was received with a secure protocol” [31]. This means that a URI requested using HTTPS should not appear in the Referer header. Yet in the traffic generated by this malware it clearly is. Legitimate web browsers are unlikely to do this. Whilst not mandated by the specification, the risk of information disclosure is too great to transmit an HTTPS URL over an HTTP channel as it may contain sensitive information such as session information relating to a user login. To put this theory to the test I performed some manual testing with Firefox version 36.0.4 to check when the Referer field is present. This revealed the following behaviour:

- **HTTP to HTTP** - Referer transmitted
- **HTTP to HTTPS** - Referer transmitted
- **HTTPS to HTTPS** - Referer sometimes transmitted, e.g. if the domain is the same for both links, or perhaps based on the policies of the web server. More study would be required to determine the exact cases when the Referer is transmitted.
- **HTTPS to HTTP** - Referer not transmitted

Based on this limited experiment, it would appear that HTTPS Referers should not appear in HTTP traffic. This could form the basis of a heuristic network IDS signature that may work across different malware families that fail to observe this. Further investigations into when HTTPS to HTTPS Referer headers are legitimately sent are not conducted but would make for interesting further work as there are security and privacy concerns over the transmission of what may be sensitive information.

The final HTTP request header to be examined is the Host header. The four different values are shown in Figure 7.33. Of note is the fact that instead of a “.” before the top level domain, there is a space character. This is an obvious flaw in the HTTP implementation the malware uses. As there is no DNS traffic recorded, and there appears to be no recognisable words in the names, the domains present in the Host header are likely to be randomly generated. A network IDS signature that looks for Host header fields with a space before the top level domain may be able to identify this traffic.
All HTTP responses have the Server header set to “Microsoft-IIS/8.5” and have a status code of 403 which means the server does not want to respond to the request. This could be an attempt by the malware authors to protect their infrastructure, i.e. only allow victims they have targeted to communicate with their infrastructure, or it could be the result of an entity such as an anti-virus firm or hosting provider taking control of the IP address and setting up a web server to reject all communications to try and protect victims of the malware.

As all four HTTP requests are the same (with respect to the features measured), they appear in the same cluster when a distance threshold of 0.1 is used. Thus, any network IDS signatures used to uniquely identify this malware’s communications could use the HTTP features previously discussed and be sure they had coverage of all the (known) traffic.

**Zeus and Dridex**

As both samples analysed generated traffic which formed one cluster, I expected that when analysed together, the traffic from the two different pieces of malware would clearly form two clusters and be completely separate. As can be seen in Figure 7.34, clustering into k clusters with k set to 2 accurately separates the network traffic from the two malware samples. The four Dridex requests are in cluster 1 whilst the 50 Zeus requests are in cluster 2. The same results can be achieved using distance threshold clustering and a threshold of 0.1.
7.5.2 Capture Framework Analysis Summary

This section has demonstrated the use of the framework in its entirety. Firstly, the capture framework is used to execute malware samples and the network traffic recorded and saved as pcap files. These are automatically uploaded to the analysis framework and a study conducted on both samples individually to build up an understanding of what the traffic consists of and what is unique or identifiable. Mistakes made by the malware authors are pointed out and discussed. Finally, HTTP requests from both samples are clustered and clearly separated by the clustering algorithm used. Such an approach allows selection of similar traffic and can be combined with the in depth analysis of a specific sample to find ways to identify the network traffic for the malware.
Chapter 8

Conclusions

This chapter concludes the project. Future work that could be undertaken is discussed, including how this relates to existing research. The project objectives are revisited and evaluated for success. Finally, overall conclusions are presented by summarising the work conducted, the contributions of this project to the field of malware analysis and the results achieved.

8.1 Future Work

A number of improvements to the framework could be made, many of which were left out of the implementation due to time constraints and to avoid having a project with too broad a focus. This section describes several of these improvements.

Some feature information is extracted but not currently used. For example, the time that each packet was recorded is saved as a feature of a packet. This could be used to identify time-based behaviour, such as noisy network communications or periodic activity by malware as identified in [14]. There is also scope to pair certain types of features, for instance source and destination IP addresses or source and destination port numbers. Each feature can then be thought of as a node on a graph, with pairs connected by an edge. This would enable visualisations such as plotting the IP address pairs that communicate as has been done in recent work for HTTP(S) requests [51]. Plotting the IP addresses opens up the possibility of the visualisation using more of the ‘click-through’ behaviour currently present. For example, clicking on an IP address could display all features relating to that IP address.

To further expand on the features currently available, a simple measure would be to calculate statistics for every field. For example, the length of an HTTP Host, the length of a URI or the length of an HTTP Referer string. These could themselves be exposed as new fields, e.g. ‘session request http request Referer length’. Averages of these could then be calculated, for example, the average length of a domain name present in the network traffic from a piece of malware. This could indicate new behaviour by malware, for instance if different malware samples always used domain names that are 20 characters in length. This type of analysis best suits textual, rather than numeric, data. For example, it could help determine if DNS covert channels [76, 93, 135] were present in the data, particularly if they resulted in a many DNS queries for extremely long hostnames.

Other types of features could be added. As mentioned in section 2.4, the network captures could be run through an IDS system and the resulting signature hits could be applied to packets/sessions as a feature. If a network capture is known to belong to a particular piece of malware (e.g. as with many of the captures in the MALICIA dataset) then each packet/session could be labelled with the malware name (i.e. as a feature). This was indirectly done in this project by using the description field for each pcap to hold the malware family name as identified in the MALICIA dataset or from VirusTotal. By having a feature for each packet/session which represents the malware name or network IDS signature hit, non-malicious traffic may be more easily ignored.

As mentioned in section 7.5.1, one malware sample spoofed the HTTP Referer header value and
used the HTTPS protocol. To fully understand the security and privacy issues over the transmission of what may be sensitive information, I conducted some limited testing into when browsers set a Referer header with an HTTPS protocol. Expanding on this, using real web browser traffic, would be an interesting area of further research.

Having a pot of non-malicious traffic to compare data too would have been useful at times. As stated in section 1.3, discovery of new malware samples, or confirmation that traffic belongs to malware, is out of scope for the project. However, having a pot of non-malicious network traffic, as used in [107], would have aided analysis by providing a baseline for ‘normal’ traffic. For example, if a piece of malware uses a hardcoded User-Agent, knowing if it ever appears in benign traffic is helpful to determine if the User-Agent can be included in a network IDS signature, i.e. if such a signature is likely to generate false positive hits.

The implementation of the framework uses a custom HTTP parser. HTTP is relatively easy to parse and extract defined fields, due to its textual nature. However, creating more protocol parsers would be a lot of work and possibly error prone. Using an existing solution such as tshark [42] or Bro IDS [106], as has been done in [3], would allow many more session protocols to be supported. However, this may be at the expense of not being able to produce as many features (without modifying the source code of another application). Focus on HTTP traffic in this project is justified by 98.5% of the MALICIA dataset containing such traffic.

Other research [129] has demonstrated the value of using a system such as Splunk Enterprise [54] to provide a query language. Such a tool could be integrated into the framework to allow data to be queried using an expressive language that supports more operations than the current JSON API. An alternative would be to expand the JSON API to provide similar functionality, such as allowing a subset of a feature to be selected (e.g. histogram only the User-Agents that are from HTTP requests with a GET method, rather than from all requests).

Several improvements can be made to the clustering used in this project. The current implantation of the transport protocol clustering operates on packets, whereas if the UDP data were to be ‘sessionised’ (this is performed in [112] for example) it would be possible to run the algorithm over transport protocol sessions. As many packets are usually contained within one session this could cut down the volume of data that needs to be processed and not present quite such a verbose output.

In practice, the packet clustering algorithm implemented proved not to be that useful as it generated too many clusters to examine for most network captures (or group of captures). Many packets are empty, such as TCP SYN and SYN/ACK packets, and there is often little value in studying them. Future packet clustering algorithms should not focus on clustering every packet but define which packets are in scope to be clustered (e.g. by focusing on a particular protocol or removing unnecessary packets such as empty TCP SYN packets).

The HTTP request clustering algorithm implemented did prove very useful to group related requests, and as such features heavily in the analysis in chapter 7. However, using a clustering validity index [52] such as the Davies-Bouldin (DB) (used in [100, 101]) or the Dunn index (used in [85]) could further aid the clustering, for example, by running the algorithm multiple times to minimise the distance between clusters generated. This could be applied if partitioning into k clusters or if using distance threshold clustering. This would save some manual work currently required to try the clustering algorithm multiple times with different parameters.

This leads on to the performance of the clustering algorithms. Runtimes reported in [101] and [107] are not directly comparable (different hardware, different clustering algorithms etc.). The feature extraction step for the MALICIA dataset is slower in this project than [107] but can be explained due to the much greater number of features extracted. The traffic clustering performance does not scale well to large amounts of data. Clustering all packets or sessions in the entire MALICIA dataset was just not possible on the hardware used within a reasonable timeframe. Implementing
a multi-step clustering process as described in [100] or [101] could go some way to remedying this. Note, it was not a goal of this project to use clustering algorithms that can operate on huge volumes of data, but as an improvement it is logical to allow analysis to take place over larger datasets. The design of the framework makes it possible to implement additional clustering algorithms alongside existing ones, so more scalable algorithms could be implemented alongside the current ones. Additionally, new algorithms that operate on different types of traffic could be implemented, for example, DNS traffic clustering or HTTP response clustering.

Finally, as has been demonstrated [100, 101, 107], network IDS signatures could be automatically generated to identify traffic in the clusters output by the framework. This would nicely complement the feature extraction approach taken in this project, as many signatures for malware traffic identify specific fields within network protocols that contain unique or uncommon values.

8.2 Project Objectives

As described in section 1.1, “the objectives for this report are split into core objectives, which MUST be achieved, and extension objectives, which MAY be achieved given sufficient time and are designed to expand on the core objectives”. I will step through each objective in turn and discuss whether it has been achieved, and if so, how.

Core objectives:

1. the design and implementation of a framework to extract information of interest from previously captured network traffic (whether from objective 6 or elsewhere, e.g. publicly available pcap files) and to produce visualisations to aid analysis (analysis framework)

   This objective has been achieved and forms the basis of the entire project. The design for the analysis framework is presented in section 3.3, the implementation detailed in section 5.2 and example visualisations shown in chapter 6.

2. the design and implementation of an API to allow automatic interaction with the analysis framework

   The API for the analysis framework has been incorporated into the design presented in section 3.3. The ‘public’ facing endpoints of the API are detailed in Table 3.2, Table 3.3 and Table 3.4. The API was used to gather the raw data for the analysis available in this report. Whilst perhaps not obvious to a user, the API is used by the web client to query the analysis framework. This objective has been met.

3. a study of previously captured network traffic from malware using the analysis framework

   This objective has been achieved and serves as a way of evaluating whether the framework designed and developed is actually of any use. The analysis present in chapter 7, in particular section 7.4, fulfils this objective and covers a variety of different malware. I show how it is possible to use the analysis framework to study network traffic from a piece of malware in great detail, and also how analysis of traffic from different samples can be conducted, whether they belong to the same malware family or different families. The entire MALICIA dataset is then studied and the benefits of my feature extraction approach are shown.

Each of the three core project objectives has been met. These were defined as objectives that MUST be met for the project to be considered a success. Based on this alone the project can be considered successfully completed. However, to further stretch myself and add extra value to the framework, five extension objectives were defined which expand on the core objectives in different ways.

Extension objectives:
4. **an exploration of visualisation methods to help identify malicious network traffic**

A brief discussion on visualisation is presented in section 2.3 and while developing the analysis framework I experimented with different visualisation libraries (section 5.3.2) to see what options were available. The final selection of visualisations used are presented in chapter 6. An alternative to implementing new visualisations would have been to review visualisation methods applicable to the type of data available in this project. However, due to time constraints, and as much more focus was placed on the other extension objectives, this extension objective can only be considered partially complete.

5. **an exploration of methods to cluster malicious network traffic**

This was the first extension objective I began work on. Previous research on malware clustering is reviewed in section 2.2 and this is used to inform the clustering algorithms specified in chapter 4. The clustering functionality is integrated into the analysis framework design in section 3.3 and implemented as new analysis modules. Several new endpoints are added to the API, shown in Table 3.4, to perform the clustering and to retrieve detailed results for packets/sessions within a cluster. The clustering, in particular the HTTP request clustering, features heavily in the analysis conducted in chapter 7, where it proves useful in separating different malware communications. This extension objective has been achieved.

6. **the design and implementation of a framework to run malware over a specified time period and to capture the resulting network traffic (capture framework)**

This extension objective has been achieved and really rounds out the framework as a whole. Instead of relying on previously captured traffic, malware samples can be collected and executed to generate network traffic. The virtual environment can be tailored to whatever setup is desired. The design for the capture framework is presented in section 3.2 and the implementation detailed in section 5.1.

7. **the design and implementation of an API to allow automatic interaction with the capture framework**

The API for the capture framework has been incorporated into the design presented in section 3.2. The ‘public’ facing endpoints of the API are detailed in Table 3.1. Whilst perhaps not obvious to a user, the API is used by the web client to submit samples to the capture framework for execution. This was the approach taken to submit the malware samples studied in section 7.5. The capture framework itself could change to use a different technology and this would have no impact on the ‘public’ facing API. This extension objective has been met.

8. **a study of identified malware using the capture framework and the analysis framework and an analysis of the network traffic observed**

The analysis conducted in section 7.5 uses both the capture framework and the analysis framework. Features of the network traffic generated are examined and mistakes made by the malware authors identified. The clustering algorithms are used successfully to separate traffic generated by the two malware samples studied. Thus, the final extension objective has been achieved.

Four of the five extension objectives have been completed successfully. One has been partially completed. However, the extension objectives were defined to allow the project room to grow within the time available, and as such completion of each extension objective is optional. I consider completion of four extension objectives a success and the work undertaken really complements the core framework and analysis.
8.3 Overall Conclusions

Final conclusions for this project are now presented. The work conducted is summarised and the contributions of this work to the field of malware analysis explained. The main results of the analysis undertaken are revisited and re-emphasised.

The project starts with research into malware analysis, the different types of malware analysis, and how network traffic from malware may be collected. Existing dynamic malware analysis systems, either publicly available or those proposed in research papers, are reviewed. Existing clustering techniques for malware behaviour are studied and this project’s focus explained: only network traffic is to be studied, not host features such as files created or processes launched. The topic of visualisation is then touched on to provide context for the visualisations required in this project. Network IDS signatures, one of the main methods used to detect the presence of malware on a network, are then described and an example signature presented. The literature search and background research section finishes with an explanation of the core malware analysis approach taken by this project, protocol feature extraction, and my existing experience relating to networking protocols and programming languages.

Chapter 3 presents the system design. The design of the whole framework is outlined and then broken down into its subcomponents: the capture framework, analysis framework and web client. The design is linked back to the relevant project objectives to explain how these have influenced it. Important design decisions taken for each component of the framework are summarised and justified. The intended flow of control through the web client is given to illustrate how a user might operate when analysing malicious network traffic using the framework.

Chapter 4 explains in detail the clustering used in the framework. This is split into two different types: packet clustering and session clustering. An algorithm is developed for each clustering type. The inspiration for the algorithms lies in existing research which was identified in the literature search phase of the project. My algorithms are adapted from the existing work and pseudocode is provided to illustrate them. The framework is designed in a modular way so that new algorithms can easily be included alongside existing ones. Part of designing a framework for analysing malicious network traffic, in my view, is that the framework is extensible. Otherwise, it is not a framework that has been developed but a specific implementation to suit a particular problem.

The different malware traffic capture types that can be analysed and clustered are then defined. These types are from: an individual malware sample, multiple samples from the same malware family, multiple samples from related malware families, multiple samples from unrelated malware families and finally from the entire dataset. These capture types feature in the analysis conducted later in the report. I then present my observations on the likely network IDS signature coverage based on the network capture types analysed and clustered. Chapter 4 concludes by making an important statement: the clustering implemented in this framework does not attempt to classify network traffic as malicious or not, but to partition the data into related communications. Those communications may or may not be generated by a piece of malware. That is determined by the network traffic input to the clustering algorithms. A recurring feature when the clustering algorithms were tested on the MALICIA dataset was the presence of traffic generated by the capture system, not malware. Fortunately, in most cases this was easily identifiable as it formed separate clusters to the malware traffic.

The implementation of the design is then described in chapter 5. The development of the framework required me to draw on my existing programming experience as well as learn new technologies. Throughout chapter 5 the specific choices made are explained and justified, for example the choice of TCP session reassembly tool is made only after evaluation of the different options available. Whilst the actual dynamic malware analysis system is not a key focus of the project, the choice of sandbox tool and customisations made are explained. The most important source code files
System security is discussed in section 5.5 and revisited in section 7.2. As malware is executed this is not a point to overlook. Running computer code known to be harmful to the intended operation of a system requires a few safeguards. The worst situation I imagined was losing the working copy of data for this project or infecting my home network! Fortunately the safeguards implemented were more than adequate in containing the malware.

Section 5.3 introduces the visualisation and web client implementation, including an early prototype version used. This is expanded upon in chapter 6 where example screenshots are shown to illustrate the visualisations. Further screenshots of the implementation are available in Appendix T.

The results of the experimentation phase of the project are available in chapter 7. This serves as an evaluation of the framework and to establish if it is actually of any use for analysing and visualising network traffic from malware. The main findings from the analysis are now summarised.

Analysis is split into two sections. The published MALICIA dataset is used for the bulk of the work whilst two malware samples are identified for the second section - testing the entire end-to-end framework.

Many of the features that can be extracted by the analysis framework do not feature in the analysis conducted. The checksum for each IP packet is one such feature. These features hold no value in relation to the study of malware. However, many features are useful to study when examining malware traffic. This is typically because malware uses an incorrect value of a protocol feature, uses a unique value, uses a fixed value or uses a value which does not make sense in the context of the host/network within which the malware resides. Examples of these from the traffic studied include:

- **incorrect feature value** - HTTP Host names observed with a space character instead of a “.” or HTTP header names with the wrong letter case, e.g. “accept-encoding” instead of “Accept-Encoding”
- **unique feature value** - domain names used by malware, such as “risparmioassicurativo.net”, or URLs such as “/0Pvo9Hnu/EpJbWNWD.exe”
- **fixed feature value** - repeated requests to the URI “/pony/gate.php” or the fixed User-Agent “Shareaza”
- **feature value inappropriate for a host/network** - the User-Agent “Mozilla/4.0 (compatible; MSIE 5.0; Windows 98)” observed from a Windows XP operating system

The visualisations available in the framework helped me to identify many of these features. For example, Figure 7.19 highlights two repeated values and in Figure 7.25 the uncommon values that rarely appear can clearly be identified by their almost non-existent bars in the bar chart.

Statistics across the entire dataset were calculated using the framework to inform where analysis should be focused. Great emphasis is placed on the HTTP application protocol, particularly as 98.5% of the MALICIA traffic captures contain HTTP. However, many other features were extracted and used throughout the analysis, including those from IP version 4, TCP, UDP and DNS. Payload size proved a useful method to infer if the same request (or response) was made repeatedly without having to study the content of the communication.

Many existing network IDS signatures use infrastructure to identify malware communications. The framework makes enumeration of infrastructure used by malware straightforward. Destination IP
addresses, DNS query names and the HTTP Host field were all used to identify infrastructure used by the malware samples studied.

The analysis using the end-to-end framework progressed in much the same way as that using the MALICIA dataset. However, a particular feature of one sample stood out. The Dridex malware set an HTTP Referer header value that started with HTTPS. There are legitimate security and privacy concerns over the type of information that could be revealed from a secure URL being transmitted over an insecure protocol. Section 8.1 suggests this as an area for further research.

Of the two clustering algorithms implemented, the HTTP request clustering algorithm proved most useful. In many of the experiments conducted, traffic generated by one malware sample was clustered into different groups. Clustering between different malware families led to a successful split of the traffic belonging to each malware. The next step, i.e. what to do with the requests in a cluster, is out of scope of this report. However, throughout this report repeated suggestions are made that the features of the requests contained within a cluster may be used to write network IDS signatures, each of which identifies specific malware network behaviour.

I learnt a great deal about how the malware samples studied use a variety of networking protocols for communication. This was greatly aided by the framework. The visualisations available really helped me to identify features of interest and interpret the raw data available. Overall, I would say that the framework is of use for analysing and visualising network traffic from malware.

To recap, the main contributions of this work are:

- a framework for analysing and visualising network traffic from malware - the full Sandet++ source code is available and has been designed and implemented in such as way as to be extensible should other researchers wish to build upon it
- an API which can be accessed in an automated manner - the framework API can be utilised by other systems that require the information Sandnet++ can provide about network traffic
- a web client that can visualise features of network traffic - set visualisation types are provided and more can easily be added
- an approach to malware analysis that focuses on features of network traffic - a feature may be a networking protocol field, such as a TCP port number, or another attribute of the network traffic, such as the packet size or protocol name. As many features as possible are identified and made available for analysis using the framework
- two new clustering algorithms, modified from existing ones - one algorithm operates on TCP/UDP packets, the other on HTTP requests
- a study of the entire MALICIA dataset with specific malware traffic examined in detail - key findings are reported and notable observations highlighted
- a study of two malware samples - both malware samples are executed for a period of ten minutes, the network traffic is recorded and then a study of the network behaviour is conducted, with key findings and notable observations reported
Chapter 9

Bibliography


Appendix A

Analysis Histograms Raw Data

An example of the raw JSON data returned by a call to the histogram endpoint of the analysis framework API, is show in Listing A.1.

```json
[
  {
    "count": 129,
    "packet.ip.dst": "192.168.0.2"
  },
  {
    "count": 72,
    "packet.ip.dst": "192.168.0.1"
  },
  {
    "count": 1,
    "packet.ip.dst": "8.8.8.8"
  }
]
```

Listing A.1: Example JSON response from the analysis framework API histogram endpoint
Appendix B

Analysis Clustering Raw Data

A snippet of the raw JSON returned by a call to the packet transport clustering endpoint is shown in Listing B.1.

```json
[
  {
    "dst_ip": "192.168.0.2",
    "proto_name": "tcp",
    "data_len": 657,
    "dst_port": 8000,
    "feature_vectors": [
      {
        "dst_ip": "192.168.0.2",
        "proto_name": "tcp",
        "src_port": 37081,
        "upload_id": "9",
        "src_ip": "192.168.0.1",
        "data_len": 657,
        "dst_port": 8000,
        "pkt_num": 547
      }
    ]
  },
  {
    "dst_ip": "224.0.0.251",
    "proto_name": "udp",
    "data_len": 138,
    "dst_port": 5353,
    "feature_vectors": [
      {
        "dst_ip": "224.0.0.251",
        "proto_name": "udp",
        "src_port": 5353,
        "upload_id": "9",
        "src_ip": "192.168.0.1",
        "data_len": 138,
        "dst_port": 5353,
        "pkt_num": 5
      },
      {
        "dst_ip": "224.0.0.251",
        "proto_name": "udp",
        "src_port": 5353,
        "upload_id": "9",
        "src_ip": "192.168.0.1",
        "data_len": 138,
        "pkt_num": 138,
```
Listing B.1: Feature vectors returned by a call to the packet transport clustering endpoint

Listing B.2 shows the raw JSON data returned by a call to the session (HTTP request) clustering endpoint.
Listing B.2: Feature vectors returned by a call to the session (HTTP request) clustering endpoint
Appendix C

Software Used

The list of software and programming/scripting languages used in this project is as follows:

- **Python 2** - used with Scapy to extract packet features
  - *Scapy* - used for packet processing of pcap files
- **Python 3** - used for the bulk of the framework
  - *Flask* - provides a basis for a web server and handles all HTTP communications for the framework API
  - *flask-compress* - provides support for compressing the HTTP responses returned to a client by the framework API
- **Tcptrace** - used to reconstruct TCP sessions
- **Bourne shell** - used in helper scripts to launch the web servers for the framework
- **Bash shell** - used to automate submissions to the capture framework and forward them to the analysis framework
- **JavaScript** - used for the web client
  - *Bootstrap* - used to apply consistent styling to the interface
  - *Jasny Bootstrap* - contains useful extensions for Bootstrap such as hiding menus off canvas and table row selection
  - *bootstrap-select* - an extension to Bootstrap, used to provide dropdown boxes that are searchable
  - *Bootstrap Table* - an extension to Bootstrap, used for all tables within the web client. Provides table pagination and searching.
  - *jQuery* - used for its plethora of helpful JavaScript functions
  - *Knockout* - used to map data models to the web interface
  - *Highcharts* - used for most of the visualisation, e.g. creating bar charts and line charts.

*Note: many software libraries, such as individual Python 3 modules are not listed.*

In addition, the following software was used in the production of this report and the framework:

- **TeXstudio** - a LaTeX editor used in production of this report
- **Eclipse** - an Integrated Development Environment (IDE) used to write all the code for the framework
- **Dia** - used to draw the framework design
Appendix D

Cuckoo Sandbox Virtual Networking

On a Linux Mint host the following commands can be run to enable a virtual machine to communicate out to the internet:

```bash
1 sudo iptables -A FORWARD -o eth0 -i vboxnet0 -s 192.168.56.0/24 -m conntrack --ctstate NEW -j ACCEPT
2 sudo iptables -A FORWARD -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
3 sudo iptables -A POSTROUTING -t nat -j MASQUERADE
4 sudo sysctl -w net.ipv4.ip_forward=1
```

Listing D.1: Bash commands to enable a Cuckoo Sandbox virtual machine to communicate out to the internet

To make this persistent across reboots the following extra commands may be run (assuming a user account called “sandnet”):

```bash
1 sudo iptables-save > /home/sandnet/iptables.rules
2 sudo vi /etc/rc.local
3
4 Then add a line with:
5
6 iptables-restore /home/sandnet/iptables.rules
7
8 sudo vi /etc/sysctl.conf
9
10 Uncomment the following line:
11
12 net.ipv4.ip_forward=1
```

Listing D.2: Bash commands to renew iptables rules at reboot

These settings are based on instructions obtained from [36].
Appendix E

Example Packet Feature Data

Listing E.1 presents an example of the raw JSON data used to describe packet features. This example represents DNS data.

```json
{
    "timestamp": 1341304469.2868,
    "pkt_num": 934,
    "protocols": [
        {
            "name": "Ethernet",
            "short_name": "ethernet",
            "fields": {
                "src": "de:ad:be:ef:00:00",
                "dst": "56:0a:33:77:44:7b",
                "type": 2048
            },
            "b64_fields": {}
        },
        {
            "name": "IP",
            "short_name": "ip",
            "fields": {
                "dst": "192.168.0.1",
                "tos": 0,
                "version": 4,
                "ihl": 5,
                "options": "b'[]'",
                "len": 60,
                "flags": 0,
                "frag": 0,
                "ttl": 128,
                "src": "192.168.0.2",
                "id": 560,
                "chksum": 46893,
                "proto": 17
            },
            "b64_fields": {
                "options": "W10=
            },
        },
        {
            "name": "UDP",
            "short_name": "udp",
            "fields": {
                "dns_server": "192.168.0.1",
                "sport": 50084,
            }
        }
    ]
}
```
"dport": 53,
"data_len": 32,
"len": 40,
"chksum": 22289
},
"b64_fields": {},
},
{
"name": "DNS",
"short_name": "dns",
"fields": {
"aa": 0,
"ancount": 0,
"an": null,
"rcode": 0,
"rd": 1,
"tc": 0,
"opcode": 0,
"qr": 0,
"qdcount": 1,
"qd-qclass": [1],
"qd-qtype": [1],
"qd-qname": ["www.google.com."]
},
"arcount": 0,
"ar": null,
"nscount": 0,
"ns": null,
"id": 54455,
"ra": 0,
"z": 0
},
"b64_fields": {}}

Listing E.1: Example packet feature data shown as JSON data returned from the analysis framework. This example represents a DNS packet.
Appendix F

Example Session Feature Data

Listing F.1 displays an example of the raw JSON data used to describe session features. This example represents an HTTP session.

```
{
    "src_ip": "192.168.0.2",
    "dst_ip": "178.32.190.142",
    "src_port": "1032",
    "dst_port": "80",
    "session_num": 5,
    "request_data_len": 169,
    "response_data_len": 362,
    "data_len": 531,
    "request_num_packets": "5",
    "response_num_packets": "5",
    "fin_closed": true,
    "request_protocols": [
        {
            "name": "Http Request",
            "short_name": "http_request",
            "data_len": 169,
            "fields": {
                "method": "GET",
                "uri": "/stat2.php?w=30461&i=000000000000000000000000d5acaf6e&a=1"
            }
        },
        {
            "name": "User-Agent",
            "short_name": "user_agent",
            "data_len": 169,
            "fields": {
                "User-Agent": "Opera/6 (Windows NT 5.1; ; LangID=409; x86)"
            }
        },
        {
            "name": "Host",
            "short_name": "host",
            "data_len": 169,
            "fields": {
                "Host": "aikezjuq.cn"
            }
        },
        {
            "name": "Connection",
            "short_name": "connection",
            "data_len": 169,
            "fields": {
                "Connection": "close"
            }
        }
    ]
}
```
"Host": "aikezjuq.cn",
"User-Agent": "Opera/6 (Windows NT 5.1; ; LangID=409; x86)",
"Connection": "close"
}
],
"response_protocols": [
{
"name": "Http Response",
"short_name": "http_response",
"data_len": 362,
"fields": {
"status_code": "200",
"reason_phrase": "OK",
"version": "1.1",
"body_len": 21,
"headers": [
{
"Date": "Mon, 03 Jan 2011 10:51:33 GMT"
},
{
"Server": "Apache"
},
{
"Set-Cookie": "CG=ES:29:Madrid; path=/"
},
{
"Accept-Ranges": "bytes"
},
{
"Cache-Control": "max-age=60, private"
},
{
"Expires": "Mon, 03 Jan 2011 10:52:33 GMT"
},
{
"Content-Type": "text/html"
},
{
"Vary": "User-Agent,Accept-Encoding"
},
{
"Content-Length": "17"
},
{
"Keep-Alive": "timeout=2, max=10"
},
{
"Connection": "Keep-Alive"
}
],
"Vary": "User-Agent,Accept-Encoding",
"Set-Cookie": "CG=ES:29:Madrid; path=/",}
Listing F.1: Example session feature data shown as JSON data returned from the analysis framework. This example represents an HTTP session.
Appendix G

Example Clustered Packets

Listing G.1 displays an example of the raw JSON data used to describe packets in a cluster. This example cluster contains two DNS packets.

```json
[
    {
        "selected_packets": [
            {
                "pkt_num": 1,
                "timestamp": 1427291659.36372,
                "protocols": [
                    {
                        "short_name": "ethernet",
                        "b64_fields": {},
                        "fields": {
                            "dst": "0a:00:27:00:00:00",
                            "type": 2048
                        },
                        "name": "Ethernet"
                    },
                    {
                        "short_name": "ip",
                        "b64_fields": {
                            "options": "W10=
                        },
                        "fields": {
                            "src": "192.168.56.101",
                            "ihl": 5,
                            "chksum": 35932,
                            "proto": 17,
                            "version": 4,
                            "flags": 0,
                            "dst": "208.67.222.222",
                            "frag": 0,
                            "ttl": 128,
                            "options": "b'[\']",n",
                            "len": 63,
                            "tos": 0,
                            "id": 1570
                        },
                        "name": "IP"
                    },
                    {
                        "short_name": "udp",
                        "b64_fields": {},
```
"fields": {
  "dns_server": "208.67.222.222",
  "len": 43,
  "chksum": 8949,
  "dport": 53,
  "data_len": 35,
  "sport": 1025
},
"name": "UDP"
},
{
  "short_name": "dns",
  "b64_fields": {},
  "fields": {
    "ar": null,
    "ancount": 0,
    "nscount": 0,
    "qd-qclass": [1],
    "qd-qname": ["vivaspace2013.com."]
  },
  "ns": null,
  "ra": 0,
  "rd": 1,
  "rcode": 0,
  "z": 0,
  "qdcount": 1,
  "an": null,
  "qd-qtype": [1],
  "tc": 0,
  "aa": 0,
  "qr": 0,
  "opcode": 0,
  "id": 43745,
  "arcount": 0
},
"name": "DNS"
}
}

"pkt_num": 277,
"timestamp": 1427292131.909731,
"protocols": [
  {
    "short_name": "ethernet",
    "b64_fields": {},
    "fields": {
      "dst": "0a:00:27:00:00:00",
      "type": 2048
    }
  }
]


```json
},
  "name": "Ethernet"
},
{
  "short_name": "ip",
  "b64_fields": {
    "options": "W10=
  },
  "fields": {
    "src": "192.168.56.101",
    "ihl": 5,
    "chksum": 20822,
    "proto": 17,
    "version": 4,
    "flags": 0,
    "dst": "208.67.222.222",
    "frag": 0,
    "ttl": 128,
    "options": "b'[]'",
    "len": 63,
    "tos": 0,
    "id": 16680
  },
  "name": "IP"
},
{
  "short_name": "udp",
  "b64_fields": {},
  "fields": {
    "dns_server": "208.67.222.222",
    "len": 43,
    "chksum": 28888,
    "dport": 53,
    "data_len": 35,
    "sport": 1025
  },
  "name": "UDP"
},
{
  "short_name": "dns",
  "b64_fields": {},
  "fields": {
    "ar": null,
    "ancount": 0,
    "nscount": 0,
    "qd-qclass": [1
    ],
    "qd-qname": ["vivaspace2013.com."
    ],
    "ns": null,
    "ra": 0,
    "rd": 1,
    "rcode": 0,
```

Page 128 of 179
Listing G.1: Example cluster containing two DNS packets. The raw JSON data returned from the analysis framework is shown.
Appendix H

Visualisation Options

<table>
<thead>
<tr>
<th>Data type</th>
<th>Visualisation</th>
<th>Visualisation options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature histogram</td>
<td>Table</td>
<td>histogram field, histogram type and sort order</td>
</tr>
<tr>
<td></td>
<td>Bar chart</td>
<td>histogram field, histogram type, sort order and label rotation</td>
</tr>
<tr>
<td></td>
<td>Column chart</td>
<td>histogram field, histogram type, sort order and label rotation</td>
</tr>
<tr>
<td></td>
<td>Line chart</td>
<td>histogram field, histogram type, sort order and label rotation</td>
</tr>
<tr>
<td></td>
<td>Area chart</td>
<td>histogram field, histogram type, sort order and label rotation</td>
</tr>
<tr>
<td></td>
<td>Pie chart</td>
<td>histogram field, histogram type, sort order and label rotation</td>
</tr>
<tr>
<td>Packet (transport layer) clustering</td>
<td>Table</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>Pie chart</td>
<td>label rotation</td>
</tr>
<tr>
<td>Session (HTTP request) clustering</td>
<td>Table</td>
<td>clustering type, k, randomMultiLongestEdges and threshold</td>
</tr>
<tr>
<td></td>
<td>Pie chart</td>
<td>clustering type, k, randomMultiLongestEdges, threshold and label rotation</td>
</tr>
</tbody>
</table>

Table H.1: Visualisations and their options available in the web client
Appendix I

Tcptrace Output

Listing I.1 displays the command to run tcptrace in brief output mode, with no host resolution and with the output of each TCP session extracted into its own file. The output of the command when run on a sample from the MALICIA dataset is also shown.

```
sandnet@sandnetHost /tmp $ tcptrace -b -n -e /opt/sandnet/uploads/pcaps/7116.pcap
1 arg remaining, starting with '/opt/sandnet/uploads/pcaps/7116.pcap'
Ostermann’s tcptrace -- version 6.6.7 -- Thu Nov 4, 2004

574 packets seen, 103 TCP packets traced
elapsed wallclock time: 0:00:00.001707, 336262 pkts/sec analyzed
trace file elapsed time: 0:10:05.833436
TCP connection info:
1: 10.3.9.89:1033 - 10.9.8.7:5432 (a2b) 5> 6< (complete)
2: 10.3.9.89:1034 - 10.9.8.7:6543 (c2d) 71> 0< (unidirectional)
3: 10.3.9.89:1035 - 10.9.8.7:1234 (e2f) 5> 4< (complete)
4: 10.3.9.89:1039 - 10.9.8.7:1234 (g2h) 5> 4< (complete)
5: 10.3.9.89:1040 - 190.9.35.199:80 (i2j) 3> 0< (unidirectional)
```

Listing I.1: Example output from tcptrace when run in brief output mode

Listing I.2 displays the command to run tcptrace in long output mode, with no host resolution and with the output of each TCP session extracted into its own file. The output of the command when run on a sample from the MALICIA dataset is also shown.

```
sandnet@sandnetHost /tmp $ tcptrace -l -n -e /opt/sandnet/uploads/pcaps/7116.pcap
1 arg remaining, starting with '/opt/sandnet/uploads/pcaps/7116.pcap'
Ostermann’s tcptrace -- version 6.6.7 -- Thu Nov 4, 2004

574 packets seen, 103 TCP packets traced
elapsed wallclock time: 0:00:00.001808, 317477 pkts/sec analyzed
trace file elapsed time: 0:10:05.833436
TCP connection info:
5 TCP connections traced:
TCP connection 1:
host a: 10.3.9.89:1033
host b: 10.9.8.7:5432
complete conn: yes
last packet: Thu May 3 20:37:31.591572 2012
elapsed time: 0:00:00.059501
total packets: 11
filename: /opt/sandnet/uploads/pcaps/7116.pcap
a->b: b->a:
total packets: 5 total packets: 6
ack pkts sent: 4 ack pkts sent: 6
```

Listing I.2: Example output from tcptrace when run in long output mode
TCP connection 2:
host c: 10.3.9.89:1034
host d: 10.9.8.7:6543
complete conn: no (SYNs: 1) (FINs: 1)
last packet: Thu May 3 20:39:11.794107 2012
elapsed time: 0:01:40.157557
total packets: 71
filename: /opt/sandnet/uploads/pcaps/7116.pcap
c->d: total packets: 71
ack pkts sent: 70
pure acks sent: 68
sack pkts sent: 0
dsack pkts sent: 0
max sack blks/ack: 0
unique bytes sent: 95
actual data pkts: 1
actual data bytes: 95
d->c: total packets: 0
ack pkts sent: 0
pure acks sent: 0
sack pkts sent: 0
dsack pkts sent: 0
max sack blks/ack: 0
unique bytes sent: 0
actual data pkts: 0
actual data bytes: 0
TCP connection 3:
host e: 10.3.9.89:1035
host f: 10.9.8.7:1234

complete conn: yes
last packet: Thu May 3 20:39:12.443994 2012
elapsed time: 0:01:40.639987

total packets: 9
filename: /opt/sandnet/uploads/pcaps/7116.pcap
e->f:
total packets: 5
total packets: 4
ack pkts sent: 4
ack pkts sent: 4
pure acks sent: 2
pure acks sent: 2
sack pkts sent: 0
sack pkts sent: 0
dsack pkts sent: 0
dsack pkts sent: 0
max sack blks/ack: 0
max sack blks/ack: 0
unique bytes sent: 111
unique bytes sent: 0
actual data pkts: 1
actual data pkts: 0
actual data bytes: 111
actual data bytes: 0
rexmt data pkts: 0
rexmt data pkts: 0
rexmt data bytes: 0
rexmt data bytes: 0
zwnd probe pkts: 0
zwnd probe pkts: 0
zwnd probe bytes: 0
zwnd probe bytes: 0
outoforder pkts: 0
outoforder pkts: 0
pushed data pkts: 1
pushed data pkts: 0
SYN/FIN pkts sent: 1/1
SYN/FIN pkts sent: 1/1
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>req sack:</td>
<td>Y</td>
<td>req sack:</td>
<td>Y</td>
</tr>
<tr>
<td>sacks sent:</td>
<td>0</td>
<td>sacks sent:</td>
<td>0</td>
</tr>
<tr>
<td>urgent data pkts:</td>
<td>0 pkts</td>
<td>urgent data pkts:</td>
<td>0 pkts</td>
</tr>
<tr>
<td>urgent data bytes:</td>
<td>0 bytes</td>
<td>urgent data bytes:</td>
<td>0 bytes</td>
</tr>
<tr>
<td>mss requested:</td>
<td>1460 bytes</td>
<td>mss requested:</td>
<td>1460 bytes</td>
</tr>
<tr>
<td>max segm size:</td>
<td>111 bytes</td>
<td>max segm size:</td>
<td>0 bytes</td>
</tr>
<tr>
<td>min segm size:</td>
<td>111 bytes</td>
<td>min segm size:</td>
<td>0 bytes</td>
</tr>
<tr>
<td>avg segm size:</td>
<td>110 bytes</td>
<td>avg segm size:</td>
<td>0 bytes</td>
</tr>
<tr>
<td>max win adv:</td>
<td>64240 bytes</td>
<td>max win adv:</td>
<td>14600 bytes</td>
</tr>
<tr>
<td>min win adv:</td>
<td>64240 bytes</td>
<td>min win adv:</td>
<td>14600 bytes</td>
</tr>
<tr>
<td>zero win adv:</td>
<td>0 times</td>
<td>zero win adv:</td>
<td>0 times</td>
</tr>
<tr>
<td>avg win adv:</td>
<td>64240 bytes</td>
<td>avg win adv:</td>
<td>14600 bytes</td>
</tr>
<tr>
<td>initial window:</td>
<td>111 bytes</td>
<td>initial window:</td>
<td>0 bytes</td>
</tr>
<tr>
<td>ttl stream length:</td>
<td>111 bytes</td>
<td>ttl stream length:</td>
<td>0 bytes</td>
</tr>
<tr>
<td>missed data:</td>
<td>0 bytes</td>
<td>missed data:</td>
<td>0 bytes</td>
</tr>
<tr>
<td>truncated data:</td>
<td>0 bytes</td>
<td>truncated data:</td>
<td>0 bytes</td>
</tr>
<tr>
<td>truncated packets:</td>
<td>0 pkts</td>
<td>truncated packets:</td>
<td>0 pkts</td>
</tr>
<tr>
<td>data xmit time:</td>
<td>0.000 secs</td>
<td>data xmit time:</td>
<td>0.000 secs</td>
</tr>
<tr>
<td>idletime max:</td>
<td>100626.8 ms</td>
<td>idletime max:</td>
<td>100626.2 ms</td>
</tr>
<tr>
<td>throughput:</td>
<td>1 Bps</td>
<td>throughput:</td>
<td>0 Bps</td>
</tr>
</tbody>
</table>

TCP connection 4:

- Host g: 10.3.9.89:1039
- Host h: 10.9.8.7:1234
- Complete conn: yes
- Last packet: Thu May 3 20:40:52.512554 2012
- Elapsed time: 0:01:39.997225
- Total packets: 9
- Filename: /opt/sandnet/uploads/pcaps/7116.pcap

- g->h:
  - Total packets: 5
  - Ack pkts sent: 4
  - Pure acks sent: 2
  - Sack pkts sent: 0
  - Dsack pkts sent: 0
  - Max sack blks/ack: 0
  - Unique bytes sent: 73
  - Actual data pkts: 1
  - Actual data bytes: 73
  - Rexmt data pkts: 0
  - Rexmt data bytes: 0
  - Zwnd probe pkts: 0
  - Zwnd probe bytes: 0
  - Outoforder pkts: 0
  - Pushed data pkts: 1
  - SYN/FIN pkts sent: 1/1

- h->g:
  - Total packets: 4
  - Ack pkts sent: 4
  - Pure acks sent: 2
  - Sack pkts sent: 0
  - Dsack pkts sent: 0
  - Max sack blks/ack: 0
  - Unique bytes sent: 0
  - Actual data pkts: 0
  - Actual data bytes: 0
  - Rexmt data pkts: 0
  - Rexmt data bytes: 0
  - Zwnd probe pkts: 0
  - Zwnd probe bytes: 0
  - Outoforder pkts: 0
  - Pushed data pkts: 0
  - SYN/FIN pkts sent: 1/1
### TCP Connection 5:

- **Host i:** 10.3.9.89:1040
- **Host j:** 190.9.35.199:80
- **Complete Conn.:** No (SYNs: 1) (FINs: 0)
- **First Packet:** Thu May 3 20:39:13.993413 2012
- **Last Packet:** Thu May 3 20:39:23.000361 2012
- **Elapsed Time:** 0:00:09.006947
- **Total Packets:** 3
- **Filename:** /opt/sandnet/uploads/pcaps/7116.pcap

#### i->j:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Packets</td>
<td>3</td>
</tr>
<tr>
<td>Ack Packets Sent</td>
<td>0</td>
</tr>
<tr>
<td>Pure Ack Packets Sent</td>
<td>0</td>
</tr>
<tr>
<td>Sack Packets Sent</td>
<td>0</td>
</tr>
<tr>
<td>Dsack Packets Sent</td>
<td>0</td>
</tr>
<tr>
<td>Max Sack Blks/Ack</td>
<td>0</td>
</tr>
<tr>
<td>Unique Bytes Sent</td>
<td>0</td>
</tr>
<tr>
<td>Actual Data Packets</td>
<td>0</td>
</tr>
<tr>
<td>Actual Data Bytes</td>
<td>0</td>
</tr>
<tr>
<td>Retransmit Data Packets</td>
<td>2</td>
</tr>
<tr>
<td>Retransmit Data Bytes</td>
<td>2</td>
</tr>
<tr>
<td>Zwnd Probe Packets</td>
<td>0</td>
</tr>
<tr>
<td>Zwnd Probe Bytes</td>
<td>0</td>
</tr>
<tr>
<td>Outoforder Packets</td>
<td>0</td>
</tr>
<tr>
<td>Pushed Data Packets</td>
<td>0</td>
</tr>
<tr>
<td>SYN/FIN Packets Sent</td>
<td>1/0</td>
</tr>
<tr>
<td>Req Sack</td>
<td>Y</td>
</tr>
<tr>
<td>Sacks Sent</td>
<td>0</td>
</tr>
<tr>
<td>Urgent Data Packets</td>
<td>0</td>
</tr>
<tr>
<td>Urgent Data Bytes</td>
<td>0</td>
</tr>
<tr>
<td>Mss Requested</td>
<td>1460 bytes</td>
</tr>
<tr>
<td>Max Segm Size</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Min Segm Size</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Avg Segm Size</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Max Win Adv</td>
<td>14600 bytes</td>
</tr>
<tr>
<td>Min Win Adv</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Avg Win Adv</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Initial Window</td>
<td>0 pkts</td>
</tr>
</tbody>
</table>

#### j->i:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Packets</td>
<td>0</td>
</tr>
<tr>
<td>Ack Packets Sent</td>
<td>0</td>
</tr>
<tr>
<td>Pure Ack Packets Sent</td>
<td>0</td>
</tr>
<tr>
<td>Sack Packets Sent</td>
<td>0</td>
</tr>
<tr>
<td>Dsack Packets Sent</td>
<td>0</td>
</tr>
<tr>
<td>Max Sack Blks/Ack</td>
<td>0</td>
</tr>
<tr>
<td>Unique Bytes Sent</td>
<td>0</td>
</tr>
<tr>
<td>Actual Data Packets</td>
<td>0</td>
</tr>
<tr>
<td>Actual Data Bytes</td>
<td>0</td>
</tr>
<tr>
<td>Retransmit Data Packets</td>
<td>0</td>
</tr>
<tr>
<td>Retransmit Data Bytes</td>
<td>0</td>
</tr>
<tr>
<td>Zwnd Probe Packets</td>
<td>0</td>
</tr>
<tr>
<td>Zwnd Probe Bytes</td>
<td>0</td>
</tr>
<tr>
<td>Outoforder Packets</td>
<td>0</td>
</tr>
<tr>
<td>Pushed Data Packets</td>
<td>0</td>
</tr>
<tr>
<td>SYN/FIN Packets Sent</td>
<td>0/0</td>
</tr>
<tr>
<td>Req Sack</td>
<td>N</td>
</tr>
<tr>
<td>Sacks Sent</td>
<td>0</td>
</tr>
<tr>
<td>Urgent Data Packets</td>
<td>0</td>
</tr>
<tr>
<td>Urgent Data Bytes</td>
<td>0</td>
</tr>
<tr>
<td>Mss Requested</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Max Segm Size</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Min Segm Size</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Avg Segm Size</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Max Win Adv</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Min Win Adv</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Avg Win Adv</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Initial Window</td>
<td>0 pkts</td>
</tr>
<tr>
<td>Initial Window</td>
<td>0 pkts</td>
</tr>
<tr>
<td></td>
<td>NA</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>ttl stream length</td>
<td>NA</td>
</tr>
<tr>
<td>missed data</td>
<td>NA</td>
</tr>
<tr>
<td>truncated data</td>
<td>0 bytes</td>
</tr>
<tr>
<td>truncated packets</td>
<td>0 pkts</td>
</tr>
<tr>
<td>data xmit time</td>
<td>0.000 secs</td>
</tr>
<tr>
<td>idletime max</td>
<td>484910.3 ms</td>
</tr>
<tr>
<td>throughput</td>
<td>0 Bps</td>
</tr>
</tbody>
</table>

Listing I.2: Example output from tcptrace when run in long output mode
Appendix J

Zeus (zbot) HTTP Requests for Executables

Listing J.1 shows the three HTTP GET requests for executable files present in MALICIA file ae7e22714babc3babcaada0aa7fbc79b5249b2e9_021033.pcap. These are attempts by the malware to download further executables, which may be other malware or plugins for Zeus. Each request is to a different domain name and uses what appears to be a randomly generated URI path. The clustering techniques used in the analysis framework are capable of grouping these three requests together.

```json
[
  {
    "request_num": 1,
    "upload_id": 11242,
    "session_num": 30,
    "request_protocol": {
      "name": "Http Request",
      "short_name": "http_request",
      "data_len": 197,
      "fields": {
        "headers": [
          {
            "Host": "risparmioassicurativo.net"
          },
          {
            "Accept": "*/*
          },
          {
            "Accept-Encoding": "identity, *;q=0"
          },
          {
            "Connection": "close"
          },
          {
            "User-Agent": "Mozilla/4.0 (compatible; MSIE 5.0; Windows 98)"
          }
        ],
        "uri_path": "/BM7c9uLn/YzHTR2j.exe",
        "version": "1.0",
        "uri_param_values": [],
        "uri_param_names": [],
        "uri": "/BM7c9uLn/YzHTR2j.exe",
        "User-Agent": "Mozilla/4.0 (compatible; MSIE 5.0; Windows 98)",
        "Accept": "*/*
      },
      "Host": "risparmioassicurativo.net"
  }
]```
"data_len": 191,
"fields": {
    "headers": [
        {
            "Host": "brilhanteservice.com.br"
        },
        {
            "Accept": "*/*
        },
        {
            "Accept-Encoding": "identity, *;q=0"
        },
        {
            "Connection": "close"
        },
        {
            "User-Agent": "Mozilla/4.0 (compatible; MSIE 5.0; Windows 98)"
        }
    ],
    "uri_path": "/rQuTTn7k/mkE.exe",
    "version": "1.0",
    "uri_param_values": [],
    "uri_param_names": [],
    "uri": "/rQuTTn7k/mkE.exe",
    "User-Agent": "Mozilla/4.0 (compatible; MSIE 5.0; Windows 98)",
    "Accept": "*/*
    "Host": "brilhanteservice.com.br",
    "Accept-Encoding": "identity, *;q=0",
    "method": "GET",
    "body_len": 0,
    "Connection": "close"
}
}
]

Listing J.1: JSON data returned by a call to endpoint /api/uploads/features/session_requests/11242,30,1;11242,31,1;11242,32,1 (MALICIA file ae7e22714babe3babcaada0aa7fbc79b5249b2e9_021033.pcap)
Appendix K

Cleaman Malware Family - Domains

The qnames of DNS queries made by the Cleaman malware family traffic samples in the MALICIA dataset are listed below.

- 010d3738e3138382e3138312e31380d4070656e74692e636f6d2e747200.lbl8.mailshell.net
- 1.0.0.127.dnsbugtest.1.0.0.127.in-addr.arpa
- 1.0.168.192.in-addr.arpa
- 109.56.235.91.in-addr.arpa
- 17.21.55.65.in-addr.arpa
- 198.35.9.190.in-addr.arpa
- 199.35.9.190.in-addr.arpa
- 222.206.58.201.in-addr.arpa
- 250.255.255.239.in-addr.arpa
- 255.255.3.10.in-addr.arpa
- 3.1.3.10.in-addr.arpa
- 33.218.106.81.in-addr.arpa
- 7.8.9.10.in-addr.arpa
- _kerberos._tcp.Whyteleafe._sites.dc._msdcs.AS.LOCAL
- _ldap._tcp.Default-First-Site-Name._sites.dc._msdcs.schulnetz.local
- android.clients.google.com
- bitcast-b.bitgravity.com
- celebsfunda.com
- espanol.victoriassecret.com
- filme-porno99.info
- htc-mobile.mywtv.cn
- hylesrv01.hylearc.local
- intsec.quickheal.com
- media.victoriassecret.com
• mtalk.google.com
• nist1-la.ustiming.org
• nist1-pa.ustiming.org
• ntp.a1.ind.br
• ntp.a1.ind.br.a1.ind.br
• ntp.a1.ind.br.a1cwb
• pool.ntp.org
• satsuki-sv2.satsuki.local
• sdl.360safe.com
• sdup.360.cn
• sdupm.360.cn
• secure.victoriassecret.com
• sso.accaglobal.com
• stat.sd.360.cn
• time-h.nist.gov
• time.windows.com
• tracker.pow7.com
• twimg0-a.akamaihd.net
• vigaman.com
• webres1.qheal.ctmail.com
• webres2.qheal.ctmail.com
• webres4.qheal.ctmail.com
• webres5.qheal.ctmail.com
• wireless.mapbar.com
• www.mastitorrents.com
• www.paperpk.com
• www25.victoriassecret.com
Appendix L

Cridex Malware Family - Domains

The qnames of DNS queries made by the Cridex malware family traffic samples in the MALICIA dataset are listed below.

- 1.0.168.192.in-addr.arpa
- 122.218.185.146.in-addr.arpa
- 138.9.65.120.IN-ADDR.ARPA
- 142.4.201.91.in-addr.arpa
- 2.2.2.2.in-addr.arpa
- 209.1.42.222.IN-ADDR.ARPA
- 226.157.202.89.IN-ADDR.ARPA
- 250.255.255.239.in-addr.arpa
- 255.255.3.10.in-addr.arpa
- 3.1.3.10.in-addr.arpa
- 7.8.9.10.in-addr.arpa
- 76.62.31.176.in-addr.arpa
- acamacookldaureglbh.ru
- aecgrgbjgaofrilwyg.ru
- agtlyyprtrzsfugnwoqb.ru
- anidgewelndzueoyl.ru
- aopltfxjzppylfhau.ru
- berezavmorda.ru
- bgadkfnegwgtkzwqzv.ru
- boofsbsmmqpcpmgcd.ru
- cerberzorberhu.ru
- ciasamkvnmavtnxiko.ru
- ckpmgcdlsidwsdnolw.ru
- cnkjufwagtlyyprtrzfr.ru
- ddgrctkhjkqhomemap.ru
nalezivmordu.ru
ngdvmtwodjjuovsnfj.ru
nolwzyzsqkhjkqhome.ru
muyiqrrkriwexgsism.ru
oaztysxycnkjufw.ru
palktkjaljob.ru
practicalcex.in
practicalcex.ru
praktikaljox.ru
pylfhauvqghelyqwt.ru
qgqpekpmgesdlewsmru
qrrkriwexgsismxsvd.ru
qwtsunaoaztysxyc.ru
rdqdykelicpqrqphcm.ru
rgglwvyzeveijgnvm.ru
scanforsecurityholes.ru
secuirteecheckme.ru
securitycheckme.ru
silvercerberhu.ru
sqkhjkqhomecmapiuig.ru
testme2securejtj.ru
testme4secureetee.ru
testnosecurity.ru
tfylzsspyflhauvqig.ru
time.windows.com
trzfugxwqboofhsbh.ru
tytsxycnkjufwagt.ru
ugnwoqboofbsbhsgq.ru
uijgtanfznuyiqrk.ru
vaopxjiaphevkcqdo.ru
vhaygrpumrlmynmcwhk.ru
vqghelyqwtfusomoz.ru
woqboofbhqgqoeckp.ru
www.google.com
The HTTP Host header values used by the Cridex malware family traffic samples in the MALICIA dataset are listed below.

- wxyccnkjufwagtlyyp.ru
- xsdvyfuaopltfxjzsp.ru
- yfuaopltfxjzsppylf.ru
- yhbyqwmrtqxvmpryon.ru
- zorberzorberzu.ru
- zsppylfauvqjghely.ru
- zyzsqkhjkqhoncmapi.ru

- 103.6.237.9:8080
- 103.6.238.9:8080
- 108.171.245.130:8080
- 108.171.246.130:8080
- 108.171.247.194:8080
- 108.171.254.66:8080
- 109.230.229.250:8080
- 109.230.229.70:8080
- 110.234.150.163:8080
- 123.49.61.59:8080
- 125.19.103.198:8080
- 128.2.172.202:8080
- 140.123.101.4:8080
- 148.208.216.70:8080
- 155.98.65.40:8080
- 163.23.107.65:8080
- 164.15.21.2:8080
- 173.192.229.36:8080
- 173.201.177.77:8080
- 173.203.102.204:8080
- 173.203.96.79:8080
- 173.224.208.60:8080
- 173.224.208.60:8080
- 173.224.215.130:8080
- 31.131.31.10:8080
- 31.17.189.212:8080
- 38.99.150.69:8080
- 41.168.5.140:8080
- 50.22.102.132:8080
- 50.22.94.96:8080
- 58.68.2.214:8080
- 59.90.221.6:8080
- 61.7.235.35:8080
- 62.28.244.251:8080
- 64.120.193.112:8080
- 64.150.187.72:8080
- 64.76.19.236:8080
- 64.85.53.168:8080
- 64.94.164.18:8080
- 66.242.19.36:8080
- 68.178.206.179:8080
- 69.64.89.82:8080
- 72.167.253.106:8080
- 72.18.203.140:8080
- 74.117.107.25:8080
- 74.117.58.80:8080
- 74.117.59.55:8080
- 74.117.61.66:8080
- 74.207.237.170:8080
- 74.63.229.10:8080
- 74.86.113.66:8080
- 77.58.193.43:8080
- 78.28.120.32:8080
- 81.93.250.157:8080
- 82.165.147.190:8080
- 83.143.134.23:8080
- 83.238.208.55:8080
- 84.22.100.108:8080
• 85.214.204.32:8080
• 85.226.179.185:8080
• 85.25.147.73:8080
• 87.120.41.155:8080
• 87.204.199.100:8080
• 87.229.26.138:8080
• 88.119.156.20:8080
• 89.111.176.87:8080
• 89.221.242.217:8080
• 89.97.55.33:8080
• 91.121.103.143:8080
• 91.228.154.199:8080
• 94.20.30.91:8080
• 94.73.129.120:8080
• 95.142.167.193:8080
• 97.74.113.229:8080
• 97.74.75.172:8080
• berezavmorda.ru
• cerberzorberhu.ru
• derezivmorda.ru
• dl.javafx.com
• internetsexcuritee4dummies.ru
• javadl-esd.sun.com
• krjffgzzzooooem.ru
• nalezivmordu.in
• nalezivmordu.ru
• palktikaljob.ru
• prakticalcex.in
• prakticalcex.ru
• praktikaljox.ru
• scanforsecurityholes.ru
• securiteechckme.ru
• securytycheckme.ru
• silvercerberhu.ru
• testme2securejtej.ru
• testme4secureetee.ru
• testnosecurity.ru
• uigjtnafznuyiqrrk.ru:8080
• www.google.com
• zorberzorberzu.ru
Appendix M

Cutwail Malware Family -
Domains

The qnames of DNS queries made by the Cutwail malware family traffic samples in the MALICIA dataset are listed below.

- 1.0.168.192.in-addr.arpa
- 1.gc.1e400.net
- 12.0.76.86.in-addr.arpa
- 145.248.163.69.in-addr.arpa
- 145.250.163.69.in-addr.arpa
- 150.57.4.46.in-addr.arpa
- 17.33.194.173.in-addr.arpa
- 206.23.82.80.in-addr.arpa
- 226.49.46.78.in-addr.arpa
- 250.255.55.239.in-addr.arpa
- 255.255.3.10.in-addr.arpa
- 3.1.3.10.in-addr.arpa
- 6.212.7.212.in-addr.arpa
- 6.6.35.31.in-addr.arpa
- 7.8.9.10.in-addr.arpa
- cache.trillinux.org
- cache.wru.pl
- cache2.bazookanetworks.com
- dogma.cloud.bishopston.net
- gwc.lame.net
- gwc.marksieklucki.com
- gwc2.wodi.org
- gweb.4octets.co.uk
- jayl.de
The HTTP Host header values used by the Cutwail malware family traffic samples in the MALICIA dataset are listed below.

- 1.gc.1e400.net
- 10.9.8.7:1234
- 10.9.8.7:5432
- 10.9.8.7:6543
- 192.168.0.2:8000
- cache.trillinux.org
- cache.wru.pl
- cache2.bazookanetworks.com
- gwc.marksieklucki.com
- gwc2.wodi.org
- jayl.de
- tenafly5k.com
Appendix N

Public IP Addresses Used for DNS Queries

The list of public IP addresses from the MALICIA dataset that are used for DNS queries is listed below. Malware bypasses the local DNS server to use these.

- 8.8.8.8
- 83.133.123.20
- 94.242.250.64
- 66.85.130.234
- 92.241.163.23
- 194.165.17.3
- 91.242.217.247
- 109.230.217.44
- 178.162.190.125
Appendix O

HTTP Headers

The HTTP request header names present in the MALICIA traffic captures are listed in Table O.1.

<table>
<thead>
<tr>
<th>HTTP header</th>
<th>Number of samples</th>
<th>Individual occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection</td>
<td>16708</td>
<td>116373</td>
</tr>
<tr>
<td>Host</td>
<td>16708</td>
<td>118035</td>
</tr>
<tr>
<td>User-Agent</td>
<td>14956</td>
<td>107498</td>
</tr>
<tr>
<td>Accept-Encoding</td>
<td>14747</td>
<td>82489</td>
</tr>
<tr>
<td>Content-Length</td>
<td>14726</td>
<td>74185</td>
</tr>
<tr>
<td>Content-Type</td>
<td>14699</td>
<td>66655</td>
</tr>
<tr>
<td>Cache-Control</td>
<td>6848</td>
<td>18007</td>
</tr>
<tr>
<td>Accept</td>
<td>5666</td>
<td>70643</td>
</tr>
<tr>
<td>Pragma</td>
<td>4002</td>
<td>5599</td>
</tr>
<tr>
<td>Referer</td>
<td>3251</td>
<td>4438</td>
</tr>
<tr>
<td>Content-Encoding</td>
<td>1408</td>
<td>51784</td>
</tr>
<tr>
<td>UA-Java-Version</td>
<td>1129</td>
<td>1890</td>
</tr>
<tr>
<td>accept-encoding</td>
<td>1129</td>
<td>1890</td>
</tr>
<tr>
<td>Cookie</td>
<td>832</td>
<td>7070</td>
</tr>
<tr>
<td>If-Modified-Since</td>
<td>811</td>
<td>1572</td>
</tr>
<tr>
<td>Accept-Language</td>
<td>530</td>
<td>2338</td>
</tr>
<tr>
<td>UA-CPU</td>
<td>359</td>
<td>360</td>
</tr>
<tr>
<td>Authorization</td>
<td>4</td>
<td>688</td>
</tr>
<tr>
<td>Proxy-Connection</td>
<td>4</td>
<td>28</td>
</tr>
<tr>
<td>X-Mining-Extensions</td>
<td>4</td>
<td>688</td>
</tr>
<tr>
<td>Content-Transfer-Encoding</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Content-length</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Content-type</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Keep-Alive</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Table O.1: HTTP request header names, the number of MALICIA samples they appear in (out of 16967) and the count of occurrences across all MALICIA samples.

The HTTP response header names present in the MALICIA traffic captures are listed in Table O.2.
<table>
<thead>
<tr>
<th>HTTP header</th>
<th>Number of samples</th>
<th>Individual occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>15666</td>
<td>112008</td>
</tr>
<tr>
<td>Date</td>
<td>15652</td>
<td>111996</td>
</tr>
<tr>
<td>Content-type</td>
<td>15640</td>
<td>15641</td>
</tr>
<tr>
<td>Content-Length</td>
<td>13586</td>
<td>102154</td>
</tr>
<tr>
<td>Content-Type</td>
<td>9739</td>
<td>88969</td>
</tr>
<tr>
<td>Connection</td>
<td>9737</td>
<td>88942</td>
</tr>
<tr>
<td>Set-Cookie</td>
<td>9612</td>
<td>89192</td>
</tr>
<tr>
<td>Cache-Control</td>
<td>9559</td>
<td>88579</td>
</tr>
<tr>
<td>Expires</td>
<td>9553</td>
<td>88573</td>
</tr>
<tr>
<td>Accept-Ranges</td>
<td>9426</td>
<td>88420</td>
</tr>
<tr>
<td>Vary</td>
<td>9408</td>
<td>88409</td>
</tr>
<tr>
<td>Keep-Alive</td>
<td>9394</td>
<td>88383</td>
</tr>
<tr>
<td>Success</td>
<td>7389</td>
<td>7389</td>
</tr>
<tr>
<td>X-Frame-Options</td>
<td>146</td>
<td>159</td>
</tr>
<tr>
<td>X-Powered-By</td>
<td>97</td>
<td>127</td>
</tr>
<tr>
<td>P3P</td>
<td>94</td>
<td>118</td>
</tr>
<tr>
<td>X-XSS-Protection</td>
<td>92</td>
<td>100</td>
</tr>
<tr>
<td>Transfer-Encoding</td>
<td>72</td>
<td>92</td>
</tr>
<tr>
<td>P3p</td>
<td>54</td>
<td>59</td>
</tr>
<tr>
<td>X-Xss-Protection</td>
<td>54</td>
<td>59</td>
</tr>
<tr>
<td>Last-Modified</td>
<td>37</td>
<td>50</td>
</tr>
<tr>
<td>Location</td>
<td>17</td>
<td>30</td>
</tr>
<tr>
<td>ETag</td>
<td>12</td>
<td>21</td>
</tr>
<tr>
<td>X-Sinkholed-Domain</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Pragma</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>Content-Language</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>X-Pingback</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Accept-ranges</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Cache-control</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Content-length</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Content-Disposition</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Content-Transfer-Encoding</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>VTag</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>X-AspNet-Version</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>MicrosoftOfficeWebServer</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X-Pad</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X-Remote-IP</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

Table O.2: HTTP response header names, the number of MALICIA samples they appear in (out of 16967) and the count of occurrences across all MALICIA samples.
## HTTP User-Agents

The HTTP User-Agents present in the MALICIA traffic captures are listed in Table P.1.

<table>
<thead>
<tr>
<th>User-Agent</th>
<th>Number of samples</th>
<th>Individual occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>JeppoWeb</td>
<td>14526</td>
<td>14526</td>
</tr>
<tr>
<td>Python-urlib/2.6</td>
<td>12511</td>
<td>12511</td>
</tr>
<tr>
<td>Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; GTB0.0; .NET CLR 1.1.4322)</td>
<td>3605</td>
<td>4927</td>
</tr>
<tr>
<td></td>
<td>2858</td>
<td>3684</td>
</tr>
<tr>
<td>Mozilla/4.0 (compatible; MSIE 5.0; Windows 98)</td>
<td>1507</td>
<td>55042</td>
</tr>
<tr>
<td>JNLP/6.0 javaws/1.6.0_14 (b08) Java/1.6.0_14</td>
<td>1129</td>
<td>1890</td>
</tr>
<tr>
<td>jupdate</td>
<td>972</td>
<td>1297</td>
</tr>
<tr>
<td>Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)</td>
<td>422</td>
<td>2163</td>
</tr>
<tr>
<td>Opera/9 (Windows NT 5.1; ; x86)</td>
<td>373</td>
<td>3054</td>
</tr>
<tr>
<td>Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)</td>
<td>157</td>
<td>235</td>
</tr>
<tr>
<td>Mozilla/5.0 (Windows; U; MSIE 7.0; Windows NT 6.0; en-US)</td>
<td>93</td>
<td>6912</td>
</tr>
<tr>
<td>Opera/6 (Windows NT 5.1; ; LangID=409; x86)</td>
<td>51</td>
<td>155</td>
</tr>
<tr>
<td>Mozilla/4.0 (compatible; MSIE 8.0; Trident/4.0; .NET CLR 2.0.50727; .NET CLR 1.1.4322; .NET CLR 3.0.04506.590; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)</td>
<td>31</td>
<td>54</td>
</tr>
<tr>
<td>Mozilla/5.0 (Windows NT 6.1; WOW64; rv:2.0b8pre) Gecko/20101114 Firefox/4.0b8pre</td>
<td>23</td>
<td>46</td>
</tr>
<tr>
<td>Mozilla/5.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/5.0);{b:2600cc:INT-3360b:09)</td>
<td>13</td>
<td>23</td>
</tr>
<tr>
<td>Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)</td>
<td>9</td>
<td>44</td>
</tr>
<tr>
<td>InetURL/1.0</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>Internet Explorer</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Mozilla/4.0 (compatible; Synapse)</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.11 (KHTML, like Gecko) Chrome/23.0.1271.97 Safari/537.11</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Shareaza</td>
<td>2</td>
<td>179</td>
</tr>
<tr>
<td>User-Agent</td>
<td>Number of samples</td>
<td>Individual occurrences</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>-------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>TRIAL.7875768FDADBB1FC</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Ufasoft bitcoin-miner/0.28 (Windows NT XP 5.1.2600 Service Pack 3)</td>
<td>2</td>
<td>334</td>
</tr>
<tr>
<td>Ufasoft bitcoin-miner/0.30 (Windows NT XP 5.1.2600 Service Pack 3)</td>
<td>2</td>
<td>354</td>
</tr>
<tr>
<td>CA 0.0.0.2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1)</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0)</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Mozilla/5.0 (Windows NT 6.1; WOW64; rv:12.0) Gecko/20100101 Firefox/12.0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Mozilla/5.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/5.0);(b:2600;c:INT-6760;1:09)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>WINXPSP3.7875768F8B00CED2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>al</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

Table P.1: HTTP User-Agents, the number of MALICIA samples they appear in (out of 16967) and the count of occurrences across all MALICIA samples. The top two belong to the MALICIA capture framework itself.
## Appendix Q

### HTTP Servers

The HTTP response Server header values present in the MALICIA traffic captures are listed in Table Q.1.

<table>
<thead>
<tr>
<th>Server</th>
<th>Number of samples</th>
<th>Individual occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleHTTPWithUpload/0.1 Python/2.7.2</td>
<td>10329</td>
<td>15009</td>
</tr>
<tr>
<td>Apache</td>
<td>9419</td>
<td>88435</td>
</tr>
<tr>
<td>SimpleHTTPWithUpload/0.1 Python/2.7.3</td>
<td>3003</td>
<td>5629</td>
</tr>
<tr>
<td>BaseHTTP/0.3 Python/2.7.2</td>
<td>2181</td>
<td>2182</td>
</tr>
<tr>
<td>gws</td>
<td>146</td>
<td>159</td>
</tr>
<tr>
<td>SimpleHTTPWithUpload/0.1 Python/2.7.4</td>
<td>127</td>
<td>210</td>
</tr>
<tr>
<td>nginx</td>
<td>76</td>
<td>77</td>
</tr>
<tr>
<td>nginx/0.5.33</td>
<td>61</td>
<td>181</td>
</tr>
<tr>
<td>nginx/1.0.12</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Microsoft-IIS/7.0</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>SimpleHTTPWithUpContent-Length: 298</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Microsoft-IIS/6.0</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Apache/2.2.21 (Unix) mod_ssl/2.2.21 OpenSSL/0.9.8m DAV/2 mod_auth_passthrough/2.1 mod_bwlimited/1.4 FrontPage/5.0.2.2635</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Apache/2.2.3 (CentOS)</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Apache/2.2.9 (Debian) mod_python/3.3.1 Python/2.5.2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Apache/2.2.16 (Debian)</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Apache/2.2.9 (Debian) PHP/5.2.6-1+lenny9 with Suhosin-Patch mod_ssl/2.2.9 OpenSSL/0.9.8g</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>WebServerX</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Microsoft-IIS/7.5</td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td>SimpleHTTPWithUpContent-type: text/html Content-Length: 298</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Apache/1.3.31 (Unix) mod_jk/1.2.5 PHP/5.2.17 FrontPage/5.0.2.2634 mod_fastcgi/2.4.2 mod_throttle/3.1.2 mod_ssl/2.8.18 OpenSSL/0.9.7d</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Apache/2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Apache/2.0.53 (Fedora)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Apache/2.2.15 (Linux/SUSE)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Apache/2.2.16 (Debian) PHP/5.3.3-7+squeeze3 with Suhosin-Patch mod_ssl/2.2.16 OpenSSL/0.9.8o mod_perl/2.0.4 Perl/v5.10.1</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>Apache/2.2.16 (Fedora)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Apache/2.2.21</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Table Q.1: HTTP Server header values, the number of MALICIA samples they appear in (out of 16967) and the count of occurrences across all MALICIA samples. Several of the Servers appear to be used by the MALICIA capture framework itself (particularly those starting “SimpleHTTP-WithUpload/0.1”)

<table>
<thead>
<tr>
<th>Server</th>
<th>Number of samples</th>
<th>Individual occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache/2.2.22 (Unix)</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>OpenSSL/0.9.8e-fips-rhel5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod_auth_passthrough/2.1 mod_bwlimited/1.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FrontPage/5.0.2.2635</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IdeaWebServer/v0.70</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LiteSpeed</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SimpleHTTPWithUpDate: Wed, 03 Oct 2012</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1Content-Length: 298</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nginx/0.8.54</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>nginx/1.0.4</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Appendix R

HTTP Response Content-Types

The HTTP response Content-Type header values present in the MALICIA traffic captures are listed in Table R.1.

<table>
<thead>
<tr>
<th>Content-Type</th>
<th>Number of samples</th>
<th>Individual occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>text/html</td>
<td>9557</td>
<td>88672</td>
</tr>
<tr>
<td>text/html; charset=UTF-8</td>
<td>152</td>
<td>173</td>
</tr>
<tr>
<td>application/octet-stream</td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td>text/html; charset=iso-8859-1</td>
<td>25</td>
<td>36</td>
</tr>
<tr>
<td>text/html; charset=utf-8</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>application/x-httpd-php</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>text/html; charset=ISO-8859-1</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>application/octet-stream; charset=ISO-8859-1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>application/x-msdownload</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>text/plain; charset=ISO-8859-1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>text/plain; charset=UTF-8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>application/gzip</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>text/plain</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Table R.1: HTTP response Content-Type header values, the number of MALICIA samples they appear in (out of 16967) and the count of occurrences across all MALICIA samples.
Appendix S

Fields Extracted from the Malicia Dataset

The fields listed below are extracted by the analysis framework from the MALICIA dataset. They are retrieved by a call to the endpoint /api/uploads/⟨upload_ids⟩/fields/. These are all the fields that are identified by the packet and session feature extraction code when run over all pcaps in the MALICIA dataset.

- packet.bootp.chaddr
- packet.bootp.ciaddr
- packet.bootp.file
- packet.bootp.flags
- packet.bootp.giaddr
- packet.bootp.hlen
- packet.bootp.hops
- packet.bootp.htype
- packet.bootp.op
- packet.bootp.options
- packet.bootp.secs
- packet.bootp.siaddr
- packet.bootp.sname
- packet.bootp.xid
- packet.bootp.yiaddr
- packet.dhcp.options.options
- packet.dns.aa
- packet.dns.an
- packet.dns.anccount
- packet.dns.an-r_a_record_ttl
- packet.dns.an-rclass
- packet.dns.an-rdata
• packet.dns.an-rlen
• packet.dns.an-rrname
• packet.dns.an-rttl
• packet.dns.an-rtype
• packet.dns.ar
• packet.dns.arcount
• packet.dns.ar-r_a_record_ttl
• packet.dns.ar-rcl
• packet.dns.ar-rdata
• packet.dns.ar-rlen
• packet.dns.ar-rrname
• packet.dns.ar-rttl
• packet.dns.ar-rtype
• packet.dns.id
• packet.dns.ns
• packet.dns.nscount
• packet.dns.ns-r_a_record_ttl
• packet.dns.ns-rcl
• packet.dns.ns-rdata
• packet.dns.ns-rlen
• packet.dns.ns-rrname
• packet.dns.ns-rttl
• packet.dns.ns-rtype
• packet.dns.opcode
• packet.dns.qd
• packet.dns.qdcount
• packet.dns.qd-qclass
• packet.dns.qd-qname
• packet.dns.qd-qtype
• packet.dns.qr
• packet.dns.ra
• packet.dns.rcode
• packet.dns.rd
• packet.dns.tc
- packet.dns.z
- packet.ethernet.dst
- packet.ethernet.src
- packet.ethernet.type
- packet.icmp.addr
d
- packet.icmp.addr
- mask
- packet.icmp.chksum
- packet.icmp.code
- packet.icmp.gw
- packet.icmp.id
- packet.icmp_in.icmp.addr
- mask
- packet.icmp_in.icmp.chksum
- packet.icmp_in.icmp.code
- packet.icmp_in.icmp.gw
- packet.icmp_in.icmp.id
- packet.icmp_in.icmp.ptr
- packet.icmp_in.icmp.reserved
- packet.icmp_in.icmp.seq
- packet.icmp_in.icmp.ts
- ori
- packet.icmp_in.icmp.ts
- rx
- packet.icmp_in.icmp.ts
- tx
- packet.icmp_in.icmp.type
- packet.icmp_in.icmp.unused
- packet.icmp.ptr
- packet.icmp_reserved
- packet.icmp_seq
- packet.icmp.ts
- ori
- packet.icmp.ts
- rx
- packet.icmp.ts
- tx
- packet.icmp.type
- packet.icmp.unused
- packet.icmpv6.destination_unreachable.cksum
- packet.icmpv6.destination_unreachable.code
- packet.icmpv6.destination_unreachable.type
- packet.icmpv6.destination_unreachable.unused
- packet.icmpv6.echo_reply.cksum
- packet.icmpv6.echo_reply.code
- packet.icmpv6.echo_reply.id
- packet.icmpv6.echo_reply.seq
- packet.icmpv6.echo_reply.type
- packet.icmpv6.echo_request.cksum
- packet.icmpv6.echo_request.code
- packet.icmpv6.echo_request.data
- packet.icmpv6.echo_request.id
- packet.icmpv6.echo_request.seq
- packet.icmpv6.echo_request.type
- packet.icmpv6.time_exceeded.cksum
- packet.icmpv6.time_exceeded.code
- packet.icmpv6.time_exceeded.type
- packet.icmpv6.time_exceeded.unused
- packet.ip.chksum
- packet.ip.dst
- packet.ip.flags
- packet.ip.frag
- packet.ip.id
- packet.ip.ihl
- packet.ip_in_icmp.chksum
- packet.ip_in_icmp.dst
- packet.ip_in_icmp.flags
- packet.ip_in_icmp.frag
- packet.ip_in_icmp.id
- packet.ip_in_icmp.ihl
- packet.ip_in_icmp.len
- packet.ip_in_icmp.options
- packet.ip_in_icmp.proto
- packet.ip_in_icmp.src
- packet.ip_in_icmp.tos
- packet.ip_in_icmp.ttl
- packet.ip_in_icmp.version
- packet.ip.len
- packet.ip.options
- packet.ip.proto
- packet.ip.src
- packet.ip.tos
- packet.ip.ttl
- packet.ipv6.dst
- packet.ipv6_extension_header_.hop-by-hop_options_header.autopad
- packet.ipv6_extension_header_.hop-by-hop_options_header.len
- packet.ipv6_extension_header_.hop-by-hop_options_header.nh
- packet.ipv6_extension_header_.hop-by-hop_options_header.options
- packet.ipv6.fl
- packet.ipv6.hlim
- packet.ipv6_in icmpv6.dst
- packet.ipv6_in icmpv6.fl
- packet.ipv6_in icmpv6.hlim
- packet.ipv6_in icmpv6.nh
- packet.ipv6_in icmpv6.plen
- packet.ipv6_in icmpv6.src
- packet.ipv6_in icmpv6.tc
- packet.ipv6_in icmpv6.version
- packet.ipv6.nh
- packet.ipv6.plen
- packet.ipv6.src
- packet.ipv6.tc
- packet.ipv6.version
- packet.ip.version
- packet.link_local_multicast_node_resolution_.query.ancount
- packet.link_local_multicast_node_resolution_.query.an-r.a_record_ttl
- packet.link_local_multicast_node_resolution_.query.an-rclass
- packet.link_local_multicast_node_resolution_.query.an-rdata
- packet.link_local_multicast_node_resolution_.query.an-rdlen
- packet.link_local_multicast_node_resolution_.query.an-rrname
- packet.link_local_multicast_node_resolution_.query.an-rttl

Page 165 of 179
• packet.link_local_multicast_node_resolution._query.an-rtype
• packet.link_local_multicast_node_resolution._query.ar
• packet.link_local_multicast_node_resolution._query.arcount
• packet.link_local_multicast_node_resolution._query.c
• packet.link_local_multicast_node_resolution._query.id
• packet.link_local_multicast_node_resolution._query.ns
• packet.link_local_multicast_node_resolution._query.nscount
• packet.link_local_multicast_node_resolution._query.opcode
• packet.link_local_multicast_node_resolution._query.qdcount
• packet.link_local_multicast_node_resolution._query.qd-qclass
• packet.link_local_multicast_node_resolution._query.qd-qname
• packet.link_local_multicast_node_resolution._query.qd-qttype
• packet.link_local_multicast_node_resolution._query.qr
• packet.link_local_multicast_node_resolution._query.rcode
• packet.link_local_multicast_node_resolution._query.tc
• packet.link_local_multicast_node_resolution._query.z
• packet.mgcp.verb
• packet.nbns_query_request.ANCOUNT
• packet.nbns_query_request.ARCOUNT
• packet.nbns_query_request.FLAGS
• packet.nbns_query_request.NAME_TRN_ID
• packet.nbns_query_request.NSCOUNT
• packet.nbns_query_request.NULL
• packet.nbns_query_request.QDCOUNT
• packet.nbns_query_request.QUESTION_CLASS
• packet.nbns_query_request.QUESTION_NAME
• packet.nbns_query_request.QUESTION_TYPE
• packet.nbns_query_request.SUFFIX
• packet.nbt_datagram_packet.DestinationName
• packet.nbt_datagram_packet.Flags
• packet.nbt_datagram_packet.ID
• packet.nbt_datagram_packet.Length
• packet.nbt_datagram_packet.NULL
• packet.nbt_datagram_packet.Offset
• packet.nbt.datagram.packet.SourceIP
• packet.nbt.datagram.packet.SourceName
• packet.nbt.datagram.packet.SourcePort
• packet.nbt.datagram.packet.SUFFIX1
• packet.nbt.datagram.packet.SUFFIX2
• packet.nbt.datagram.packet.Type
• packet.ntp.delay
• packet.ntp.dispersion
• packet.ntp.id
• packet.ntp.leap
• packet.ntp.mode
• packet.ntp.orig
• packet.ntp.poll
• packet.ntp.precision
• packet.ntp.recv
• packet.ntp.ref
• packet.ntp.sent
• packet.ntp.stratum
• packet.ntp.version
• packet.padding.load
• packet.protocols
• packet.sebek.header.counter
• packet.sebek.header.magic
• packet.sebek.header.time.sec
• packet.sebek.header.time.usec
• packet.sebek.header.type
• packet.sebek.header.version
• packet.sebek.v3.command
• packet.sebek.v3.data
• packet.sebek.v3.data.length
• packet.sebek.v3.fd
• packet.sebek.v3.inode
• packet.sebek.v3.parent.pid
• packet.sebek.v3.pid
• packet.sebek_v3.uid
• packet.skinny.len
• packet.skinny.msg
• packet.skinny.res
• packet.snmp.community
• packet.snmp.PDU
• packet.snmp.version
• packet.tcp.ack
• packet.tcp.chksum
• packet.tcp.data_len
• packet.tcp.dataofs
• packet.tcp.dport
• packet.tcp.flags
• packet.tcp_in_icmp.ack
• packet.tcp_in_icmp.chksum
• packet.tcp_in_icmp.dataofs
• packet.tcp_in_icmp.dport
• packet.tcp_in_icmp.flags
• packet.tcp_in_icmp.options
• packet.tcp_in_icmp.reserved
• packet.tcp_in_icmp.seq
• packet.tcp_in_icmp.sport
• packet.tcp_in_icmp.urgptr
• packet.tcp_in_icmp.window
• packet.tcp.options
• packet.tcpreserved
• packet.tcp.seq
• packet.tcp.sport
• packet.tcp.urgptr
• packet.tcp.window
• packet.udp.chksum
• packet.udp.data_len
• packet.udp.dns_server
• packet.udp.dport
- packet.udp in icmp.chksum
- packet.udp in icmp.dport
- packet.udp in icmp.len
- packet.udp in icmp.sport
- packet.udp.len
- packet.udp.sport
- session.data_len
- session.dst_ip
- session.dst_port
- session.fin_closed
- session.protocols
- session.req_or_resp.http.request.Accept
- session.req_or_resp.http.request.accept-encoding
- session.req_or_resp.http.request.Accept-Encoding
- session.req_or_resp.http.request.Accept-Language
- session.req_or_resp.http.request.Authorization
- session.req_or_resp.http.request.body_len
- session.req_or_resp.http.request.Cache-Control
- session.req_or_resp.http.request.Connection
- session.req_or_resp.http.request.Content-Encoding
- session.req_or_resp.http.request.Content-length
- session.req_or_resp.http.request.Content-Length
- session.req_or_resp.http.request.Content-Transfer-Encoding
- session.req_or_resp.http.request.Content-Type
- session.req_or_resp.http.request.Cookie
- session.req_or_resp.http.request.headers
- session.req_or_resp.http.request.Host
- session.req_or_resp.http.request.If-Modified-Since
- session.req_or_resp.http.request.Keep-Alive
- session.req_or_resp.http.request.method
- session.req_or_resp.http.requestPragma
- session.req_or_resp.http.request.Proxy-Connection
- session.req_or_resp.http.request.Referer
• session.req_or_resp.http_request.UA-CPU
• session.req_or_resp.http_request.UA-Java-Version
• session.req_or_resp.http_request.uri
• session.req_or_resp.http_request.uri_param_names
• session.req_or_resp.http_request.uri_param_values
• session.req_or_resp.http_request.uri_path
• session.req_or_resp.http_request.User-Agent
• session.req_or_resp.http_request.version
• session.req_or_resp.http_request.X-Mining-Extensions
• session.req_or_resp.http_response.Accept-ranges
• session.req_or_resp.http_response.Accept-Ranges
• session.req_or_resp.http_response.body_len
• session.req_or_resp.http_response.Cache-control
• session.req_or_resp.http_response.Cache-Control
• session.req_or_resp.http_response.Connection
• session.req_or_resp.http_response.Content-Disposition
• session.req_or_resp.http_response.Content-Language
• session.req_or_resp.http_response.Content-length
• session.req_or_resp.http_response.Content-Length
• session.req_or_resp.http_response.Content-Transfer-Encoding
• session.req_or_resp.http_response.Content-type
• session.req_or_resp.http_response.Content-Type
• session.req_or_resp.http_response.Date
• session.req_or_resp.http_response.ETag
• session.req_or_resp.http_response.Expires
• session.req_or_resp.http_response.headers
• session.req_or_resp.http_response.Keep-Alive
• session.req_or_resp.http_response.Last-Modified
• session.req_or_resp.http_response.Location
• session.req_or_resp.http_response.MicrosoftOfficeWebServer
• session.req_or_resp.http_response.P3p
• session.req_or_resp.http_response.P3P
• session.req_or_resp.http_response.Pragma
• session.req_or_resp.http_response.reason_phrase
• session.request.http_request.Host
• session.request.http_request.If-Modified-Since
• session.request.http_request.Keep-Alive
• session.request.http_request.method
• session.request.http_request.Pragma
• session.request.http_request.Proxy-Connection
• session.request.http_request.Referer
• session.request.http_request.UA-CPU
• session.request.http_request.UA-Java-Version
• session.request.http_request.uri
• session.request.http_request.uri_param_names
• session.request.http_request.uri_param_values
• session.request.http_request.uri_path
• session.request.http_request.User-Agent
• session.request.http_request.version
• session.request.http_request.X-Mining-Extensions
• session.request_num_packets
• session.request_protocols
• session.response_data_len
• session.response.http_response.Accept-ranges
• session.response.http_response.Accept-Ranges
• session.response.http_response.body_len
• session.response.http_response.Cache-control
• session.response.http_response.Cache-Control
• session.response.http_response.Connection
• session.response.http_response.Content-Disposition
• session.response.http_response.Content-Language
• session.response.http_response.Content-length
• session.response.http_response.Content-Length
• session.response.http_response.Content-Transfer-Encoding
• session.response.http_response.Content-type
• session.response.http_response.Content-Type
• session.response.http_response.Date
• session.response.http_response.ETag
- session.response.http_response.Expires
- session.response.http_response.headers
- session.response.http_response.Last-Modified
- session.response.http_response.Location
- session.response.http_response.MicrosoftOfficeWebServer
- session.response.http_response.P3p
- session.response.http_response.P3P
- session.response.http_response.Pragma
- session.response.http_response.reason_phrase
- session.response.http_response.Server
- session.response.http_response.Set-Cookie
- session.response.http_response.status_code
- session.response.http_response.Success
- session.response.http_response.Transfer-Encoding
- session.response.http_response.Vary
- session.response.http_response.version
- session.response.http_response.VTag
- session.response.http_response.X-Frame-Options
- session.response.http_response.X-Pad
- session.response.http_response.X-Pingback
- session.response.http_response.X-Powered-By
- session.response.http_response.X-Sinkholed-Domain
- session.response.http_response.X-Xss-Protection
- session.response.http_response.X-XSS-Protection
- session.response.num_packets
- session.response.protocols
- session.session_num
- session.src_ip
- session.src_port
Appendix T

Miscellaneous Web Client Screenshots

This appendix displays screenshots from other parts of the web client which have not made it into the body of the report.

T.1 Main Page

Figure T.1 shows the main page of the web client. Clicking the menu icon in the top left slides out a menu which displays similar options to those visible on the main page. Clicking on the title (‘Sandnet Visualiser’) will return a user to this main page, whichever page they are currently on.

Figure T.1: The main page of the web client
T.2 Upload File

Figure T.2 shows the web form that is submitted when uploading a file to be run in the capture framework. The name and description are required. The timeout must be a minimum of 10 seconds. If the ‘Wait for analysis?’ checkbox is ticked then the form submission does not complete until the capture framework has run the malware sample submitted.

Figure T.2: The upload file page of the web client. This uploads a file to the capture framework for analysis.
### T.3 Upload Pcap

Figure T.3 shows the web form that is submitted when uploading a packet capture to the analysis framework. The name and description are required. Currently only a capture type of ‘pcap’ is supported.

![Sandnet Visualiser](image)

Figure T.3: The upload pcap page of the web client. This uploads a packet capture to the analysis framework.


## T.4 View Uploads

Figure T.4 displays the metadata for uploads/pcaps (see section 3.3) and allows a user to select one for individual analysis. Clicking on a row will launch the visualiser for that upload/pcap. Column visibility can be toggled using the menu icon in the top right of the table. In this screenshot several columns have been hidden to better fit this report. The refresh icon reloads the table and the search box allows a user to type in text to filter the rows displayed in the table. Table pagination is used to avoid displaying too much data. Different pages can be selected using the buttons underneath the table, as can the number of uploads/pcaps displayed per page.

![Sandnet Visualiser](image)

Figure T.4: The view uploads page of the web client
T.5 View Multiple Uploads

Figure T.5 displays the metadata for uploads/pcaps (see section 3.3) and allows a user to select multiple uploads/pcaps for analysis. Clicking on a row will mark its checkbox. When the desired number of rows have been selected the ‘Multi Visualise Selections’ button can be pressed which launches the visualiser for the selected uploads/pcaps. Other options are exactly as described in section T.4.

Figure T.5: The view multiple uploads page of the web client
T.6 Visualiser

Figure T.6 displays the empty visualiser page. Clicking on the ‘Select visualisation’ button displays the menu of available visualisations. Clicking on one of them displays the visualisation itself, along with relevant visualisation options.

Figure T.6: The visualiser, when first loaded, and the visualisation options available