Network Traffic Behaviour Profiling

Michal Piskozub

Wolfson College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2020

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Ivan Martinovic for his continued support and his willingness to go the extra mile for his students. He introduced me to the world of academic research and inspired me to pursue the DPhil degree in Cyber Security.

I would like to thank my parents for offering me support and guidance throughout this DPhil journey, and motivating me in moments of doubt.

I would like to thank everyone I met both in the College and Department; all members of the SSL group who have made my time in Oxford enjoyable and shared their opinions and took time to discuss research challenges. I am also very grateful to have had the opportunity to work with Riccardo Spolaor, Fabio De Gaspari and Freddie Barr-Smith who made this journey one of its kind.

I would like to thank Wolfson College Running Club and Tom Carruthers for always welcoming me to the weekly runs and showing me the beauty of this sport.

I would like to express my sincere gratitude to the CDT in Cyber Security, its staff, and The Engineering and Physical Sciences Research Council (EPSRC) that made it all possible. I am especially grateful for the support and kindness offered by Maureen York and David Hobbs, who were always there to provide a helping hand.

I would like to thank Prof. Andrew Martin and Prof. Guillermo Suarez-Tangil for taking the time to evaluate this thesis, discussing various aspects of proposed methods and giving suggestions for future work.

I would like to thank Dr Alfonso Gazo-Cervero and OxCERT for providing and helping understand ground truth in network traffic from the University of Oxford.

I am grateful for the support and services offered by VirusTotal and Google Cloud Security, in particular Emiliano Martinez, Bernardo Quintero and Sunil Potti.

Lastly, I would like to thank Wolfson College for making a large part of my life in Oxford feel like at home.

Abstract

Nowadays, computer networks have become incredibly complex due to the evolution of online services and the rapid growth of the number of smart devices such as smartphones, tablets and laptops. Most of users' information, even the most sensitive ones, are transmitted over the Internet. Unfortunately, due to this phenomenon we also see an increasing interest of malware developers who are able to find and exploit novel vulnerabilities in network devices to carry out their malicious intents. To tackle these threats, network analysts should be aided with advanced techniques to identify malicious traffic in order to guarantee the security of networks.

In this thesis, we aim to reduce the asymmetric advantage of attackers by examining malware detection and classification using flow-level network traffic. Our methods explore the ability to extract network behaviours generated by malware. We further evaluate the challenge of working with limited amount of data offered by flows to detect and classify network traffic of malware. Malicious flows are intertwined with benign ones originating from a production network to simulate the real-world settings. We gather one of the largest network flow datasets of malware in order to evaluate our proposals and show that we can detect unseen malware variants.

Moreover, we explore the behaviour profiling of network hosts in order to identify them on large networks. We extract unique behaviours and show that we can work only with the amount of information exchanged by hosts in order to successfully extract their unique behaviours and hence distinguish them from others. We show that while such an approach could be used for maintenance of networks, it may also be employed as an attack against network-based moving target defence (NMTD) systems, which is followed by countermeasures and guidelines to avoid such scenarios.

Finally, we propose a novel method of storing network flow data in a domain-specific binary file format, which is motivated by the lack of sufficient methods to process large-scale network data on the order of billions of flows. The binary format makes the analyses of methods in this thesis possible, especially when working with the University of Oxford dataset, which contains more than 181 billion flows. We show that our binary format improves the state of the art in terms of storage, while offering faster data processing techniques.

Contents

List of Figures **xiii**

List of Tables **xvii**

List of Abbreviations **xix**

1 Introduction **1**

1.1 Motivation 1
1.2 Contributions of Our Research 4
1.3 Scope of This Thesis 6
1.4 Published Work 6
1.5 Thesis Outline 7
1.6 Work Done in Collaboration 9

2 Background **11**

2.1 Overview 11
2.2 Network Intrusion Detection Systems 12
2.3 Network Traffic Types 12
2.3.1 Packets 13
2.3.2 Web Proxy Logs 14
2.3.3 NIDS Logs 14
2.3.4 Flows 15

3 Literature Review of Network Traffic Analysis **17**

3.1 Overview 18
3.2 Research Themes and Challenges 18
3.3 Behaviour Extraction from Network Traffic 18
3.3.1 Attack Detection 19
3.3.1.1 SSH 20
3.3.1.2 Denial of Service 21
3.3.1.3 Phishing 22
3.3.2 Malware Detection 22
3.3.2.1 Botnet 23

3.3.2.2 Domain Generation Algorithm-Based 25
3.3.3 Anomaly Detection 27
3.3.4 Network Traffic Classification 27
3.3.5 Network Topology Discovery 28
3.3.5.1 NAT Detection 29
3.4 Network Traffic Processing 29
3.4.1 Quantity-Related Challenges 29
3.4.1.1 Sampling 30
3.4.1.2 Quicker Processing Methods 31
3.4.1.3 Traffic Engineering 31
3.4.1.4 Visualization 32
3.4.2 Accuracy-Related Challenges 35
3.4.2.1 Timing Error Correction 36
3.4.2.2 Flow Watermarking 36

4 Host Profiling Attacks on Moving Target Defence Systems 37
4.1 Introduction 38
4.1.1 Contributions 39
4.2 Moving Target & Host Profiling 40
4.2.1 Threat Model 42
4.3 Related Work 43
4.3.1 Network-Based Moving Target Defence 43
4.3.1.1 NMTD on Real Host Identifiers 43
4.3.1.2 NMTD on Virtual Host Identifiers 44
4.3.2 Network Behaviour Analysis 44
4.3.2.1 Anomalies and Malware Detection 45
4.3.2.2 Host Profiling 45
4.4 Hostbuster Design 46
4.4.1 Fingerprint Generation Procedure 47
4.4.1.1 Flow Aggregation 47
4.4.1.2 Feature Extraction 48
4.4.1.3 Feature Selection 50
4.4.2 Classification Procedure 51
4.4.2.1 Ambiguous Fingerprints Identification 52
4.4.2.2 Model Training 54
4.5 Evaluation 54
4.5.1 Fingerprint Creation 55
4.5.1.1 Timeout Analysis 55
4.5.1.2 Macroflow Properties 57

4.5.1.3 Feature Selection 59
4.5.2 Experimental Results 62
4.5.2.1 Ambiguous Fingerprints Identification 62
4.5.2.2 Host Behaviour Profiling 64
4.5.2.3 Long-Term Analysis 65
4.5.2.4 Computational Efficiency 65
4.6 Discussion 67
4.6.1 Limitations 68
4.6.2 Evasion 69
4.7 NMTD Guidelines 69
4.8 Summary 71

5 Malware Detection in Large-Scale Network Traffic 73
5.1 Introduction and Background 74
5.1.1 Contributions 76
5.2 Related Work 77
5.3 Methodology 79
5.3.1 Flowset Creation 79
5.3.2 Feature Engineering 81
5.3.2.1 Feature Extraction 81
5.3.2.2 Feature Selection 82
5.3.3 Malware Detection 83
5.4 Evaluation 84
5.4.1 Dataset Specification 84
5.4.2 Malware Type Discrimination 89
5.4.3 Suspicious Traffic Identification 94
5.5 Discussion 94
5.6 Conclusion 98

6 Fine-Grained Malware Detection Using Enhanced Network Flows 99
6.1 Introduction 101
6.1.1 Contributions 102
6.2 Background 103
6.2.1 Challenges of Network-Based Detection 104
6.3 Related Work 105
6.3.1 Machine Learning Applied to Malware Analysis 105
6.3.2 Network Flows 106
6.3.3 Entropy of Packets 107
6.3.4 Detection of Malware from Benign 107
6.3.5 Family Classification 108

6.3.6 Comparison to Other Proposals 109
6.4 System Architecture 111
6.4.1 System Overview 112
6.4.1.1 Flow Extraction and Encoding 114
6.4.2 Denoising and Classification 115
6.5 Datasets 116
6.5.1 Noise Injection 121
6.6 Evaluation and Results 122
6.6.1 Clean Samples Classification 123
6.6.1.1 Phase 1: Binary Classification 124
6.6.1.2 Phase 2: Type Classification 125
6.6.1.3 Phase 3: Family Classification 126
6.6.2 Noisy Samples Classification 129
6.6.2.1 Phase 1: Binary Classification 130
6.6.2.2 Phase 2: Type Classification 131
6.6.2.3 Phase 3: Family Classification 133
6.6.2.4 Phase 1: Binary Classification of Unseen Malware . 134
6.6.3 Tier Comparison 135
6.7 Discussion and Limitations 136
6.7.1 Malware Taxonomy 136
6.7.2 Network Flows 137
6.7.3 Noise Injection 138
6.7.4 Unseen Type Classification 138
6.7.5 Sandboxes 138
6.7.6 Evasion 139
6.8 Future Work 140
6.9 Summary 141

7 Network Flow Analysis System and Binary File Format 143
7.1 Introduction 144
7.1.1 Contributions 146
7.2 Related Work 146
7.2.1 Packet-Level Traffic Collection 147
7.2.2 Flow-Level Traffic Collectors 147
7.2.2.1 Storage Formats 148
7.2.2.2 Indexing Methods 149
7.3 CompactFlow Format Design 150
7.3.1 CompactFlow File Header 151
7.3.2 Flow Binary Representation 154

7.4 Evaluation 157
7.4.1 Average Flow Size Experiment 160
7.4.2 Big Data Flow Storage 161
7.4.3 Port Analysis 161
7.4.4 CompactFlow Compression Experiment 161
7.5 Discussion 163
7.6 Summary 165

8 Conclusion **167**
8.1 Summary of This Thesis 167
8.2 Future Work 169
8.2.1 Flow Data Enhancement 170
8.2.2 Behaviour-Based Malware Taxonomy 170
8.3 Final Remarks 171

References **173**

List of Figures

2.1 Network traffic types ordered by information loss. 13
2.2 Example of conversion from packets to unidirectional flows. 16

4.1 Example of a discovery attack against an NMTD adopting periodic or reactive IP address hopping. 41
4.2 Dataset generation procedure. 47
4.3 Analysis of inactive timeout impact on macroflow extraction with the active timeout of 500 seconds. 56
4.4 Analysis of macroflow properties: distribution of flows. 58
4.5 Analysis of macroflow properties: distribution of macroflows. 58
4.6 Histogram of the most frequently used ports. 60
4.7 Impact of the number of considered features on classification performance with 95% confidence intervals. 62
4.8 Performance comparison of preprocessing with and without ambiguous fingerprint identification. 63
4.9 Performance comparison varying the number of hosts. 64
4.10 Analysis of macroflow properties: number of flows and macroflows. . 66
4.11 Long-term analysis on 25 randomly selected hosts. 66

5.1 Number of flows per flowset in the malware dataset. 85
5.1 Number of flows per flowset in the malware dataset (continued). . . 86
5.2 Distribution of the top two features projected on a two dimensional space. 89
5.3 Confusion matrix of malware categories classification. 91
5.4 Performance and proportion of considered samples varying the classification confidence threshold. 92
5.5 Samples (i.e. fingerprints) from the University of Oxford dataset, identified as generated by various malware types. 95
5.5 Samples (i.e. fingerprints) from the University of Oxford dataset, identified as generated by various malware types (continued). 96

6.1 MalPhase pipeline. Flows from an individual source are encoded and ran through a pipeline of three classifiers: (i) a binary benign/malicious classifier; (ii) a malware type classifier and (iii) malware family classifier for the given malware type. 111
6.2 MalPhase detailed architecture. A sliding window extracts M flows, which are encoded and compressed in a compact representation. The encoded flows are then ran through a denoising autoencoder to filter-out benign traffic. The remaining malware flow features are then passed on to the binary classifier, and successively to the type and family classifiers if the sample is malicious. 113
6.3 MalPhase tier system. MalPhase flow features are extracted using M sliding windows of increasing size, and successively fed to a window-specific Classification Module. 115
6.4 Precision, recall and F1-score for different tiers of the Binary Classification Modules. 124
6.5 F1-score for different tiers of the Malware Type Classification Module. 125
6.6 F1-score for different tiers of the five Malware Family Classification Modules. 127
6.7 Confusion matrix of tier 1 family classifer for trojan family. 128
6.8 F1-score for the tier 3 Binary Classification Module, with varying ratio of benign-to-malicious traffic. Noise ratio plotted on a logarithmic scale. 131
6.9 F1-score for the tier 3 Type Classification Module, with varying ratio of benign-to-malicious traffic. Noise ratio plotted on a logarithmic scale. 132
6.10 F1-score for the five tier 3 Family Classification Modules, with varying ratio of benign-to-malicious traffic. Noise ratio plotted on a logarithmic scale. 133
6.11 Precision, recall and F1-score on unseen malware families for the tier 3 Binary Classification Module, with varying ratio of benign-to-malicious traffic. Noise ratio plotted on a logarithmic scale. 134
6.12 F1-score for Binary Classification Modules of different tier, with varying amount of noise ratio. Noise ratio plotted on a logarithmic scale. 136

7.1 CompactFlow file format (in bytes). 151
7.2 Binary schema of CompactFlow records (in bytes). 153
7.3 CompactFlow control fields (in bits). 155
7.4 Bitmap index of IP addresses. 156
7.5 Comparison of binary file formats. 158

7.6 181 million flows from the University of Oxford stored using different flow collectors. 161
7.7 Most commonly used ports in one month of the University of Oxford dataset. 162

List of Tables

4.1 Roles of hosts included in the evaluation. 59
4.2 Features with highest RMI values after filtering the ones with highest correlation (selected features in bold). 61

5.1 Malware datasets details. 87
5.2 Features with highest RMI values (selected features in bold). 88
5.3 Malware types classification results on 5 fold cross-validation. . . . 90
5.4 Malware types classification performance on 5 fold cross-validation and proportion of considered samples with three classification confidence thresholds. 93

6.1 Malware sample size in related work. 105
6.2 Comparison of related work to our proposed systems: MalAlert and MalPhase. 110
6.3 Overview of malicious datasets. 117
6.4 Overview of benign datasets. 118
6.5 Overview of families selected for the evaluation. 120
6.6 Overview of unseen families. 121

7.1 Comparison of open-source flow collectors. 148
7.2 Comparison of compression methods used with a CompactFlow file of 67,257,407 unidirectional IPv4 flow records. 162

List of Abbreviations

ANN . . . . . . Artificial Neural Network

API . . . . . . Application Programming Interface

APT . . . . . . Advanced Persistent Threat

BOM . . . . . Byte Order Mark

C&C . . . . . . Command and Control

CE . . . . Conditional Entropy

CPU . . . . . . Central Processing Unit

DBMS . . . . . Database Management System

DDoS . . . . . Distributed Denial-of-Service

DGA . . . . . . Domain Generation Algorithm

DHCP . . . . . Dynamic Host Configuration Protocol

DNN . . . . . . Deep Neural Network

DNS . . . . . . Domain Name System

DoS . . . . . . Denial-of-Service

DPI . . . . . . Deep Packet Inspection

DT . . . . Decision Tree

FTP . . . . . . File Transfer Protocol

GDP . . . . . . Gross Domestic Product

HDD . . . . . . Hard Disk Drive

HIDS . . . . . Host-based Intrusion Detection System

HMM . . . . . Hidden Markov Model

HTTP . . . . . Hypertext Transfer Protocol

HTTPS . . . . Hypertext Transfer Protocol Secure

ICMP . . . . . Internet Control Message Protocol

IDS . . . . . . Intrusion Detection System

IETF . . . . . Internet Engineering Task Force
IP . . . . Internet Protocol
IPFIX . . . . . IP Flow Information Export
IRC . . . . . . Internet Relay Chat
ISP . . . . Internet Service Provider
kNN . . . . . . k-Nearest Neighbors
LOF . . . . . . Local Outlier Factor
MAC . . . . . Media Access Control
MAD . . . . . Mean Absolute Deviation
ME . . . . Marginal Entropy
MI . . . . Mutual Information
MIB . . . . . . Management Information Base
MTD . . . . . Moving Target Defence
MTU . . . . . Maximum Transmission Unit
NAT . . . . . . Network Address Translation
NIDS . . . . . Network Intrusion Detection System
NMTD . . . . Network-based Moving Target Defence
OFDP . . . . . OpenFlow Discovery Protocol
P2P . . . . . . Peer-to-Peer
PCA . . . . . . Principal Component Analysis
PCAP . . . . . Packet Capture
QoS . . . . . . Quality of Service
RAM . . . . . Random-Access Memory
RLE . . . . . . Run-Length Encoding
RMI . . . . . . Relative Mutual Information
SD . . . . Standard Deviation
SDN . . . . . . Software-Defined Networking
SNMP . . . . . Simple Network Management Protocol
SOC . . . . . . Security Operations Center
SSD . . . . . . Solid-State Drive
SSH . . . . . . Secure Shell

SVM . . . . . . Support Vector Machine

TCP . . . . . . Transmission Control Protocol

TLS . . . . . . Transport Layer Security

TOR . . . . . . The Onion Router

TTL . . . . . . Time to Live

UDP . . . . . . User Datagram Protocol

URL . . . . . . Uniform Resource Locator

VM . . . . . . Virtual Machine

VOIP . . . . . Voice Over IP

VPN . . . . . . Virtual Private Network

YAF . . . . . . Yet Another Flowmeter

1 Introduction

Contents

1.1 Motivation 1
1.2 Contributions of Our Research 4
1.3 Scope of This Thesis 6
1.4 Published Work 6
1.5 Thesis Outline 7
1.6 Work Done in Collaboration 9

1.1 Motivation

Network traffic analysis is the process of recording and examining the data flowing on a network in order to maintain its availability and security. It serves network administrators and analysts as a source of invaluable information regarding the state of their networks. These can easily consist of hundreds of nodes. The ability to monitor them in a robust way is becoming a harder task with current trends that increase their size and complexity. This results in larger amounts of data exchanged, which makes it harder to successfully maintain and protect networks from attacks.

Computer networks are reaching new levels of complexity with the increasing number of devices connected to the Internet. Under such conditions, network administrators have to face threats by adversaries who constantly look for ways

to deploy targeted attacks, such as ransomware, worms, and DDoS. For this reason, guaranteeing the security of networks against attacks has become crucial for enterprises and network administrators.

Researchers in the field of network security have proposed solutions that monitor hosts' communications to detect malicious behaviours or unwanted traffic. Some traffic analysis methods aim to obtain insights about the nature of traffic by examining network communications on a packet-level using techniques such as deep packet inspection (DPI). However, while packet encryption guarantees the privacy of user communication, it makes these approaches unfeasible since they cannot inspect the content of packets. To cope with this shortcoming, researchers have proposed a number of traffic analysis methods that rely on packet-level communications to extract information about their content, identify malicious behaviour or detect anomalies [1]. Most of these methods are robust against packet encryption and their goals range from identifying network services (e.g. peer-to-peer, HTTP) [2, 3] to inferring used apps and user actions on mobile devices [4, 5].

Due to a massive amount of traffic generated by large-scale networks, packet-level analysis is extremely difficult to carry out. For this reason, researchers propose other methods that can work with aggregated information about network communications - network flows. These are sequences of packets exchanged by two hosts and identified by the same protocol and pair of IP addresses and ports in respective directions, from which it is possible to extract statistical data at the cost of losing additional packet-level information. In previous years, many standards to capture flow data were proposed, such as Cisco's NetFlow [6] or IPFIX [7]. Those standards aim to provide network administrators with tools for high-level traffic monitoring (i.e. filtering based on a given port or most active hosts).

Among various goals of network traffic analysis, behaviour profiling is a useful technique that aids in network management and helps detect unwanted events. Behaviour profiling methods aim to build a model to describe a network host by extracting its behaviour in terms of communication patterns. A robust host profile can help:

(i) Discriminate between normal and anomalous traffic (which could be a result of a user accessing a different set of network resources).

(ii) Identify hosts with similar behaviour.

(iii) Infer the role of a host within a network (e.g. an Internet access point, web or mail server).

(iv) Continuously authenticate a host based on its behaviour (i.e. a set of fingerprints).

(v) Adopt measures to improve the health and performance of the whole network (e.g. Quality of Service).

(vi) Detect suspicious traffic that is caused by malware or an ongoing attack.

The quantity of new, unique malware samples has greatly increased over the years, from approximately six new daily samples in 2000 [8] to what a recent study estimates at 344,041 unique daily samples in 2018 [9]. Today, new malware families still find widespread success, with malware campaigns providing large economic incentives to their creators and recent attacks such as WannaCry and Zeus resulting in billions of dollars of losses. Many techniques have been proposed in response to this rapid growth in daily threats, ranging from traditional signature-based detection methods, to more recent approaches based on machine learning and behavioural modeling (for a good survey on the subject, the reader is referred to [10]). Generally, these approaches rely on a deep analysis of the behaviour of the malware and its interactions with network, file system and other running processes, requiring ad hoc software running on each system that needs monitoring. The field of malware network traffic analysis differentiates from these holistic approaches, focusing on methods to detect malware using only network-level information. Network-based approaches tend to provide simpler scalability and maintainability compared to holistic behavioural analysis approaches, as network traffic can be easily captured and analysed at a single (or a few) point(s) in the network. Moreover, network-level analysis is also more resilient to malware that attempts to identify and evade

the detection system itself, as analysis is usually performed by Network Intrusion Detection Systems (NIDS) that are physically separated from the monitored systems.

The main goal of this thesis is to examine novel ways of extracting behavioural profiles from low-level network traffic (i.e. network flows) and using them to detect an entity that generated them. We show that such profiles can be applied to identify and track network hosts based on their unique traffic. Furthermore, we study the problem of malware detection using real-world data from the University of Oxford and present a method to classify malware into types and families from only network flows, and to detect unseen malware.

1.2 Contributions of Our Research

This section focuses on providing details with regards to the contributions of this thesis that are based on our publications. It follows the structure of a research question followed by an answer in the form of our research outcomes.

(RQ1) Does network flow-level traffic contain enough information to create unique behavioural profiles of network hosts?

Similarly to real life, network hosts exhibit patterns of behaviours that repeat periodically. A subset of them may be considered unique across the "population" of hosts. In this part of our research, we aim to design a method that would allow to accurately model such behaviours. We limit our approach to use only network flows as the data source, which are the de facto ubiquitous and standard format of capturing network traffic. As a result we show that we can identify individual hosts from a group of up to 200 hosts, while using only the patterns of bytes transferred by them.

(RQ2) Is it possible and to what degree, to detect and classify malware from network flows?

Malware poses a significant threat with the increase of network traffic and the number of connected devices. Classical detection methods used on a per-host basis

do not scale under this new reality. Network traffic offers a promising way of capturing data from a number of devices in one centralised point. A large part of our research is devoted to finding novel methods of malware identification. We examine whether network flows contain enough information to create fingerprints of network communications initiated by malware. We then test the limits of our proposals by evaluating how effectively we can classify malware into types or even more fine-grained families. Our outcomes show that we can detect malware with the F1 score of more than 98%, classify them into types (F1 ranging from 79% to 96% depending on type), and distinguish between families that generated malicious traffic (F1 ranging from 84% to 99%). We also aim to limit the number of false positives from our results by presenting additional confidence values output by our proposals.

Additionally, in order for our method to be applicable to real-world scenarios, it should be able to work with network traffic that is a mixture of benign and malicious communications. Under the regular conditions, the vast majority of traffic generated by a host is considered benign. We include this assumption in our proposal by varying the number of injected benign traffic into traffic generated by malware. Our experiments show that it is possible to detect malware in such conditions (F1 of 83% for an equal amount of benign and malicious traffic which drops to 56% for traffic that contains eight times more benign communications than malicious). We further show that our method maintains its capability of classifying malware into types and families with the presence of benign traffic.

(RQ3) Is it possible to store network traffic in an efficient manner that could be used for future analyses?

In our research, in addition to malware traffic datasets, we use a large-scale dataset of flow-level network traffic from the University of Oxford. Due to its volume, storing or processing such amounts of flows has shown to be impractical using regular approaches. This motivated us to devise a new, domain-specific framework that allows for efficient storage of flows while maintaining high processing speed.

We compared our approach to the state of the art to find out that it allows to store flows using on average almost three times less space than previous work, and 24% less space than the most efficient proposal.

1.3 Scope of This Thesis

All proposals in this thesis work exclusively with network flow data. We highlight on multiple occasions that while packet-level data was a standard way of capturing network traffic in the past, it is currently not feasible due to the volume of network traffic, which implies large and costly capture infrastructure. Moreover, the capture of packets has become impractical with the common use of encryption for network connections. Network flows contain a minimal amount of information about a network communication (only select metadata is saved). While this might seem disadvantageous, it is the standard network data capture format, which is privacy-preserving as it does not convey any information regarding the content of packets.

1.4 Published Work

The base of this thesis is formed by published peer-reviewed papers. I am the lead author in the following papers:

(P1) Michal Piskozub, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. *"On the Resilience of Network-based Moving Target Defense Techniques Against Host Profiling Attacks"*. In Proceedings of the 6th ACM Workshop on Moving Target Defense (MTD '19), 2019. [11]

(P2) Michal Piskozub, Riccardo Spolaor, and Ivan Martinovic. *"MalAlert: Detecting Malware in Large-Scale Network Traffic Using Statistical Features"*. In ACM SIGMETRICS Performance Evaluation Review Vol. 46, No. 3, 2019. [12]

(P3) Michal Piskozub, Fabio De Gaspari, Frederick Barr-Smith, Luigi Mancini, and Ivan Martinovic. *"MalPhase: Fine-Grained Malware Detection Using Network Flow Data"*. In Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21), 2021. [13]

(P4) Michal Piskozub, Riccardo Spolaor, and Ivan Martinovic. *"CompactFlow: A Hybrid Binary Format for Network Flow Data"*. In IFIP International Conference on Information Security Theory and Practice, 2020. [14]

1.5 Thesis Outline

In this section we give a brief overview of this thesis. Each content chapter is summarised with regards to the proposed method, contributions, the research question it addresses and a paper it is based on.

- **Chapter 2** provides background information on network traffic analysis. It gives a short historical perspective on intrusion detection systems and describes their role in network-based analysis. Building on that, it proceeds to the description of network traffic types that have been used in the last decades with regards to the data they capture and their role in modern security solutions.

- **Chapter 3** is a guide through the literature of network traffic analysis from the security perspective. It proposes a taxonomy of research themes and the associated challenges by dividing the literature into behaviour profiling and network traffic processing. While the proposed methods in this thesis are all flow-based, we also discuss prominent instances of packet-based related work for the sake of completeness.

- **Chapter 4** is concerned with the detection of network hosts based on their network traffic. It is based on paper **P1** and addresses **RQ1**. We propose a system, Hostbuster, which is capable of aggregating network flows in a way

that allows to extract and select a set of features that form a fingerprint. Not each fingerprint can be considered unique depending on a set of hosts from which we aim to identify a host. Therefore, we use and adjust a method that relabels ambiguous fingerprints while outputting unique ones. We use a random forest classifier to train models that are responsible for identifying hosts. The results show that we are able to identify a host from a group of 25 and 200 with the F1 score of 94% and 77% respectively. In the long-term evaluation, we are able to identify a host even after 3 months based on the dataset from the University of Oxford. We present Hostbuster as an attack in the network-based moving target defence (NMTD) setting and describe countermeasures as guidelines for deployment parameters of NMTD systems.

- **Chapter 5** presents MalAlert – a system for detecting malware and then classifying them into types. It is based on paper **P2** and addresses **RQ2**. It is inspired by the method from Chapter 4, which it adapts for the malware context. Since malware traffic is much rarer than its benign counterpart, we define different aggregation criteria that aim to create more groupings of flows. In order to deal with malware that exhibit the same behaviours, we use a confidence threshold metric, which is based on the consensus of the random forest estimators. MalAlert is evaluated on a malware dataset comprising of more than 65 thousand samples and a real-world dataset from the University of Oxford. It detects and classifies malware into types with F1 score up to 94% and detects a number of malicious fingerprints in the real-world dataset.

- **Chapter 6** is a continuation of malware detection and classification theme. It is based on paper **P3** and addresses **RQ2**. We propose a system, MalPhase, which increases the scope from the MalAlert presented in the previous chapter, and aims to classify malware into types and also the more fine-grained families. It uses mixed benign and malicious traffic and is based on a new method that uses deep learning, that is autoencoders to filter out noise and neural networks to train the detection models. The phases from the name relate

to a multi-stage approach that starts with binary malware detection, and continues through malware type classification to malware family detection. In order to evaluate MalPhase we collected, to the best of our knowledge, one of the largest datasets of malware traffic that amounts to almost 1 billion flows that we obtained with the help of VirusTotal [15]. The results show that MalPhase is not only able to detect and classify malware into types and families even from the mixed network traffic, but it is also capable of detecting unseen malware samples.

- **Chapter 7** examines efficient ways of storing and processing network flows on a big-data scale. It is based on paper **P4** and addresses **RQ3**. It presents a binary file format, CompactFlow, which reduces the required storage space for network flows. Its design is hybrid as it features and builds upon a number of techniques from previous binary file formats and database-based solutions. In the evaluation, CompactFlow is compared to binary formats of the state-of-the-art flow collectors. It is used to store the University of Oxford dataset which contains over 181 billion flows and improves the amount of space required to store flows by the state-of-the-art collector by at least 24%.

- **Chapter 8** summarizes the thesis and its contributions and highlights the lessons learned from the presented research. It also briefly comments on the future work with regards to the use of flows and malware detection.

1.6 Work Done in Collaboration

The research presented in this thesis is based on the publications outlined in Section 1.4, which were created with the help of our collaborators. This section describes and gives credit to the researchers who contributed to the successful completion of those papers. All work and contributions not listed below were provided by me under the supervision of Prof. Ivan Martinovic.

In Chapter 4, Dr Riccardo Spolaor provided insights that allowed us to implement the method used in Hostbuster and proved to be an invaluable research companion

who contributed with discussions about the NMTD topic. He helped with the implementation of the ambiguous fingerprints identification (Section 4.4.2.1) and improved it, from the original method it is based on, to work on multiple processor cores which made the evaluation significantly faster.

In Chapter 5, Dr Riccardo Spolaor introduced the idea of a confidence threshold to cope with fingerprints that are shared across malware types. He contributed by implementing the code used to generate some of the figures, especially Figure 5.2 and Figure 5.5. Finally, he helped with the testing and validating the correctness of cross-validation and data labeling.

In Chapter 6, Dr Fabio De Gaspari helped with writing the underlying paper and contributed with valuable discussions about malware detection, the use of injected benign traffic and deep learning. He brought fresh opinions about which network flow traffic we should use (bidirectional) and which additional features to include (packet payload entropy). He proved to be irreplaceable with regards to the design and implementation of denoising autoencoders as pre-processing of data in our method (Section 6.4.2), which resulted in an improvement of MalPhase's detection and classification performance. Freddie Barr-Smith (a DPhil student) contributed by sharing his experience about malware, and made it possible to obtain malware network traffic from VirusTotal. Prof. Luigi V. Mancini contributed by discussing the early visions of MalPhase and providing feedback throughout the process of paper creation.

In Chapter 7, Dr Riccardo Spolaor helped verify the completeness of related work, especially with regards to database solutions (Section 7.2.2.1). He also provided a valuable discussion about the recovery techniques and conditions of faulty flows (Section 7.5).

2
Background

Contents

2.1 Overview **11**
2.2 Network Intrusion Detection Systems **12**
2.3 Network Traffic Types **12**
2.3.1 Packets 13
2.3.2 Web Proxy Logs 14
2.3.3 NIDS Logs 14
2.3.4 Flows 15

2.1 Overview

This chapter explains core terms and concepts for network traffic analysis from a security perspective. It starts by introducing network intrusion detection systems (NIDS) – services that monitor network traffic for the presence of unwanted or potentially malicious communications. NIDS may contain a number of analysis methods to achieve their goals. The approaches proposed in Chapters 4-6 may be utilized as parts of NIDS machine learning-based components.

Subsequently, we concentrate on network traffic types to explain the differences with regards to the amount of information, usefulness and popularity.

2.2 Network Intrusion Detection Systems

Intrusion detection is a field that is now several decades old, from the initial proposal by Denning [16] from 1987 that proposed methods of analysis of logs of various provenance to identify malicious activity. This led to the development of Network Intrusion Detection Systems (NIDS) such as Zeek (previously known as Bro) and Snort that facilitate the detection of cyber attacks in computer networks. A number of commercial intrusion detection systems followed.

Initially such systems operated by matching a set of static rules (also known as signatures) against the observed network traffic. However, it is important to highlight that a NIDS serves a different role than a firewall. The main role of a NIDS is to raise alerts after the suspicious connection has happened rather than automatically blocking it as it is taking place. The type of NIDS evolved with time to complement the detection by creating a profile of a network and looking at anomalous traffic. While this allowed for a more complete detection that does not require manual tweaking for every possible scenario, this approach may lead to a large number of erroneous alerts (false positives). Finally, a modern variation of a NIDS contain methods based on machine learning that use more complex models than signature-based approaches to learn (in a supervised or semi-supervised fashion) what constitutes a malicious or unwanted communication.

2.3 Network Traffic Types

Network traffic comes in a variety of formats. It is important to understand what each of them offers and what trade-offs to make when picking one for future analyses. Figure 2.1 shows an overview of main types on network traffic in a decreasing order of the amount of information contained in each type. In what follows, we describe and comment on each of them.

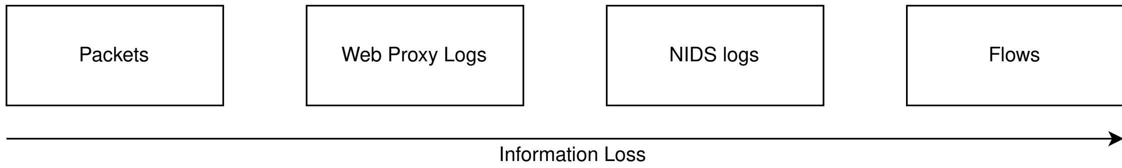


Figure 2.1: Network traffic types ordered by information loss.

2.3.1 Packets

All network communications between network-enabled devices use packets as the most fundamental unit to exchange data. Packets contain all the information that is transferred on a network, hence packet captures can be used to recreate all events that have happened in the past. However, this comes at a price of requiring a lot of storage to save them. Capturing them on large networks require expensive infrastructure and is often infeasible. Moreover, because packets contain not only routing metadata, but also content (i.e. payload) of each network communication, they are considered privacy-sensitive. Enterprises and internet service providers (ISP) in charge of their networks are wary of sharing them with third parties, which makes the choice of packets as data source for external NIDS impractical.

Deep packet inspection (DPI) is a popular technique to analyse packet payloads to determine their content. Historically, this was done for security reasons as part of NIDS or to enforce corporate policies. For example, some mobile ISPs used DPI to reduce the size of transmitted data by replacing detected images in packets with a scaled-down versions to limit network utilization. Nowadays the role of DPI has significantly diminished with the ubiquity of encrypted communications (i.e. TLS, QUIC). This negated the advantages of packets as data sources for security-based analyses and in some cases yielded them obsolete having in mind the challenges of storing and processing them. However, they are still the most important network traffic type from which others extract a subset of information.

2.3.2 Web Proxy Logs

Large networks usually employ a centralised service (i.e. a proxy), which queries websites on users' behalf and forwards replies to them. It also features a caching functionality which saves the frequently visited websites and forwards it to other users directly instead of communicating with the website again. Each such proxy creates logs to document which websites were visited by whom. In particular, web proxy logs contain the following fields: *timestamp*, requested *domain name*, *URL*, resolved *IP address*, *port*, *time elapsed* to serve the requested data, *amount of bytes to be sent*, *amount of actually sent bytes*, *transport-layer protocol*, *HTTP request method*, *HTTP status code*, and the *user-agent string* identifying user's browser. The logs also contain information about unsuccessful and unauthorized requests.

Web proxy logs are limited when compared to packets with regards to type of traffic they serve (only web traffic, usually using HTTP application-layer protocol), as well as with regards to the fields they contain. As a data source, they serve as an important type for methods that analyse behavioural patterns of web traffic. However, the same privacy-related problems apply to them as to packets, due to the inclusion of visited domains and downloaded resources (via the URL).

2.3.3 NIDS Logs

As we have seen in Section 2.2, there are a variety of NIDS that range from simple ones matching predefined signatures to sophisticated machine learning-based ones. The data they produce depends on a type of employed NIDS and its configuration. On an example of Zeek, the data can contain a connection, DHCP, DNS, HTTP, FTP or even file logs. This means that depending on the settings, the entirety of data can be quite large in terms of captured fields. However, for the purpose of machine-learning intrusion detection approaches, connection logs are the most popular data type due to their universality and no dependence on any specific underlying application-layer protocol. Connection logs contain the following fields: *timestamp*, *transport-layer protocol*, *duration*, detected *application-layer protocol*,

connection state, local or remote *connection flag*, *connection state history*, *source* and *destination IP addresses, ports, packets and bytes* exchanged. As we will see in Section 2.3.4, they are quite similar to flows in terms of the fields they contain. However, some of the fields in connection logs are unique to them (e.g. detected application-layer protocol and local or remote connection flag), hence they are a superset of flows in most cases.

2.3.4 Flows

On large-scale networks, traffic is collected by aggregating and assigning packets into network flows. This is due to a large number of hosts and a huge amount of traffic that they generate. Storing packet-level data, as we mentioned in Section 2.3.1, is not feasible in most situations, because of a significant cost of additional hardware to process and store such data. For this reason, information that are commonly used by traditional analysis methods (e.g. DPI, packet-level traffic modeling), which some NIDS relied on, cannot be applied anymore.

A network flow contains only vital data extracted from information exchanged between two network hosts. Even though there exist different network flow standards that allow capturing arbitrary data from packet-level traffic, the core fields constituting a flow are: *source* and *destination IP address*, *transport-layer protocol*, *source* and *destination port*, *start timestamp*, *duration*, number of *packets* and *bytes* sent. Flows can be divided with regards to their directionality into: unidirectional and bidirectional. The former captures only packets sent in one direction (e.g. from a local computer to a remote host), while the latter contains both directions. This implies that, when using a unidirectional format, the number of flows doubles to account for each direction (assuming that there exists at least one reply packet per request).

Compared to packets, flow-level network traffic is more privacy-preserving, and more scalable over the amount of traffic and number of connected devices in modern networks (i.e. packets are aggregated). The early mentions of a network flow define it as “an artificial logical equivalent to a call or connection” [17] and “a sequence of

#	Source IP	Source Port	Destination IP	Destination Port	Bytes
1.	10.8.0.1	55555	1.1.1.1	53	77
2.	10.8.0.1	55555	1.1.1.1	53	88
3.	1.1.1.1	53	10.8.0.1	55555	169
4.	10.8.0.1	55555	1.1.1.1	53	82

(a) Packets.

#	Source IP	Source Port	Destination IP	Destination Port	Packets	Bytes
1.	10.8.0.1	55555	1.1.1.1	53	3	247
2.	1.1.1.1	53	10.8.0.1	55555	1	169

(b) Flows.

Figure 2.2: Example of conversion from packets to unidirectional flows.

packets sent from a particular source to a particular unicast, anycast, or multicast destination" [18]. The first standard for exporting network flow information was NetFlow. Initially, NetFlow version 5 was released by Cisco in 1996 and then extended to version 9 in 2004 [6]. Subsequently, the Internet Engineering Task Force (IETF) in 2013 released the IP Flow Information eXport (IPFIX) Internet Standard [7] which is a further enrichment of NetFlow v9.

The process of flow conversion consists of grouping together packets with the same: *source* and *destination IP address*, *source* and *destination port* and *transport-layer protocol*. This set of fields is referred to as the 5-tuple. When operating in the unidirectional mode, packets are divided into two groups, each representing a flow in one direction, based on the order or source and destination IP addresses and ports. On the other hand, when operating in bidirectional mode the order in the 5-tuple does not matter as all packets are divided into one group based on the 5-tuple. Figure 2.2 shows the result of converting a set of packets to the resultant unidirectional flows. When packets are segregated based on the 5-tuple, their number and sizes are counted for each direction and appended as two additional fields to each flow that represents a direction.

3

Literature Review of Network Traffic Analysis

Contents

3.1 Overview . . . 18
3.2 Research Themes and Challenges . . . 18
3.3 Behaviour Extraction from Network Traffic . . . 18
3.3.1 Attack Detection . . . 19
3.3.1.1 SSH . . . 20
3.3.1.2 Denial of Service . . . 21
3.3.1.3 Phishing . . . 22
3.3.2 Malware Detection . . . 22
3.3.2.1 Botnet . . . 23
3.3.2.2 Domain Generation Algorithm-Based . . . 25
3.3.3 Anomaly Detection . . . 27
3.3.4 Network Traffic Classification . . . 27
3.3.5 Network Topology Discovery . . . 28
3.3.5.1 NAT Detection . . . 29
3.4 Network Traffic Processing . . . 29
3.4.1 Quantity-Related Challenges . . . 29
3.4.1.1 Sampling . . . 30
3.4.1.2 Quicker Processing Methods . . . 31
3.4.1.3 Traffic Engineering . . . 31
3.4.1.4 Visualization . . . 32
3.4.2 Accuracy-Related Challenges . . . 35
3.4.2.1 Timing Error Correction . . . 36
3.4.2.2 Flow Watermarking . . . 36

3.1 Overview

This chapter shows challenges in the field of network traffic analysis and research areas addressing them. It examines the literature to show current state of the art in terms of topics and approaches in network-based security. It is meant to be a roadmap of research themes in the field of network traffic analysis, which describes previous work rather than being a comparative evaluation thereof. We present the latter in Chapters 4-7 that report methods and contributions of this thesis.

3.2 Research Themes and Challenges

In our taxonomy, we consider flow-based network monitoring due to its feasibility and therefore popularity to be employed on modern networks. However, the review contains a small number of publications that concern Software-Defined Networks (SDN) or perform analyses using packet or proxy log data. These are included to complement current work in flow analysis and in many cases can inspire future research in that field.

Network traffic analysis challenges are divided into two main categories that are concerned with extracting underlying behaviours or dealing with structure and volume of data.

3.3 Behaviour Extraction from Network Traffic

Network traffic can be viewed as a source of distinct behaviours based on observed communications. Those behaviours can be applied to detect attacks or the source of infection in terms of infected hosts. It is especially convenient when compared to classical host-based approaches, which sought to detect malware based on its executable, which was prone to change with each malware mutation. However, the set of behaviours of each mutation remains the same, which makes network-based behaviour analysis a perfect candidate to tackle malware detection. Moreover, network data is obtained from centralised points instead of having to collect it from

each network device. Additionally, network behaviours may be used to help profile a variety of hosts by creating their unique fingerprints, which could be used to detect their topology or continuously authenticate devices.

3.3.1 Attack Detection

Identification of an attack using Intrusion Detection Systems lacks contextual information to get a wider understanding of the malicious event. Koning et al. [19] propose inclusion of network flow data to IDS data in a system – CoreFlow. The system is evaluated on 3 Bro IDS deployments and more than 50 routers.

Li et al. [20] present a survey of network flow applications, which includes an overview of existing network flow frameworks, network monitoring and attack detection such as port scanning, denial of service, worms, botnet and policy validation. Additionally, Hoque et al. [21] categorize network attacks and provide a taxonomy of tools that can help both network attackers and defenders. Buczak et al. [10] review machine learning and data mining methods for network attack detection in an exhaustive survey.

Sperotto et al. [22] explain that while in the past, packet monitoring was a popular practice it is no longer possible on current high-speed networks. Hence, the rising interest in network flow analysis to detect attacks. It presents a different paradigm, rather than concentrating on meaning of single packet, it focuses on the flow of data on a network. This paper is also a survey of attack detection using network flow data. It shows different taxonomies of attacks and discusses which of them are feasible to be detected using network flow data – denial of service, scans, worms and botnets.

Zhang et al. [23] divide intrusion detection systems (IDS) into two types: signature- and anomaly-based. The former tries to match an intrusion with a previously recorded one based on a calculated signature. The latter looks for deviations from a typical baseline behaviour. The authors investigate anomaly-based network intrusion detection methods: local outlier factor (LOF) and isolation

forest (iForest). They evaluate them on network flow data from University of Virginia network and conclude that iForest performs better in detecting anomalous events that are distinctive. However, they notice that their study takes into account only HTTP traffic from two static IP addresses.

Paredes-Oliva et al. [24] examine the impact of flow sampling on the detection of port scanning. The authors evaluate two existing port scan detection methods by testing their robustness with sampled NetFlow data. The methods include threshold random walk (TRW) and time-based access pattern sequential hypothesis testing (TAPS). Their results show that TAPS is much more resilient to sampling.

3.3.1.1 SSH

While the subject of detecting SSH attacks has been studied in the literature, one can not be sure of the outcomes of such attacks. Hofstede et al. [25] present a method to detect whether an SSH attack was successful based on network flows. It was evaluated on almost 100 servers, achieving close to 100% accuracy. There are scenarios where the proposed method fails. The first one is a technique called flow stretching in which random data is inserted to SSH packets to alter the similarity of resulting flows. The second is retransmission of data which does not create a new flow but only increases number of packets and packet sizes of the original flow.

Najafabadi et al. [26] present a method to detect SSH brute force attacks using machine learning on network flow data. Additionally, it includes legitimate users' data of failed SSH logins to examine whether and how accurately they can be distinguished from malicious attempts. The authors use an aggregation of NetFlow data, because it gives a better division of brute force data and legitimate attempt data. They extract flow features using domain knowledge of SSH. They confirm that those features are discriminative enough by testing them on predictive models and a number of classification algorithms. They find that decision trees C4.5 are easier to explain, because they show the contribution of features in the classification task. They evaluate the method on collected data from a campus network. The results

show that the built model achieves very good accuracy in detecting SSH brute force attacks. However, the authors conclude that NetFlow data are too coarse-grained to distinguish between legitimate failed logins and brute force attempts.

3.3.1.2 Denial of Service

Yu et al. [27] propose a traffic flooding attack detection and an in-depth analysis system. The authors developed a method that detects traffic flooding attacks classifies it by type by taking advantage of SNMP Management Information Base (MIB) data using a C4.5 decision tree. They analyse attack patterns using association rule mining. The method is evaluated on a testbed network that comprises of victim's device, attacker's agent devices, attacker's handler device and data collector. Traffic is generated using a tool that simulates DDoS attack traffic. The results show that attacks were identified with a 99% accuracy and attack types were classified with 100% accuracy.

This objective of paper [28] is to detect and defend against DDoS attacks in real-time. The authors use an Artificial Neural Network (ANN) algorithm trained to expose TCP, UDP and ICMP DDoS attacks. They create a clone of a real network with simulated traffic and DDoS attacks, and train the detector in Java Neural Network Simulator (JNNS) which is then fed into Snort-AI. The results show that different types of DDoS attacks are detected with 98% accuracy, which is higher than other known approaches. The authors evaluate their method further by training it on both recent and old dataset, which resulted in 100% and 95% detection accuracy for known and unknown DDoS attacks respectively.

Krupp et al. [29] present a method of detecting the origin of amplification DDoS attacks. It consists of two steps. First, it fingerprints scanners that are used during the reconnaissance stage of the attack. The fingerprint is used to correlate scanners with the attacks. Second, they link the scanners to the systems starting the attack by localizing them using Time-to-Live (TTL) feature.

3.3.1.3 Phishing

Han et al. [30] propose a honeypot system that is used to analyse phishing kits. The authors present a sandbox that is able to deactivate the phishing kit while leaving it working. This way they can continue examining the kit while user privacy is secure. The system is evaluated during an experiment that lasts 5 months and helps understand the lifecycle of phishing attacks.

3.3.2 Malware Detection

Yen et al. [31] present a system (TAMD) that detects hosts infected with stealthy malware. It takes into account that even though the operation of such malware is subtle, their behaviour can be exposed by looking for similar traffic patterns when more hosts on a local network are infected. The authors refer to those patterns as characteristics based on which communication aggregates are created. They evaluated TAMD on traffic captured at the edge of a university network. It detected new infections of a number of bots and spyware. This paper uses Argus flow records, however they are customized to include other non-standard flow features, e.g. 64 bytes of payload.

Bartos et al. [32] propose a classification system to detect known and unseen-before security threats. It consists of classifiers that use statistical feature representation calculated from network traffic. The authors created a method to group samples into bags in order to be invariant to common changes of malware behaviour. The system was evaluated on large corporate networks, where it detected 2090 new and previously unseen malware variants with the precision of 90%.

Rossow et al. [33] note that dynamic malware analysis usually concentrates on host-level activities such as system calls. They observe that analysis of malware behaviour on the network is not as popular and address it in their overview of over 100,000 malware samples. Behaviour of those samples was analysed in a system the authors propose - SandNet, which focuses on the use of DNS and

HTTP protocols and is meant to be used as a complementary tool to existing network traffic analysis systems.

Grégio et al. [34] observe that with the evolution of malware to complex entities, it is no longer sufficient or meaningful to rely on malware naming schemes or symbols which the researchers label them with. Instead, they notice that malware exhibit similar behaviours and propose a taxonomy based on them. It is focused on behaviours on a local and network level. The suspicious behaviours are classified into: evasion, disruption, modification and stealing. The taxonomy is evaluated in a case study involving over 12,000 malware samples, of which 21% were not detected by any antivirus tool, but were classified as malicious using the taxonomy. The authors conclude that the proposed method can be used as a complementary source of information to antivirus scan results.

Celik et al. [35] propose a set of flow features that discriminate malware. They discard features that are prone to show false positives. The remaining ones are referred to as tamper-resistant features. The authors divide the identified features into static and dynamic, which are the ones that are independent of packet headers and the ones that are calculated from packet headers throughout flow lifetime respectively. The paper examines the evolution of malware evasion techniques based on 16 malware families. While the results show good detection accuracy, it drops significantly for newer malware families that disguise their traffic by misusing ports linked to common protocols and by using traffic shaping techniques to hide their activities. Features selected in this paper require capture of additional data by the flow collector, e.g. min data size feature relies on payload size whereas flow collector captures only sizes of whole packets. Therefore, it is not possible to apply the method presented in this paper to regular flow data.

3.3.2.1 Botnet

A large body of work aims to detect botnets and their communications from network traffic due to their heavy reliance on the use of the Internet to propagate and function.

BotHunter [36] is an application that monitors infections and coordination dialog on a network. It tracks connections between internal and external devices and gathers them to create a history of actions between a malicious entity and the infected device. BotHunter includes a correlation engine that detects the infection progression and merges intrusion alerts with external traffic that corresponds to the infection. The authors call this approach dialog-based correlation. The work is evaluated on live and simulated systems. The authors conclude that the system is highly scalable and reliable.

BotSniffer [37] is a system to detect botnets based on anomalies and independent of signatures or command & control (C&C) host IP addresses. It can detect both infected hosts and C&C servers. The authors explain that their method uses spatial-temporal correlation and similarity that exploits similar behaviour of bots communicating with the C&C server. BotSniffer is evaluated on several real-world network datasets.

BotMiner [38] is a system to detect botnets that does not depend on their command & control (C&C) layout (centralised or peer-to-peer) or protocol used for communication. The authors take advantage of the fact that botnet traffic from various clients has similar behaviour patterns. They cluster hosts' behaviour and correlate them, which highlights hosts with similar traffic in terms of destination and malicious activity. The results show that BotMiner can detect IRC-, HTTP-, and P2P-based botnets with a very low false positive rate.

BotTrack [39] incorporates an unusual method of detecting botnets. It correlates NetFlow data to create a host dependency model. It uses a novel approach by extending a popular PageRank algorithm that is commonly used to analyse linked data. PageRank helps detect peer-to-peer botnets that operate stealthily and do not generate large volumes of network traffic. The authors evaluate the method on data from a major ISP in Luxembourg.

Kheir et al. [40] observe that botnets are becoming more complex and shift to a peer-to-peer (P2P) architecture. They notice that detection methods of such botnets rely on multiple infected hosts on a local network and fail to detect botnets using

more sophisticated means of P2P communications such as public P2P networks, anonymity networks (e.g. Tor) or encapsulation of P2P data in HTTP traffic. The paper proposes a behaviour-based approach to detect botnets – BotSuer. It uses high-level network features available in NetFlow data. The authors run malware samples in a sandbox and capture statistical features which are then used in a machine learning algorithm to create a detection model. The results show that the system is able to detect single infected P2P bots with a very low false positive rate.

DISCLOSURE [41] is a large-scale, wide-area botnet detection system. It contains novel methods that reduce the weaknesses of NetFlow data. The authors select features that allow them to detect command & control (C&C) channels, which include flow size, client access patterns and temporal behaviour. They include a number of external reputation scores in their system to lower the false positive rate. They evaluate the system on two large, real-world networks, where it is able to perform a real-time detection of botnet C&C channels on the order of billions of flows per day.

3.3.2.2 Domain Generation Algorithm-Based

It is known that botnets using a centralised design communicate with botmaster's server, which becomes the weakest link in that infrastructure. To alleviate that botnet writers removed the centralised command & control server by changing the layout to peer-to-peer. However, it increases complexity of the malware creation process and therefore delays its completion. Another approach, that is an evolution of a centralised design, is to generate domains automatically following an algorithm, which is referred to as domain generation algorithm (DGA). Modern botnets frequently produce a large number of domains using such algorithms and select only a few of them for actual C&C use, which are valid for a short amount of time. The usual countermeasure is to understand how the algorithm works, usually by reverse engineering malware binaries, and register those domains before the botnet to neutralise it.

Antonakakis et al. [42] present a different method in their DGA-based malware detection system (Pleiades) that analyses DNS traffic. It correlates unsuccessful DNS requests with operation of DGA-based malware, thereby presenting a novel way of detecting them instead of manual reverse engineering of malware executables to find more details about the DGA. The authors evaluated Pleiades over the course of 15 months in a major ISP. It detected 6 known and 6 unknown DGA-based malware.

Yadav et al. [43] present a method to detect domain flux attacks of botnets, where each bot requests a set of domain names in order to find out whether they exist, with the intention of registering only a small subset of them for command & control (C&C) purposes. The authors explain that their approach looks for patterns in domain names that are representative of domain generation algorithms (DGAs). They compare their method on distance metrics that include K-L distance, edit distance, and Jaccard measure. They test their method on data from a Tier-1 ISP and successfully detect domain fluxing in Conficker botnet. They further evaluate it on a campus DNS dataset and uncover an unknown botnet with a more sophisticated domain generation algorithm.

Grill et al. [44] observe that botnet detection methods such as deep packet inspection (DPI) or reverse engineering suffer from privacy invasion issues. They propose a system that is able to process network traffic on a scale of big data without reverse engineering botnet binaries or performing Non-Existent Domain inspection. Their method uses a ratio of DNS requests to the number of contacted IP addresses for every source IP on a local network. They expectation is that DGA malware will perform significantly more DNS queries than distinct traffic connections. Every deviation from their model is labelled as an anomaly and potential DGA-based malware in operation. Their evaluations indicated that their method detected one infected host and several hosts performing many DNS requests, that upon examination appeared to be servers that were logging DNS information from their clients.

3.3.3 Anomaly Detection

Lazarevic et al. [45] perform a comparative study of anomaly detection methods in the context of network intrusion detection, which include k-nearest neighbor (kNN), Mahalanobis distance, local outlier factor (LOF) and support vector machines (SVM). It evaluates supervised and unsupervised anomaly detection schemes on the DARPA 1998 dataset with the 74% and 56% accuracy for multiple- and single-connection attacks respectively.

Rehak et al. [46] present a method that augments intrusion detection systems with trust models spread across multiple detection agents. The detection process is divided into three parts: anomaly detection, trust model update and collective trusting decision. The result of this process is a trustfulness score for every flow. When in doubt, the score can be inspected manually, which decreases the number of flows to analyse. The authors evaluate the method on a set of experiments performed on a real network data.

Bhuyan et al. [1] present a survey of anomaly detection methods that is meant to be a comprehensive introduction to that field for researchers. They provide a topology of network anomaly detection techniques and show popular attacks detected by network intrusion detection systems. Additionally, they outline tools used for network monitoring and datasets available for use for anomaly detection.

3.3.4 Network Traffic Classification

Network traffic classification has become a popular topic. There are several machine learning-based methods proposed in the literature. However, the authors of [47] notice that there is hardly any research on traffic classification of sampled flow data. They evaluate popular methods to classify traffic to find that the results are similar in terms of accuracy to packet-based classification methods, but they diminish with the introduction of sampling. They propose an improved sampling method which achieves comparable accuracy, with 1/100 sampling, when compared to classification of unsampled flows. The authors identify limitations of current

machine learning approaches and observe that the lack of publicly available network data hinders progress in this area of research.

A survey paper [48] shows an overview of machine learning-based methods for the classification of network traffic by reviewing significant 18 papers. The authors provide a guideline in terms of requirements for applying machine learning-based methods of traffic classification to existing networks.

Karagiannis et al. [2] present a traffic classification method that is based on identifying host behaviour patterns using network flow data. The main contribution of the paper is the division of patterns into three levels of increasing detail: social, functional and application. The approach is oblivious to port numbers and contents of the communication. The results show that the method is able to classify 80-90% of the traffic with the accuracy above 95%.

Mariconti et al. [49] presents a method that is concerned with distinguishing two types of malware, targeted and generic, using network behaviour patterns. The authors create an array of sandboxed environments, which they infect with malware samples. They then capture all network traffic produced by those environments and choose features based on an analysis of the whole dataset. They extract 441 features based on Markov chains and reduce their number by using Principal Component Analysis (PCA). They use statistical classifiers to group traffic to targeted and generic malware groups with an accuracy of almost 96% in the best case scenario.

3.3.5 Network Topology Discovery

Pakzad et al. [50] note that in SDNs, contrary to regular networks, forwarding devices are not responsible for network management. It is a task of the SDN controller, which needs to have an up-to-date view of networking devices in order to work properly. To address this, the authors propose a topology discovery method that is a modified version of the technique used in SDN controller frameworks (OpenFlow Discovery Protocol - OFDP). It successfully decreases the number of

packets sent to networking devices to update the topology of the network. Depending on network layout, it offers a significant improvement over the OFDP.

Azzouni et al. [51] present a method for faster and more secure topology discovery for software-defined networks (SDNs).

3.3.5.1 NAT Detection

Verde et al. [52] recognize that network traffic of users can be used as their biometric signature. However, on large networks, IP addresses on a local network are mapped to one static IP address that is publicly visible on the Internet. That process is referred to as network address translation (NAT) and it dilutes data of a single user by aggregating traffic of a larger number of users. This paper presents a fingerprinting framework that is able to separate traffic belonging to given users using only NetFlow data. It is evaluated on a large metropolitan WiFi network reaching 200,000 users with an average of 1000 users active simultaneously behind 2 NAT'd IP addresses. The framework was able to detect single-user traffic with 90% accuracy.

3.4 Network Traffic Processing

While flow-level network traffic offers a significant improvement in terms of data stored over packet-level data, it still requires complex infrastructure to be able to process data on large-scale networks. In this section, we divide network traffic processing challenges into quantity- and accuracy-related.

3.4.1 Quantity-Related Challenges

Estan et al. [53] propose an improved version of NetFlow that is able to dynamically adjust the sampling rate - Adaptive NetFlow. It does so without a decrease of accuracy and it is available as an update for NetFlow-supported routers. The second contribution of this paper is an extension to count non-TCP flows. Adaptive NetFlow is evaluated on simulated DoS attacks that confirm its robustness.

3.4.1.1 Sampling

Sampling is a technique that captures a subset of data by discarding some pieces of them. It implies information loss. The goal of the techniques presented in the literature is to try to decrease it.

Bartos et al. [54] propose a method of sampling flows that minimizes the change of statistical properties of the data, which is frequently present in sampling algorithms. It introduces two new concepts: late sampling (features can be extracted before sampling and attached to surviving flows, which preserves statistical distribution) and adaptive sampling (distribution of surviving flows is skewed to account for rare flows, which preserves the variability of the data). The proposed method was evaluated on NetFlow data and HTTP proxy logs.

Brauckhoff et al. [55] analyse how the sampling of packets in a flow affects the anomaly detection metrics. The authors focus on a single anomalous event (Blaster worm). They assume to have fine-grained information about flows (packets sizes). They find that sampling does not impact anomaly sizes in terms of packet sizes and number of packets, but it impacts the number of generated flows. Entropy-based summarizations of packet and flow counts are methods that are more resilient to sampling.

Sampling is a successful way of dealing with large quantities of data. Several flow frameworks, such as Cisco NetFlow v9, provide sampling capabilities by default. It makes flow capture possible in multi-node enterprise networks, which would not be feasible otherwise. However, it further decreases the amount of information that are already reduced to the minimum by the flow capture process. The methods presented in the literature show that the negative effects of sampling can be diminished, but it does not mean that sampled flows could be treated equally with flows captured without sampling in terms of the amount of information and certainty that they accurately represent real-world events.

3.4.1.2 Quicker Processing Methods

Hofstede et al. [56] provide an excellent tutorial on all phases of flow monitoring. It explains the motivations of moving from packet-level data capture to network flows. It builds reader's understanding from the bottom up and starts by explaining history of flow export protocols. Then it moves to an overview of a common flow monitoring architecture that is divided into packet observation, flow metering & export, data collection and data analysis. It is a tutorial worth reading for everyone interested in understanding the details behind network flow capture and analysis process.

Lee et al. [57] notice the explosive growth of Internet traffic that hinders successful network monitoring. They address it with a scalable, Hadoop-based traffic monitoring system. It is able to perform parallel modifications of network data in the form of packets and NetFlows. To simplify processing and interaction with big data they implemented a web-based interface in Hive to be able to query and visualize the data. Their evaluation shows that the system achieves up to 14 Gbps of throughput for 5 TB input files.

Marchal et al. [58] observe that network traffic analysis gets increasingly harder with larger volumes of data. They propose a scalable architecture for large scale intrusion detection on DNS, HTTP and honeypot traffic using the NetFlow data format. They propose a framework and evaluate five most popular big data management solutions (Hadoop, Hive, Pig, Spark and Shark) to find the most suitable for their methods that include data extraction, storage and correlation scheme. They choose Spark and Shark based on performance for their intrusion detection methods.

3.4.1.3 Traffic Engineering

In order to optimize and evaluate the performance of networking infrastructure, traffic engineering methods are used to predict the amount of data that will occur in the future.

Wallerich et al. [59] analyse network traffic data in terms of Zipf's law for IP flow rates. They focus on large flows and create a function of persistency properties to time. They define elephant flows as ones that constitute the largest size of packets; and mice as the opposite. They argue that understanding of persistency features and behaviour of elephant flows would result in better traffic engineering efforts, which aim to optimize the network to be able to manage them. The authors identify the difficulty when exploring persistency aspects is the format of data. It is either too fine-grained (packet-level data) for larger quantities of data or too coarse-grained (flow-level data) to get enough information. The paper also examines and compares the accuracy of analysis based on packet-level and NetFlow data. The results show that if a flow is an elephant at least once then it is very likely to remain as such for majority of time throughout its lifetime. The authors find that coarse-grained data is enough for most scenarios, however elephant flows should be examined in more detail using fine-grained methods.

Liu et al. [60] present a framework (UnivMon) for software-defined flow monitoring that addresses the problem of estimating the amount of network traffic and is at the intersection of low fidelity approaches such as sampling and high fidelity ones that are tuned to calculate only a specific metric. UnivMon features monitoring module in the data plane and estimation algorithms in the control plane. It uses statistics from the data plane to calculate application-level metrics.

3.4.1.4 Visualization

Yin et al. [61] present a visualization tool for network administrators that enhances their ability to spot anomalies on a network. It uses a parallel axes visualization that shows a high-level overview of the network data between internal and external hosts with the aim of increasing user's ability to detect attacks and gain better situational awareness. User can filter and zoom in to find more information about selected network communication. The visualization uses four main views: global, domain, internal, and host statistics. In global view three parallel axes are used for: external sender, internal host, and external receiver. While the idea is good,

it seems that the data is mirrored (because of the use of unidirectional flows). It would be much better if the flow initiated from an external sender were depicted by a line from first to second axis, while flows initiated by internal host were shown as a line from second to third axis. This way it would be possible to see who initiated the connection, but it would require flow data in a bidirectional format (which was not known at the time of writing that paper). However, it will not be possible to see requests to which there were no responses with the proposed model.

Irwin et al. [62] presents visualization that uses Hilbert curve to map IP addresses. It preserves distances between consecutive IP address ranges and shows data flow on a network. The authors used the tool to evaluate propagation of a worm from a simulated dataset.

The introductory paper [63] provides a good overview of network visualization techniques. It divides them into packet trace and network flow visualizations mentioning examples from the literature for each category.

Taylor et al. [64] present an interactive visualization tool that shows historical network flow data. It creates a 3D impulse graph to show timestamps, ports and total size of packets from network flows. User can rotate the graph to obtain 2D orthogonal views of timestamps and bytes; or ports and bytes.

Mansman et al. [65] present a visual analytics tool that shows graphically the behaviour of hosts by changes in position in a force-directed graph, which is an adaptation of Fruchterman-Reingold graph layout. It helps analyse temporal changes visually. The tool was evaluated on network traffic from a university gateway router and IDS alerts from Snort.

Fischer et al. [66] propose a system (NFlowVis) to examine NetFlow data using a relational database. The system introduces a visualization that combines a tree map, clustering algorithm and Hierarchical Edge Bundles to group and present flows. The authors conducted three case studies to evaluate capability of the system to detect attacks and monitor traffic on a network. It detected a distributed SSH attack that was unknown before because of its low throughput, which helped evade the detection by intrusion detection systems.

While threshold alerting is easy to implement to monitor network behaviours it does not offer a way of correlating traffic or devices. Stoffel et al. [67] address this with a visual analytics application that uses similarity models and time-series visualization analytics to help users examine and investigate correlations in time-series data. It shows a set of vertical line charts placed next to each other that represent network services. They explained how the system can be used by an analyst to detect anomalies with minimal cognitive effort.

Braun et al. [68] introduce an interactive web application (Flow-Inspector) for dynamic network flow visualization. It is implemented using JavaScript (D3.js and Backbone.js libraries) and it features a number of visualizations: time series, host distribution, force graph, hierarchical edge bundle and hive plot. The tool is open-source and available online.

Gray et al. [69] present a visualization technique to aid with origin identification of attacks. The authors refer to it as contextual navigation. They show that the Internet can be mapped topologically using it.

Angelini et al. [70] propose a novel visual analytics environment (PERCIVAL) that provides better situational awareness for network monitoring purposes. It allows to compare proactive network security measures with the progress of an attack. PERCIVAL was evaluated in a user study with the results indicating that in 71% of cases the system is effective for dealing with the intended problem.

Guimaraes et al. [71] create a comprehensive survey of visualization techniques that support the network and service management process. It provides a systematic literature review that classifies 285 articles and papers ranging from historical to current state-of-the-art ones. The authors conclude by predicting future research directions in the field.

Arendt et al. [72] present a visualization technique that provides a flexible map of a network. It was built to show the progression of Cyber Defense Exercise (CDX).

Coudriau et al. [73] propose visualization of network traffic data derived from the Mapper algorithm from the field of topological data analysis (TDA). It shows patterns of network traffic data and make it easier to spot malicious connections

in large datasets. The authors applied their tool to darknet data and discovered patterns that were not detected by a popular Suricata IDS.

Post et al. [74] notice that while in classical networking solutions every node is responsible for itself, in the emerging world of software-defined networking (SDN) there is a separation between network control and data plane layer. SDNs consist of network controllers and networking nodes. The latter form a hierarchy and report to the controller, which can make decisions about the whole network of nodes. Their paper presents an interactive analysis tool of SDN hierarchy and data flow. It contains the following visualizations: linked view (divides network hierarchy into circles and shows data flow between them using hierarchical edge bundles) and flow tracking view (a subway map-inspired visualization with time on x-axis and nodes forming horizontal containers on y-axis, where flows are connected between nodes by color-coded splines). The rationale behind the latter, is the ability to see temporal changes of flows such as their direction and traveled-through nodes, which the former visualization lacks. Therefore, the flow tracking view complements the linked view, which the authors show in a use case that includes a text-based tree view. They evaluate the system using a generated dataset that simulates the operation of an SDN.

He et al. [75] present a visualization system (NetFlowVis) to analyse communication patterns and anomalies from NetFlow data. It features four views: communication trajectories view (a link-node tree that shows connections between clients and servers), traffic line view (statistical protocol data in both directions), snapshot view (captures anomalies for later processing) and protocol view (traffic distribution based on protocols).

3.4.2 Accuracy-Related Challenges

This section describes briefly the research themes regarding the precision and correctness of the captured network flow data.

3.4.2.1 Timing Error Correction

Trammell et al. [76] correct the timing error in NetFlow v9 data. The authors observe that the timing error, that is clock skew and export delay, is caused by different flow framework implementations. They find another one - cyclic error. It is responsible for the delays of up to one second and is inherent to the export protocol design. They present a method of correcting it, which is limited to the accuracy of 70ms on routers on their network.

3.4.2.2 Flow Watermarking

Active traffic analysis is a trending field of research. One of its subsets is flow watermarking - a technique to embed a recognizable pattern in flow features with the intent to identify selected flows at specific points in the network.

Iacovazzi et al. [77] present a survey of popular watermarking methods from the literature, which they categorize and analyse. It highlights challenges and goals of a flow watermarking system and provides detailed description of its architecture. The authors observe that the most important factor is the choice of the watermark carrier. They also point out that watermarks should be robust (in terms of survivability of a watermark) and invisible.

4
Host Profiling Attacks on Moving Target Defence Systems

Contents

4.1 Introduction **38**
4.1.1 Contributions 39
4.2 Moving Target & Host Profiling **40**
4.2.1 Threat Model 42
4.3 Related Work **43**
4.3.1 Network-Based Moving Target Defence 43
4.3.1.1 NMTD on Real Host Identifiers 43
4.3.1.2 NMTD on Virtual Host Identifiers 44
4.3.2 Network Behaviour Analysis 44
4.3.2.1 Anomalies and Malware Detection 45
4.3.2.2 Host Profiling 45
4.4 Hostbuster Design **46**
4.4.1 Fingerprint Generation Procedure 47
4.4.1.1 Flow Aggregation 47
4.4.1.2 Feature Extraction 48
4.4.1.3 Feature Selection 50
4.4.2 Classification Procedure 51
4.4.2.1 Ambiguous Fingerprints Identification 52
4.4.2.2 Model Training 54
4.5 Evaluation **54**
4.5.1 Fingerprint Creation 55
4.5.1.1 Timeout Analysis 55
4.5.1.2 Macroflow Properties 57
4.5.1.3 Feature Selection 59
4.5.2 Experimental Results 62

4.5.2.1 Ambiguous Fingerprints Identification 62
4.5.2.2 Host Behaviour Profiling 64
4.5.2.3 Long-Term Analysis 65
4.5.2.4 Computational Efficiency 65
4.6 Discussion 67
4.6.1 Limitations 68
4.6.2 Evasion 69
4.7 NMTD Guidelines 69
4.8 Summary 71

Researchers propose Moving Target Defence (MTD) strategies for networking infrastructures as a countermeasure to impede attackers from identifying and exploiting vulnerable network hosts. In this chapter, we investigate the weaknesses of Network-based Moving Target Defence (NMTD) against passive host profiling attacks. In particular, we consider periodical and reactive approaches to change hosts' identifiers. To evaluate the capabilities of a host profiling attack, we design Hostbuster, a tool that reidentifies hosts based on network flow data. We experimentally evaluate its effectiveness using real-world network traffic from the University of Oxford. We show the robustness of learned host profiles, which are valid for more than two months. On average, our experiments result in 80% classification performance given by the F1 score. As a result of these analyses, we provide guidelines to strengthen NMTD against these types of attacks.

4.1 Introduction

The advances in network traffic monitoring and analysis to secure large-scale networks do not stop attackers who continue to probe hosts via scanners to discover the topology of a network and to find vulnerabilities that they can exploit. Although it requires a significant amount of time and resources, this process is performed as a one-off task as typically network topologies are static. To hinder attackers' efforts in discovering the topology of a network and vulnerable hosts, security experts adapted and applied Moving Target Defence (MTD) strategy to networks. MTD

is a military strategy that aims to conceal the identities of assets by changing their locations. In computer networks, a popular application of MTD consists of randomly changing identifiers of a host [78, 79]. Those changes can be triggered automatically in a periodic fashion or only when an attack is detected [80, 81]. Employing MTD results in a significant effort for an attacker, who now has to redo the reconnaissance phase every time hosts' identifiers change.

In this chapter, we discuss and analyse the weaknesses of Network-based Moving Target Defence (NMTD) strategies against a passive attacker who is able to monitor network flows. Such strategies involve changing hosts' identifiers, in a process that is referred to as hopping, to hide network resources from potential adversaries. To show that an attacker can still reidentify a host by overcoming the aforementioned constraints, we design and implement a host profiler, Hostbuster. Our tool relies only on patterns of bytes exchanged, to build the profile of a host. Such a profile is composed of fingerprints that capture the unique behaviour of a host. Since Hostbuster has to cope with traffic generated by large-scale networks, it is designed to be efficient in terms of computational resources, storage space, and time required. We evaluate our tool in a set of experiments by taking into account a large number of hosts and studying its classification effectiveness over an extended period of time. We assess performance of Hostbuster on a real-world dataset of network activity from the University of Oxford. Such a dataset comprises a total of $119,853,493,260$ network flows in a unidirectional format collected in a period of three months from December 2015 to the end of February 2016 (i.e. 91 days). According to our results we discuss which hopping strategies and parameters can be used to overcome this type of attack, in an attempt to strengthen NMTD deployment settings.

4.1.1 Contributions

The contributions of Hostbuster can be summarized as follows:

(i) We investigate a passive attacker's capabilities in NMTD scenarios. In particular, we analyse the implications of hopping strategies (i.e. periodic and reactive).

(ii) We assess the capabilities of a passive attacker by presenting Hostbuster - a tool that passively profiles hosts from their flow-level traffic on large-scale networks. Since our aim is to reidentify hosts in an NMTD scenario, we design our tool to be time-efficient. We evaluate Hostbuster on a real-world dataset in two analyses based on a large number of hosts and an extended time period of three months.

(iii) In the light of our results, we provide guidelines about the security of the strategy, timing and identifiers involved in the NMTD hopping process.

4.2 Moving Target & Host Profiling

Techniques that rely on NMTD are mainly in place to disrupt the reconnaissance phase of an attack (i.e. to correlate an IP address of a host with its vulnerabilities) by randomly changing hosts' identifiers such as protocols, ports, MAC and IP addresses. As a desirable property of an NMTD, the identifier hopping has to be unpredictable and on a large range of possible values (hopping space). Depending on the adopted strategy, hopping can be triggered periodically or as a reaction to a detected attack, e.g. by a Network Intrusion Detection System (NIDS). A hybrid approach that combines two aforementioned strategies is also possible. It is worth noting that hopping has a cost in terms of network traffic overhead (see Section 4.3).

In a network, some hosts can provide services that are available from the Internet (e.g. web servers). NMTD cannot be applied to such hosts since they have to be reachable from external users, thus they are constantly exposed to attacks. While there are techniques to be able to apply NMTD to public-facing devices (e.g. using a load-balancer in front of web servers), their use can be too expensive for a number of devices providing small-scale services.

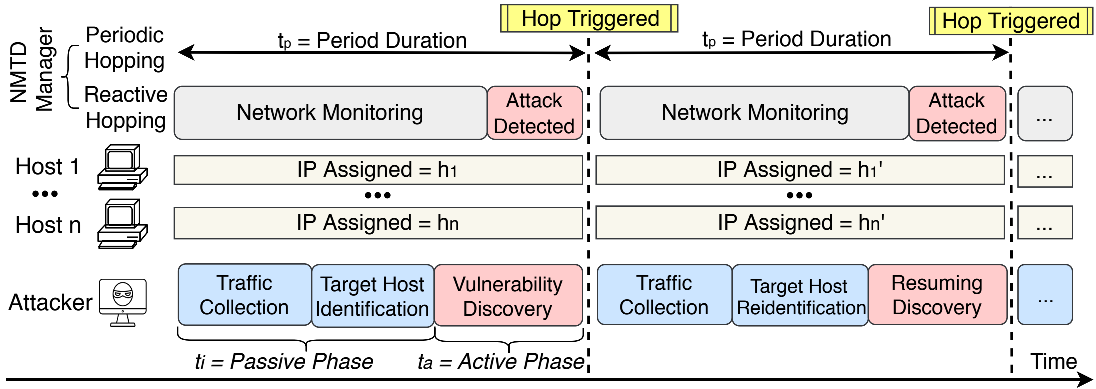


Figure 4.1: Example of a discovery attack against an NMTD adopting periodic or reactive IP address hopping.

An attacker has to avoid detection while reidentifying hosts in a time-efficient manner before the hopping is triggered. For this reason an attacker has to rely on a passive approach which fits these constraints. A suitable candidate for this task is host behaviour profiling: a technique that aims to build a model to describe a host by extracting its behaviour in terms of communication patterns. In this case, an attacker should be able to build a host profile that is robust against the previously mentioned changes made by NMTD.

In Figure 4.1, we provide a possible attack scenario considering both the periodic and reactive hopping strategies. In particular, an attacker first collects and aggregates network traffic in order to find the new hosts' identifiers (as a result of the previous hopping). At this point an attacker can start or continue vulnerability discovery. In the case of periodic hopping, the active attack can continue until the next NMTD hop. Additionally, an attacker can learn when a hop occurred to restart the identification process and resume their attack. In the case of reactive hopping (assuming there is not another attack ongoing), an attacker has enough time to identify hosts, but the time window for an active attack lasts until it is detected and triggers the hopping.

An attacker can use host profiles to infer the role of a host within a network (e.g. an Internet access point, web or mail server), identify hosts with similar behaviour, or track users across different networks according to their network

behaviour. Moreover, an attacker who is able to obtain the traffic logs of a network (even if anonymized) can infer sensitive information about hosts or discover further vulnerabilities.

4.2.1 Threat Model

In our threat model, we consider an attacker whose objective is to find and exploit network devices. The intention of such exploitation is not limited to gain quick access to a resource (e.g. to exfiltrate sensitive data) but it can also include long-term use of a compromised device. Networks that employ NMTD techniques pose a challenge for an adversary as the identifiers of the hosts change periodically. Even if the attacker has already managed to identify or compromise a set of hosts on a network, they still need to reidentify those hosts to continue their malicious activities. The attacker cannot afford to install a backdoor that would communicate back as this would generate anomalous traffic, which could trigger a reactive hop. We assume that an attacker has access to the network flow data of a targeted network. The source of network flows may come from access to a flow collector (which is a device in charge of storing flow data) or even access to a flow exporter (which is a device such as a router that aggregates packets into flows).

Summarizing, a host profiler has to have the following properties to cope with the challenges in this threat model:

(i) Profile a host from its flow-level traffic since it is not feasible to collect packet-level information in a large-scale network due to the amount of traffic involved.

(ii) Extract fingerprints, in an efficient way in terms of time and storage, which effectively represent the network behaviour of a host.

(iii) Accurately identify a host before its identifiers are changed by hopping.

4.3 Related Work

In this section, we present the research work related to our proposal. We first summarize the state of the art of network-based Moving Target Defence strategies in Section 4.3.1. We then present, in Section 4.3.2, an overview of network traffic analyses that range from network application identification to host profiling.

4.3.1 Network-Based Moving Target Defence

In recent years, researchers have proposed security solutions that rely on MTD strategy to cope with attackers on various levels of networking infrastructure that range from the network topology and routing paths to the hosts' operating system, services and file locations [78, 82, 83]. Network-based MTD can be deployed on new cloud-based solutions and software-defined networks [84–86]. An important distinction in NMTD is whether the hopping affects the real or virtual host identifiers.

4.3.1.1 NMTD on Real Host Identifiers

As pioneering work in this research direction, Kewley et al. in [87] propose DYNAT, a system that obfuscates the TCP/IP header of packets in a way that only legitimate hosts can retrieve the original packet header.

Antonatos et al. in [88] propose Network Address Space Randomization (NASR), a technique that aims at tackling hitlist worms. NASR relies on a Dynamic Host Configuration Protocol (DHCP) server that provides IP addresses with a lease time, hence a host is forced to contact the DHCP server for a new IP address when such a lease expires. To cope with the problem of limited IPv4 address space, Dunlop et al. in [89] propose to enlarge the hopping space using IPv6. Other approaches aim at tackling synchronization and active connections dropping problems [90, 91]. Unfortunately, real host identifier hopping introduces a significant network and computational overhead (especially on large networks).

4.3.1.2 NMTD on Virtual Host Identifiers

This NMTD approach performs hopping only on virtual host identifiers and maintains the real ones hidden from outside of the network. Generally, the hopping in this kind of NMTD is performed relying on Software Defined Networking (SDN) [84–86, 92, 93], which is a new networking paradigm that allows dynamic configuration of a network via the division into the control and forwarding functions making networks more efficient and flexible to changes. SDN technology allows for virtual host identifier hopping, avoiding active connection drop and synchronization problems.

Despite the use of SDN capabilities, the resources needed to periodically change virtual host identifiers increase with hopping frequency [94]. For this reason, some proposals [80, 81, 95] reduce the network and computational overheads by triggering hopping only when an attack (e.g. scanning) is detected. Unfortunately, this kind of countermeasure will only be able to detect active attacks. As previously mentioned (in Section 4.2), Hostbuster will not trigger this mechanism since it does not generate any network traffic.

4.3.2 Network Behaviour Analysis

In the last decades, researchers have been investigating techniques to obtain meaningful insights from network traffic. This research field includes analyses on various domains which are influenced by the final goal of the analysis, the format of network traffic traces, and the devices that generate network traffic.

Researchers apply network analysis techniques to achieve different goals which range from traffic classification to anomaly detection [96–98]. The location and format in which network traffic traces are captured lead researchers to make preliminary assumptions for their analyses. As an example, DPI techniques have to be applied on unencrypted packet-level traces [99], while they cannot be applied on more coarse-grained flow-level traces (e.g. IPFIX or NetFlow formats) [56]. In this section, we discuss the analyses that consider hosts' behaviour in terms of network traffic.

4.3.2.1 Anomalies and Malware Detection

Profiling entities according to their network behaviour is also useful for identifying anomalies [1, 100, 101], botnets [37, 38, 41, 102] and other malware activities [32, 49, 103]. Among this work, Minarik et al. in [100] present a method to profile behaviour of network hosts which is based on bidirectional network flow features such as a number of communication peers, amount of traffic, and client and server traffic counts of bidirectional network flows. The proposed method detects anomalies that are the result of a distributed attack. While anomaly detection is used to detect suspicious behaviours, it is not as useful for determining identifiers of a host in a network that uses NMTD.

Favaretto et al. in [101] propose an anomaly detection system based on logs of web services' internal areas. Their system relies on Hidden Markov Models and Continuous Time Markov Chains to build a user behaviour profile on three different layers of logs: accesses, visited pages, and database logs. Unfortunately, due to the private nature of such logs, the authors had to rely on synthetic data to evaluate their models. Moreover, both publications rely on data sources other than network flows, which makes them unfeasible in most modern scenarios.

Allman et al. in [104] present a system to share behaviours of network hosts. This system features a distributed database that aggregates information about each network actor that is updated by contributors of varying reputation. However, the target of this proposal is unsuitable for the method to be adapted to NMTD.

4.3.2.2 Host Profiling

Researchers investigate host profiling to infer hosts' properties, roles and social behaviours from the network traffic perspective. Xu et al. in [105] present a method to build behaviour profiles that is based on clustering. They consider packet headers as their data source, which are converted into network flows. However, they depend on source and destination ports in their grouping, which should not be assumed.

Wei et al. in [106] propose a method to profile and cluster hosts on a network that uses information from headers of packet traces (which makes the method unsuitable for our scenario). Their method uses a Dice-based distance metric to cluster hosts with similar behaviour, relying on features such as unique IP addresses contacted, number of bytes and host's open ports.

Karagiannis et al. in [107] present a method to profile end hosts which builds graph-based structures from network flow data. Unfortunately, their proposal is focused on profiling users and their behaviours, which are governed by different constructs (e.g. NAT), rather than a variety of devices or services.

The work that is most related to our proposal is the one by Verde et al. in [52]. Similarly to our system, they consider network flow-based analysis on a large network. In particular, their target is to identify single users on a network that uses Network Address Translation (NAT). Contrary to our proposal, they use flow sequences to train a set of Hidden Markov Models (HMM) and rely on all available network flow fields. This makes their proposal unsuitable for reidentifying hosts under NMTD assumption, since ports, protocols and IP addresses are subject to hopping. Additionally, NAT does not hop periodically, but is rather based on DHCP lease time.

We propose a tool that extracts a statistical-based fingerprint from aggregated network flows for host behaviour profiling. To the best of our knowledge, we are the first to propose an attack tailored to the NMTD strategy. In fact, we do not rely on any flow field that can be spoofed or which is subject to host identifier hopping. Under the NMTD assumption, our trained model does not need to be retrained, being robust enough to give high classification even after a long period of time.

4.4 Hostbuster Design

We describe the structure of Hostbuster, which consists of: a procedure to build a dataset of fingerprints, and a procedure for using the dataset to classify hosts.

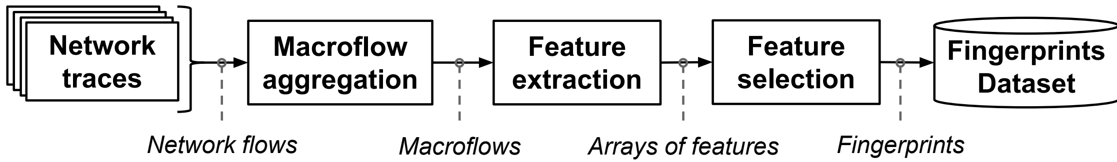


Figure 4.2: Dataset generation procedure.

4.4.1 Fingerprint Generation Procedure

In this section, we describe steps of the procedure to generate fingerprints from network traffic. An outline of this procedure is presented in Figure. 4.2.

4.4.1.1 Flow Aggregation

Since a single flow alone does not contain enough useful information to build a reliable fingerprint of a host's behaviour, we group together flows that contain a host's IP address and occur within the same time frame. This approach aims to extract a pattern of flows that represents a host's behaviour.

Our aggregation procedure selects a set of temporally correlated flows which we refer to as a *macroflow*. Given a host h, a macroflow contains flows where the source or destination IP address is equal to the IP address of host h, and the difference of timestamp values satisfies specific timeout conditions. A macroflow stores flows in a collection sorted by increasing timestamp values. We consider two timeout types:

- *Active timeout* is the time difference between the considered flow and the first flow belonging to the macroflow;
- *Inactive timeout* is the time difference between the considered flow and the last flow belonging to the macroflow.

It is worth noting that in the same macroflow, we consider flows with different port numbers. We evaluate the active timeout with the intention of prioritizing talkative hosts (ones that generate a large volume of traffic). The prioritization means that there will be fewer macroflows; however, each of them will be composed of more

flows which will result in more accurate fingerprints. We also aim to minimize the inactive timeout value to account for possible NMTD changes. For example, if the NMTD is triggered then the inactive timeout should not block macroflow cache (see [56]), which leads to waste of resources, but rather finalize the macroflow creation. At the same time, we aim to minimize the number of extracted macroflows as well as the number of macroflows containing only a single flow.

4.4.1.2 Feature Extraction

As a subsequent step, we extract features from macroflows in a format that could be used to build our model. It is important for our method to define a transformation that aims to preserve the original information while returning a constant-size output for each macroflow. We extract 490 statistical features from each macroflow. We underline that we follow a systematic approach to extracting features, hence we consider a complete set of fields given by a flow, even those that could be changed by NMTD hopping (i.e. ports, protocols and IP addresses).

Given a macroflow, our approach calculates a number of series, each corresponding to a specific flow field or a derived field. The former fields comprise of: number of packets (*packets*), size of packets (*bytes*), duration (*dur*), protocol (*proto*), source IP address (*srcIP*), destination IP address (*destIP*), source port (*srcPort*), and destination port (*destPort*). The latter fields contain a difference between timestamps of consecutive flows (*timing*), and a number of flows in a macroflow (*flowCount*).

For each of the aforementioned series (with the exception of source IP and destination IP), we extract three additional series according to the direction of flows with respect to the IP address based on which the macroflow is created. More precisely, the directions include incoming flows, outgoing flows, and all flows (i.e. incoming and outgoing). We refer to them using prefixes *in*, *out* and *all*, respectively. If the seed IP (i.e. IP address belonging to the local network that the macroflow is built on) is in the role of a source in a flow, then the flow is defined as outgoing.

Likewise, if the seed IP is a destination, then the flow is incoming. For source IP and destination IP we consider only one direction: incoming *srcIP* and outgoing *destIP*. In fact, other directions would not contain any meaningful information as they would only contain the seed IP address (i.e. outgoing *srcIP* or incoming *destIP*) or they would be a superset of respective considered directions with the addition of the seed IP address (i.e. all *srcIP* or all *destIP*). In total, we consider 23 series. For each series, we calculate a vector of statistical features. We define the following two classes that contain them: basic and extended. The basic class comprises: *mean absolute deviation (MAD)*, *minimum*, *maximum*, *mean*, *kurtosis*, *skewness*, *standard deviation (SD)*, *variance*, *quantiles* given in percentiles from 10th to 90th and the count of *unique* values. The extended class contains all basic class features and it enhances them by appending the following: counts of flows for each of the ten most relevant ports; a count of *other ports* that are below 49152 and are not equal to the ones mentioned; and a count of *ephemeral ports* that are in the range from 49152 to 65535. All port counts are then normalized to show the ratio of the frequency of a port to the overall port frequency. A detailed explanation of the most relevant ports is shown in Section 4.5. Fields of given series belong to the following classes:

- basic: *packets, bytes, timing, dur, proto, srcIP, destIP*
- extended: *srcPort, destPort*

The series are further divided into two types based on the nature of its underlying field: numeric and semantic. Numeric fields are used directly to extract statistical features. Semantic fields are considered entities, hence it would not be informative to calculate the difference between them or any other statistical function. For example, ports are described using numbers, which are only meaningful when associated with a specific service. The distance between two ports cannot be reliably interpreted. For this reason, we use one-hot encoding [108] to work with occurrences of underlying field values. Rather than treating raw flow field values as a collection from which features are calculated, it creates a mapping of each value with regards to how

Algorithm 1 Feature extraction from a macroflow.

```
1:  function GET_FEATURES_FROM_MACROFLOW(mf)
2:      features[]
3:      directions ← [incoming, outgoing, all]
4:      for all direction in directions do
5:          flows ← mf.getFlows(direction)
6:          fields ← [packets, bytes, dur, proto, srcIP, destIP,
7:                  srcPort, destPort, timing]
8:          for all field in fields do
9:              if field = srcIP and dir = {out or all} then
10:                 continue
11:             if field = destIP and dir = {in or all} then
12:                 continue
13:             fieldValues ← getFieldValues(flows, field)
14:             if isSemantic(field) then
15:                 fieldValues ← oneHotEncoding(fieldValues)
16:             features.add(getBasicFeatures(fieldValues))
17:             if isExtended(field) then
18:                 features.add(getExtendedFeatures(fieldValues))
19:         features.add(flows.Count)
20:     return features
```

frequently the value is present. Features are then computed using those frequency values. The fields of the series have the following types:

- numeric: *packets, bytes, timing, dur*
- semantic: *proto, srcIP, destIP, srcPort, destPort*

After calculating the statistical features of all fields described in the previous paragraphs, we concatenate them to form an array of 490 features for each macroflow as shown in Algorithm 1.

4.4.1.3 Feature Selection

As a next step, we analyse the significance of each feature with the intention of identifying hosts' behaviours and selecting only the most contributing ones. We aim to limit the computational complexity of extracting final features and to avoid the phenomenon known as the curse of dimensionality. The first step relies on

relative mutual information (RMI) to achieve this task. This metric serves as a measure of informativeness between each of the 490 features and a host's IP address. We formally define RMI as:

$$RMI(X,Y) = \frac{MI(X,Y)}{ME(Y)}$$
$$MI(X,Y) = ME(Y) - CE(Y|X)$$

where MI is mutual information, ME is marginal entropy, CE is conditional entropy, X is a 2-dimensional matrix (rows - hosts' macroflows; columns - the 490 features), and Y is a 1-dimensional matrix of labels in the form of hosts' IP addresses that represent rows in X. The result of the RMI is in the range from 0.0 to 1.0, where the former means that a feature conveys no information about host's behaviours and the latter vice versa.

The second part of reducing the number of features is the computation of a feature correlation matrix using the Pearson correlation coefficient. The features in this matrix are sorted from the highest to lowest RMI score. For each row in the matrix, we remove features that have a correlation value higher than the chosen threshold. The resulting features form a collection from which the final set of features is selected based on the classification procedure with increasing number of top features from this collection. The experiments and ultimate features are presented in Section 4.5.1.3. Once the final set of features is established, the values of those features in each macroflow form a fingerprint that is subsequently used in classification.

4.4.2 Classification Procedure

Hostbuster performs classification in NMTD networks. Before we are able to classify hosts, fingerprints need to be checked for overlap in the given domain (a set of hosts to be profiled in an experiment) and those fingerprints that do not constitute a unique behaviour of a host are relabeled. Once this procedure is finished, we move on to the classification.

4.4.2.1 Ambiguous Fingerprints Identification

The goal of our analysis is to identify a specific host according to its unique behavioural traits. Unfortunately, in some cases, the network behaviour of two or more hosts within the set of considered hosts overlaps, generating samples that are very similar to each other. We label such fingerprints as ambiguous since they do not uniquely identify a host. Ambiguous fingerprints hinder our analysis since they might lead to misclassifications. Moreover, a host may exceptionally deviate from its usual behaviour (e.g. malware attacks, intrusions or other anomalies). In that case, a fingerprint related to this situation would not be useful to our analysis since it does not represent the usual behaviour of the host. In order to cope with these anomalies, we have to identify and relabel such fingerprints as ambiguous in our training set.

In our analysis, we identify ambiguous fingerprints relying on two subprocedures based on the k-Nearest Neighbor (kNN) algorithm. In the first subprocedure, we identify abnormal host behaviour using the Local Outlier Factor (LOF) [109]. Such an algorithm identifies outliers by measuring the distance of fingerprints from their neighborhood. In particular, LOF uses a density-based clustering (i.e. kNN) to identify isolated fingerprints. We define the parameter k of kNN used by LOF as the k_{LOF}. The resulting fingerprints from this subprocedure are relabeled as *outlier* and are not considered in the second subprocedure.

In the second subprocedure, we aim to identify borderline or overlapping fingerprints that belong to different hosts. In this approach, we also use kNN, but differently from LOF, we focus on the labels of a fingerprint's neighbors rather than its distance from them. In particular, we adapt the PReprocessing Instances that Should be Misclassified (PRISM) algorithm [110] to the domain of our analysis. In fact, while we still rely on four heuristics defined in PRISM, we consider both overlapping and border fingerprints as ambiguous and thus unreliable for identifying a host. The most important heuristic proposed by PRISM is called *k-Disagreeing Neighbours* and given a fingerprint it quantifies the rate of nearest neighbors that

are generated by a different host:

$$DN(x) = \frac{|\{n : n \in kNN(x) \wedge host(x) \neq host(n)\}|}{k_{PRISM}}$$

where $host(m)$ is the host that generates a fingerprint m and $kNN(m)$ is the k_{PRISM} fingerprints in the neighborhood of m. This heuristic is used to determine whether a fingerprint x may be border or overlapping. In fact, if $DN(x)$ is higher than a threshold t_{DN}, it means that the neighborhood $kNN(x)$ includes more than $k_{PRISM} \times t_{DN}$ fingerprints generated by one or more hosts that are different from the one that generated x.

Algorithm 2 Ambiguous fingerprint identification and relabeling.

```
 1:                                  ▷ dataset = array of pairs (label, fingerprint);
 2:                                  ▷ k_LOF = number of neighbors for LOF;
 3:                                  ▷ k_PRISM = number of neighbors for PRISM
 4: function IDENTIFY__AMBIGUOUS(dataset, k_LOF, k_PRISM)
 5:                                  ▷ Precomputing kNN
 6:     neighbors ← kNN(dataset, max(k_LOF, k_PRISM))
 7:                                  ▷ Identifying and filtering outliers
 8:     outliers ← LOF(dataset, neighbors, k_LOF)
 9:     dataset_tmp                  ▷ Temporary collection for filtered data
10:     for all datum in dataset do
11:         if datum ∉ outliers then
12:             insert datum into dataset_tmp
13:                                  ▷ Identifying overlapping or borderline fingerprints
14:     overlaps ← PRISM(dataset_tmp, neighbors, k_PRISM)
15:                                  ▷ Relabeling overlapping or borderline fingerprints
16:     for all datum in dataset do
17:         if datum ∈ outliers or datum ∈ overlaps then
18:             datum.label ← Ambiguous
19:     return dataset
```

We apply the whole procedure only on the training set since the PRISM subprocedure relies on information about fingerprints' hosts and we cannot use this information in a testing set. Hence, as a result of the whole procedure, we do not discard such uncertain fingerprints (i.e. outliers, overlapping, and border), but we relabel them as belonging to an additional class *Ambiguous*. By doing this, we are able to first classify ambiguous fingerprints on a testing set and then filter

them out as unreliable. The whole procedure for the identification and relabeling of ambiguous fingerprints is described in Algorithm 2. We underline that for the PRISM subprocedure we precompute kNN with k_{PRISM} and Decision Trees (DT) of other heuristics in [110] since they are used for each fingerprint in the training set. Moreover, we optimize the whole procedure to run on multiple processor cores.

4.4.2.2 Model Training

Given a training set composed of a collection of processed fingerprints, we train a model that is able to classify unseen fingerprints to their respective hosts. In particular, we rely on a random forest algorithm - a machine learning ensemble method that aggregates weak estimators (e.g. decision trees) on a random subset of features (i.e. bag of features) to obtain a stronger classifier. Given a test sample, the random forest classifier provides a classification based on the ensemble consensus among single estimators. The final classifier's decision is obtained by the majority of the estimators' classifications. Although random forest is able to handle multiclass classification, it does not scale well with the increase of the number of classes [111]. To cope with this problem, we use a one-vs-rest method to decrease the complexity of the resulting model. The one-vs-rest technique builds a binary classifier for each targeted class. The final classification is given by the class associated with a binary classifier which has the highest consensus within a set of trained classifiers. It is worth noting that a classifier considers only the given classes it is trained on and extracts this knowledge from the provided training set, i.e. it works under the closed-world assumption. This assumption is suitable for our analysis as we aim to identify specific hosts within a fixed set.

4.5 Evaluation

The proposed contributions are evaluated using a real-world dataset of network activity from the University of Oxford. It comprises network flow data that spans

three months from December 2015 to the end of February 2016. In total there are 119, 853, 493, 260 network flows in a unidirectional format. The data are obtained taking into account ethical considerations. We do not reveal any IP addresses in the dataset, even though the majority are on an internal network.

In this section, we present the experiments we carry out to evaluate our proposal. For each of them, we use a different subset of the real-world dataset, which is dependent on the variety of claims being validated. All of the experiments were run on a computer with 32 logical processor cores (dual Intel Xeon Gold 6134), 512GB of RAM and NVMe SSDs, using Python (with scikit-learn library [112]) and C# programming languages.

4.5.1 Fingerprint Creation

In this section, we analyse in detail the transition from network flows to host fingerprints. Firstly, we study and motivate the choice of timeouts to aggregate flows into macroflows. Secondly, we discuss properties of resulting macroflows. Finally, we study the most expressive features that allow to better identify hosts.

4.5.1.1 Timeout Analysis

Hostbuster relies on aggregation of network flows into macroflows as one of the first steps. Grouping of flows in our procedure depends on active and inactive timeout parameters, which influence the number of flows in a macroflow. Under usual conditions, the inactive timeout value is smaller than the active timeout value. The notion of timeouts is inspired by network flow capture, where timeouts are one of the conditions packets have to fulfill to be assigned to a flow. For example, in Cisco NetFlow the default inactive and active timeouts are 15 seconds and 30 minutes respectively [22]. Another similar approach that relies on timeouts to aggregate network packets into flows has also been used in [4]. Since our scenario is different from the ones mentioned above, we carry out a study using network

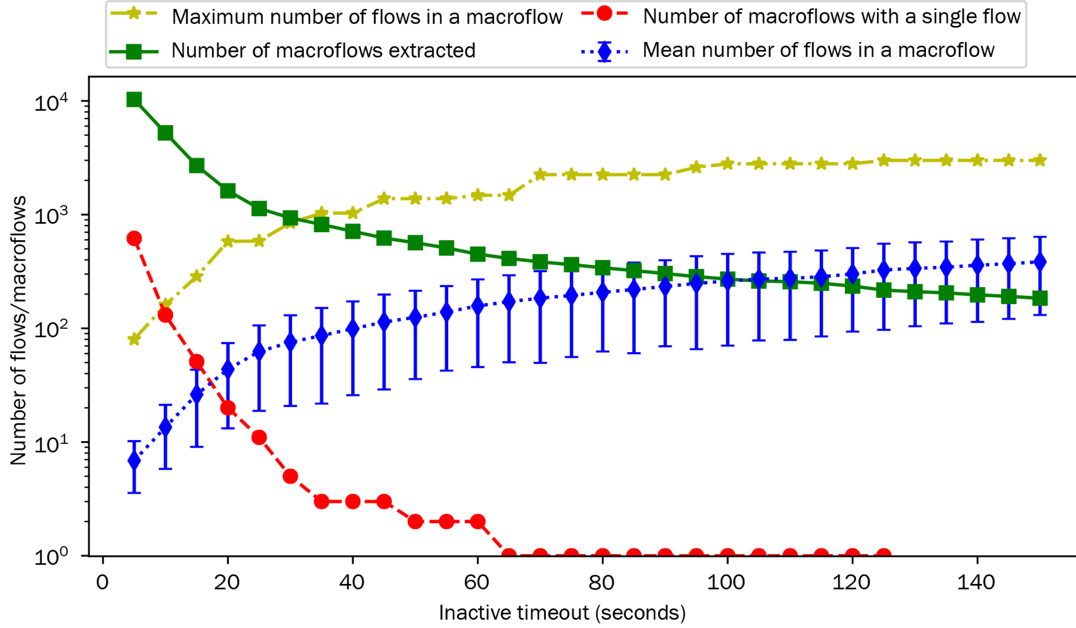


Figure 4.3: Analysis of inactive timeout impact on macroflow extraction with the active timeout of 500 seconds.

flows from one hour of the University of Oxford dataset. It consists of 2,161,630 flows. Considering the active timeout of 500 seconds, we analyse inactive timeout. Based on the results in Figure 4.3, we conclude that the optimal inactive timeout value is 20 seconds. These timeouts are used in the macroflow creation procedure that is shown in Algorithm 3. Since we do not know which hosts we are profiling a priori, we cannot apply variable timeout values to different hosts.

As an additional experiment, we assess that certain macroflows are not suitable for our goal. For example, a macroflow containing only incoming flows is not usable since it does not show any active behaviour of the profiled host. Similarly, macroflows with only one outgoing flow do not contain enough information to constitute a reliable fingerprint. For this reason, we conclude that only macroflows that consist of at least five outgoing flows should be considered in our profiling method.

Algorithm 3 Macroflow creation.

```
1:                                         ▷ aT = active timeout; inT = inactive timeout
2: function GET_MACROFLOWS(file, ip, aT, inT)
3:     matchingFlows[]                           ▷ Sorted collection by timestamp
4:     for all flow in file do
5:         if ip = {flow.srcIP or flow.destIP} then
6:             matchingFlows.add(flow)
7:     mfs[]                        ▷ Macroflows containing a collection of flows
8:     for all f in matchingFlows do
9:         aDiff ← f.t - mfs[last].firstFlow.t
10:        inDiff ← f.t - mfs[last].lastFlow.t
11:        if mfs.size = 0 or aDiff < aT or inDiff > inT then
12:            mfs.add()                                 ▷ Timeouts triggered
13:        mfs[last].flows.add(f)
14:    return mfs
```

4.5.1.2 Macroflow Properties

We evaluate our flow grouping method with selected timeout values to assess its properties using a subset of the dataset that includes $4,949,653,716$ flows and spans 91 days. The chosen days are shown in a set of figures, which depict the variation of considered data in terms of the amount of flows (Figure 4.4) and macroflows (Figure 4.5), as well as how the number of extracted macroflows is influenced by the number of flows (Figure 4.10). The data in the figures are a result of adding flows and macroflows of a set of ten hosts selected for the evaluation that belong to the same subnetwork. We selected those ten hosts from a list of most frequently communicating hosts in our dataset (i.e. hosts that generate the largest amount of flows) while taking into account their role on the network. We also aim to select hosts with diverse roles. Table 4.1 lists selected hosts and their respective roles, which include backbone networks, DNS, web and mail servers as well as a host that corresponds to a personal-use computer.

The main objective of a macroflow is to provide a unit that balances the number of flows by grouping temporally related communications. The frequency of those communications becomes implicitly a behavioural feature and contributes to the profiling. Such a "flattening" of flows is clearly visible in Figure 4.10, where for

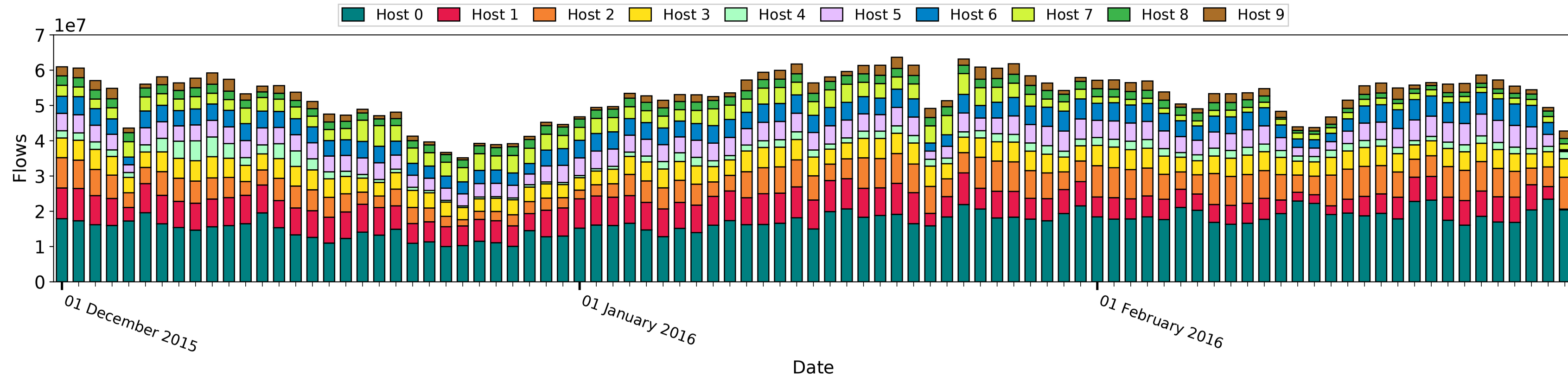


Figure 4.4: Analysis of macroflow properties: distribution of flows.

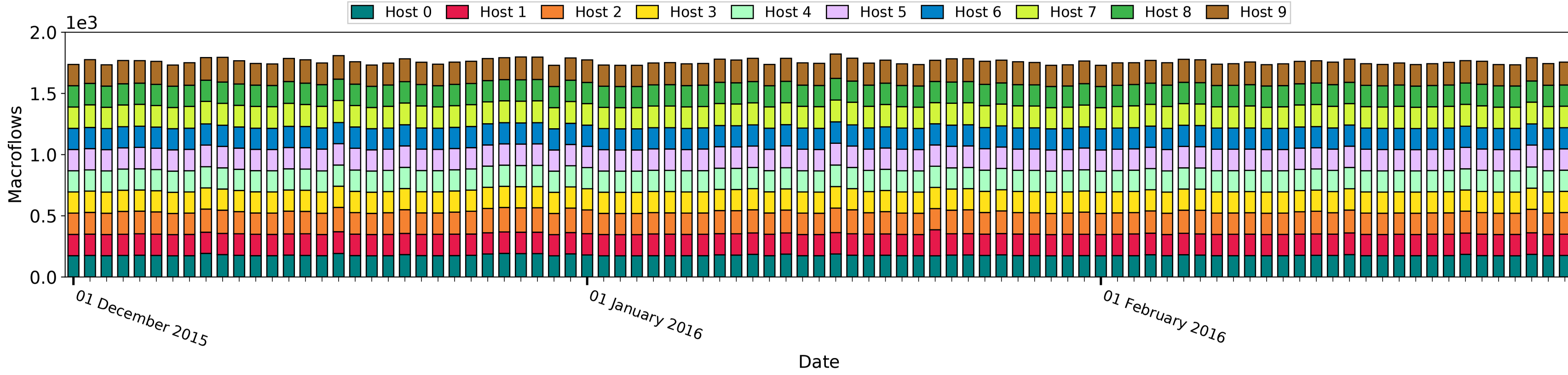


Figure 4.5: Analysis of macroflow properties: distribution of macroflows.

Host Index	Network Function
Host 0	College residential network
Host 1	Departmental backbone network
Host 2	University mail server
Host 3	University DNS server
Host 4	University backbone network
Host 5	Specific departmental computer
Host 6	Number-crunching computer
Host 7	Web server
Host 8	Departmental network
Host 9	Departmental gateway

Table 4.1: Roles of hosts included in the evaluation.

each day we extracted a steady number of macroflows (i.e. around 1750) while the amount of flows varied from 35 to 63 million.

4.5.1.3 Feature Selection

In engineering our features, we include the number of flow occurrences with specific port values. Those values represent the most frequently used ports and are obtained by performing an analysis on over 250 million flows from the real-world dataset, shown in Figure 4.6. We pick the top ten ports and add the count of them to the feature extraction procedure.

As the last step of fingerprint generation, we perform feature selection. Once the features are ranked by RMI, they are used to create a correlation matrix (as described in the Methodology section). Features with highest correlation are filtered out. To do so, we consider 0.9 as the threshold of high correlation having in mind the trade-off between the intention to remove the most similar features while not overly restricting the final number of them. The resulting list of features is presented in Table 4.2. We included a random feature to check the correctness of our feature selection method. As expected, it is one of the last features with only eleven scoring lower (all based on specific port numbers). The total number of features is significantly lower than a total of 490 features that we initially extract due to the

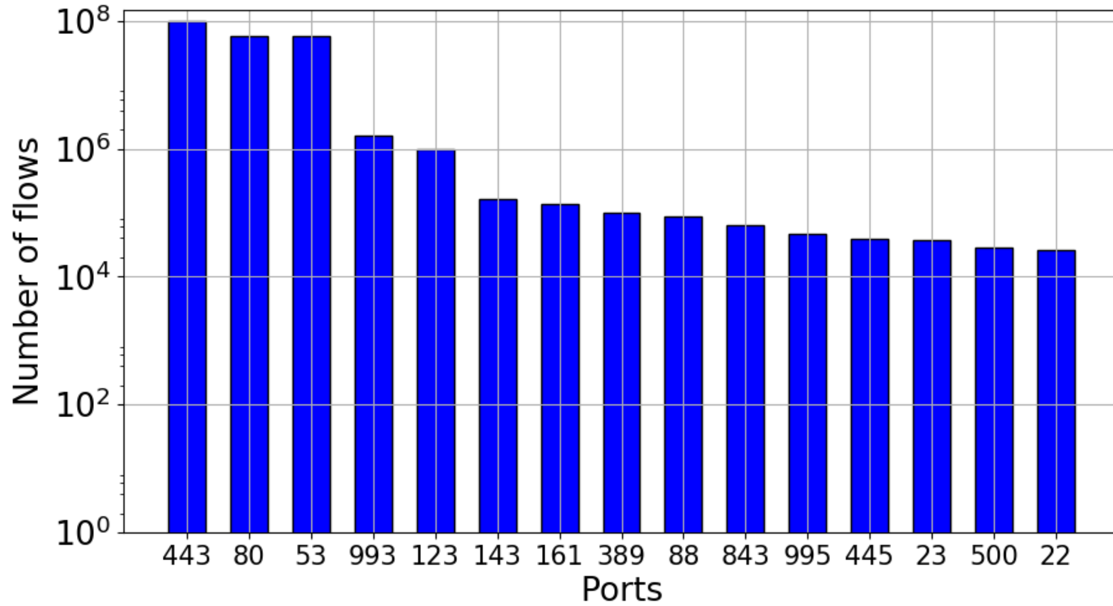


Figure 4.6: Histogram of the most frequently used ports.

remaining ones being filtered by the procedure based on correlation. In order to assess the final set of selected features, we examine the classification performance while increasing the number of top features (Figure 4.7). In the process, we rely on a dataset partitioning method known as cross-validation. Such a technique divides the dataset into k folds having the same number of samples. In our experiments, we stratified the dataset to keep the folds balanced, assuming fingerprints as the most basic unit and hosts as labels. We consider five folds. For each iteration, to train a model we apply the classification procedure described in Section 4.4: ambiguous fingerprint identification and a random forest classifier using the one-vs-rest strategy. For testing, we consider a fifth of the dataset for each iteration as given by the fold. It is worth noting that the testing set contains only original labels that represent each class. After receiving predictions from the trained model, a number of true labels may be assigned to an ambiguous class (that is added by the ambiguous fingerprints identification subprocedure in the training phase). In the performance computation, we do not consider classifications that belong to the ambiguous class. Using the remaining classifications, we calculate classification performance metrics: accuracy, precision, recall, and F1 score. Since our task is a multiclass classification

Rank	Direction	Field	Feature	RMI Value
1	**all**	**bytes**	**max**	**0.22623333**
2	**in**	**bytes**	**max**	**0.21370440**
3	**out**	**bytes**	**max**	**0.20414153**
4	**all**	**bytes**	**quantile 0.9**	**0.19154731**
5	**all**	**bytes**	**quantile 0.6**	**0.18588126**
6	**out**	**bytes**	**quantile 0.5**	**0.17665746**
7	**out**	**bytes**	**quantile 0.9**	**0.17489754**
8	**in**	**bytes**	**quantile 0.9**	**0.17268278**
9	**out**	**bytes**	**quantile 0.2**	**0.17245904**
10	**in**	**bytes**	**quantile 0.1**	**0.15946455**
11	all	bytes	min	0.14234995
12	all	bytes	SD	0.13166923
13	out	packets	max	0.12821883
14	out	packets	mean	0.12514600
15	out	packets	min	0.11005842
...	...	...	...	...
181	n/a	n/a	random	0.00016596

Table 4.2: Features with highest RMI values after filtering the ones with highest correlation (selected features in bold).

(that uses a joint decision of binary classifiers given by the one-vs-rest strategy), we compute the above-mentioned metrics using a weighted average of samples. The experiment presented in Figure 4.7 is carried out by using a subset of the real-world dataset. It is obtained by randomly picking a set of 25 unique hosts 25 times, which in total results in 625 hosts. Each data point is computed by taking an average of the values in each iteration. From this analysis, we pick ten top features as the final ones, which are shown in bold in Table 4.2. Even though there exist a higher number of features for which the score is higher in terms of the F1 score, the difference is less than 2 percentage points while adding an additional feature results in a performance penalty. For this reason, we opt for the most balanced approach. A macroflow with only those 10 selected features constitutes a fingerprint. It is worth emphasizing that we perform the procedures described in this subsection only once. We then use them to profile a variety of hosts in all experiments. The suggested parameters can be used to apply Hostbuster on a number of hosts from multiple domains.

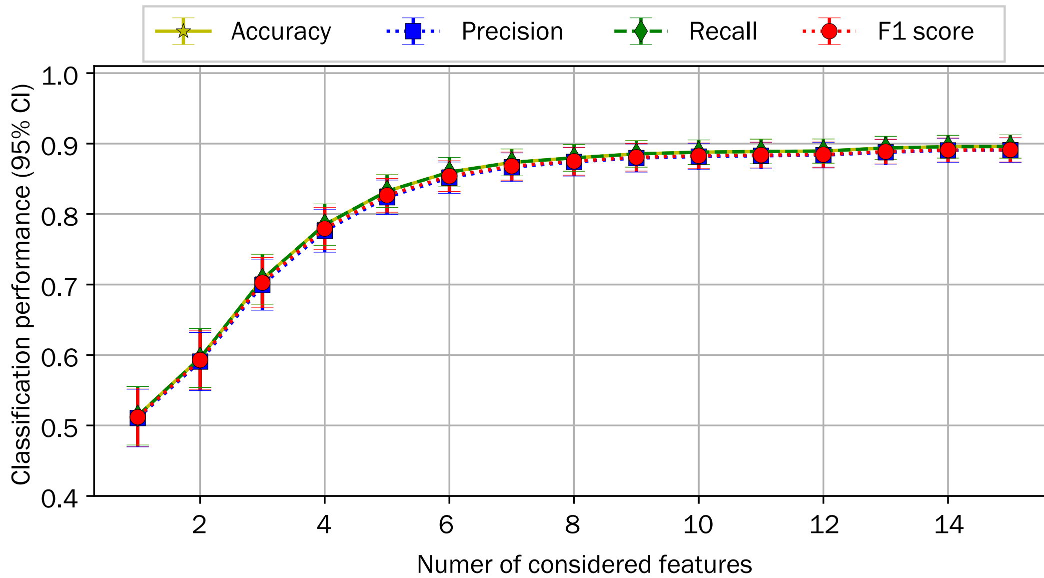


Figure 4.7: Impact of the number of considered features on classification performance with 95% confidence intervals.

4.5.2 Experimental Results

In this section, we present the result of our experiments. We evaluate our proposal from several points of view. We first discuss the impact of the ambiguous fingerprints identification procedure on the classification performance. Then, we evaluate our proposal varying the number of targeted hosts. Finally, we analyse the robustness of a model over an extensive period of time. While in this section we report the results of the experiments, we provide an in-depth discussion of the results in Section 4.6.

4.5.2.1 Ambiguous Fingerprints Identification

Relabeling fingerprints is an important step that increases the performance of our method. In fact, we notice that some fingerprints represent behaviours that can be shared or can deviate from typical ones. These cases are not useful for uniquely identifying a host, hence we relabel them as ambiguous. Particularly, in this domain LOF and PRISM allow us to identify ambiguous fingerprints.

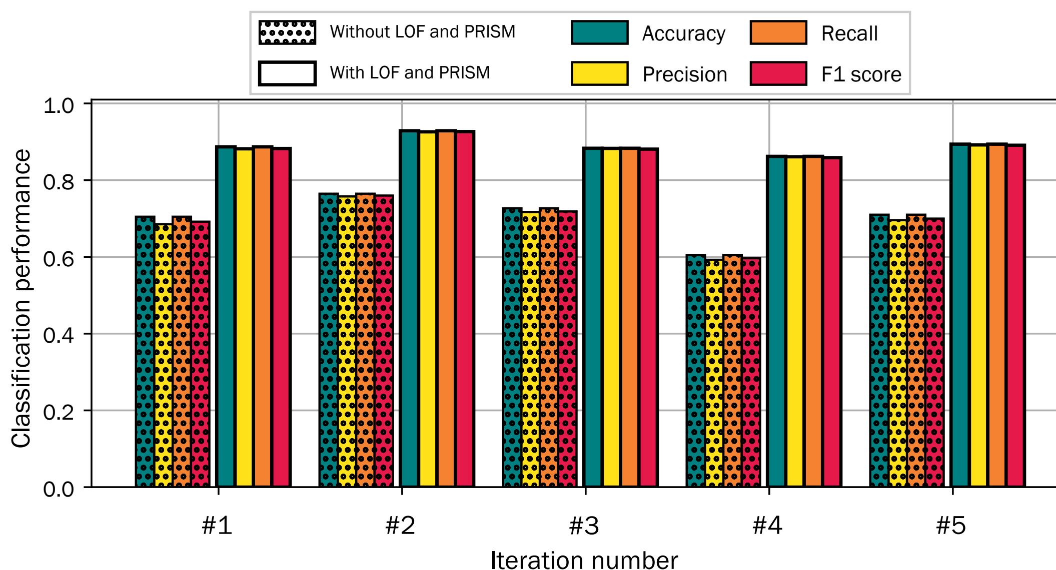


Figure 4.8: Performance comparison of preprocessing with and without ambiguous fingerprint identification.

In this experiment, we consider 5 iterations of 25 randomly selected hosts, which results in a total of 125 hosts. We apply the ambiguous fingerprint identification procedure using the following set of parameters. For the identification of outliers via LOF we use $k = 5$ and Minkowski distance metric [113]. Regarding the PRISM algorithm, we conducted a preliminary analysis on the number of k and we assessed that a reasonable trade-off between a number of ambiguous flows and a number of valid flows is 17 as given by [110]. We consider the given default values for the other parameters. Similarly to the analysis in Section 4.5.1.3, we study the improvement of performance metrics by applying ambiguous fingerprint identification.

As we can observe in Figure 4.8, applying LOF and PRISM improves the performance by 18 percentage points on average. This comes at a cost of relabeling 62% of fingerprints as ambiguous, thus is not useful for correctly classifying a host. In other words, we train our model considering around 1 out of 3 fingerprints as valid for classification.

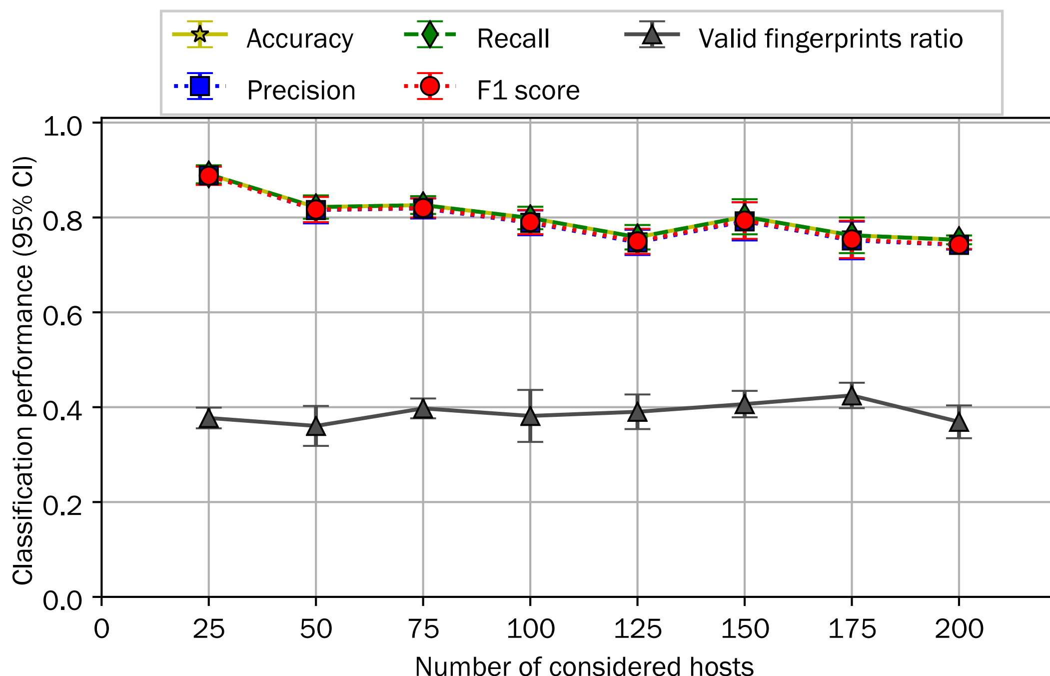


Figure 4.9: Performance comparison varying the number of hosts.

4.5.2.2 Host Behaviour Profiling

In order to assess its robustness, we evaluate Hostbuster using an extensive number of hosts from our dataset. To do so, we run our analysis on a different number of hosts N in a range from 25 to 200 with an increment of 25 considering a dataset of five days of network traffic. For each N, we randomly chose five sets of N hosts and we average the classification results. In total there are 4500 hosts selected. In this experiment, we apply the same methodology as in Section 4.5.1.3, i.e. a selection of 10 features, 5-fold cross-validation using a set of binary random forest classifiers (one-vs-rest).

We report the results of this analysis in Figure 4.9. As we can observe, the classification performance drops minimally with the increase of hosts. Despite the hosts being picked randomly, we can also notice that the variation in classification performance within the same number of considered hosts is negligible. Moreover, the ratio of valid fingerprints (not ambiguous) remains constant regardless of the number of considered hosts.

4.5.2.3 Long-Term Analysis

In this analysis, we investigate how the classification performance of a trained classifier changes as the time from training progresses. To do so, we perform an evaluation on three months of the real-world network traffic generated by 25 randomly selected hosts. In particular, we train a set of random forest classifiers (using one-vs-rest strategy) using data extracted from five days and test them on the remaining 86 days.

Based on the performance metrics shown in Figure 4.11, we can conclude that Hostbuster is able to build robust network host profiles that persist with time. In fact, it can reidentify a host even after almost three months from training. It is worth observing that with only five days of training the classification performance follows the ratio of valid fingerprints. Based on this observation, we can assess the expected classification performance by only looking at the valid fingerprints ratio. This means that we are not forced to constantly retrain our classifier, which is desirable under our threat model.

4.5.2.4 Computational Efficiency

We also assess the efficiency of our tool in terms of computational power required and processing time. To do so, we perform the long-term analysis from Section 4.5.2.3 on a single-board computer - Raspberry Pi 3 Model B+, which has a 4-core 1.4GHz CPU and 1GB of RAM. The ambiguous fingerprint identification and model training phases take 24 and 13 minutes respectively. The testing of one day's worth of fingerprints takes 8 seconds on average. Summarizing, a Raspberry Pi is able to compute the entire long-term analysis in under 50 minutes. This proves that a high-end computer is not required by an attacker to use our host profiling tool.

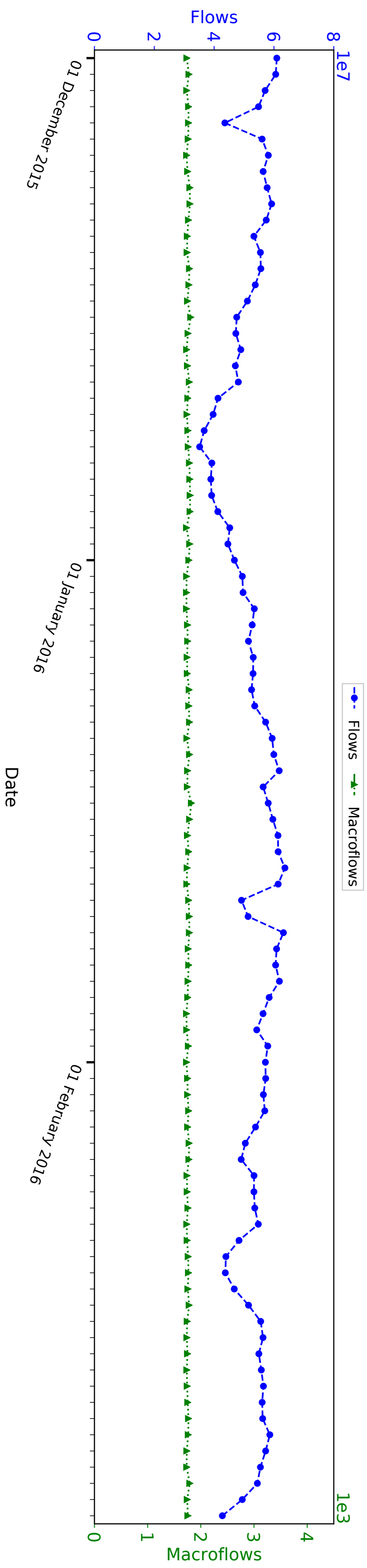


Figure 4.10: Analysis of macroflow properties: number of flows and macroflows.

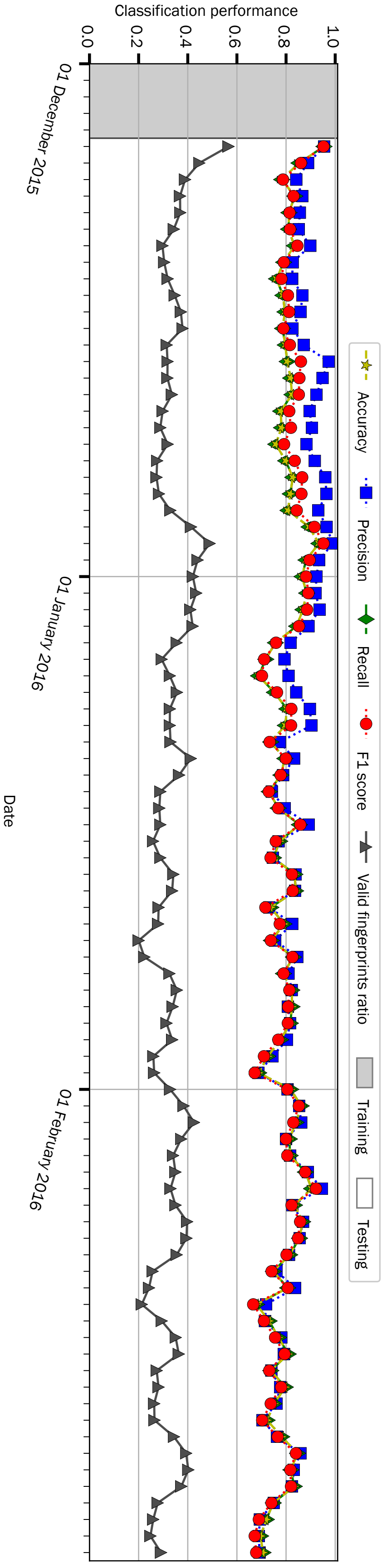


Figure 4.11: Long-term analysis on 25 randomly selected hosts.

4.6 Discussion

Hostbuster profiles hosts according to their network behaviour. In our threat model, we assume that an adversary is able to collect network flows of a targeted network. We also make an assumption that the collected flows come from an exporter or collector. This is the case because collecting and storing packet-level traffic is not feasible due to the staggering amount of traffic involved.

Once an attacker has access to network flow data, they can group flows that involve the same host within a time interval. As shown in Section 4.5.1.2, our system generates an approximately constant number of macroflows regardless of the amount of flows that a host generates. This allows us to work with containers of data, a number of which is invariant to an amount of flows. Such a property combined with statistical feature extraction and selection results in host fingerprints which have the following advantages:

(i) Comprise of only 10 features.

(ii) Are based only on bytes, and are thus immune to IP address, port and protocol hopping which are adopted by NMTD.

(iii) Can be extracted linearly as our method relies on the single flow field instead of those we initially considered.

As an additional use case, network administrators can integrate their security systems with Hostbuster to monitor the behaviour of hosts and discover anomalies that are not detected using traditional NIDS. In fact, our host profiling solution can assess whether a host's network traffic diverges from its typical behaviour. Moreover, from a privacy perspective Hostbuster does not depend on the value of an IP address of a profiled host. Network flows can be used in a format where IP addresses are obfuscated using anonymization techniques such as hashing, as long as each IP address corresponds to one hash value and there are no collisions. An additional desirable property is the use of methods that preserve the prefix of IP addresses (such as [114]), which allows to distinguish between subnetworks.

Hostbuster is evaluated on real-world network flow data from the University of Oxford. The dataset is representative enough of an enterprise network as it contains hundreds of active hosts which communicate using different services. There are a number of web, DNS and mail servers that are characteristic of most large companies, as well as VPN servers and specific mission-critical network segments. A limitation with regards to the nature of Oxford dataset might be the implicit diversity of users on the network which could translate to their different network behaviours and ultimately lead to increased classification performance of the method. The devices are spread between departments and colleges within the university. In a regular company the behaviours might be more homologous due to their more targeted area of specialty.

4.6.1 Limitations

In our analysis we have to make the following assumptions. In order to perform robust host profiling, a model needs to be trained on a sufficient number of samples. Not all extracted macroflows can be considered in the classification. Only those macroflows are taken into consideration that contain at least five outgoing flows. This limits our method to only those hosts that generate enough traffic per time period.

Once the model has been trained with hosts' fingerprints, our method requires time to observe and collect network traffic, assuming that host identifiers remain the same in the meantime. In the case of hopping, a host identifier will change and our host identification procedure has to start from scratch. In particular, the part of our proposal that requires more time is the network flows collection within the active and inactive timeouts. The fingerprint generation and classification require a negligible amount of time when compared with such timeouts.

Another limitation is the number of valid fingerprints required to perform classification. Based on our experiments in Section 4.5.2, 38% fingerprints on average can contribute towards host profiling. While this number is not surprising since the majority of hosts' traffic should not represent unique behaviours, it still poses a further limitation on the amount of traffic that a host has to generate.

4.6.2 Evasion

In the light of arguments in previous sections, we discuss possible countermeasures to our attack. Since our host profiler needs a period of time to collect valid fingerprints, one could perform frequent host identifier hopping. However, because we are immune to port or protocol changes, the identifier involved in hopping should be only IP address. Another possible countermeasure is based on the observation that the fingerprints are based on the amount of data transmitted. One can change the Maximum Transmission Unit (MTU) parameter to affect the bytes flow field. In a different approach, one can deliberately send bogus packets to artificially inflate the total amount of bytes. However, these techniques can affect the overall performance of a network by introducing additional latency and consuming more bandwidth.

As an additional method to evade our attack, a host using a Virtual Private Network (VPN) can greatly reduce the probability for Hostbuster to extract valid fingerprints. This is currently possible to be used with NMTD systems with virtual host identifier hopping as shown in [84]. On the other hand, if the attacker knows this, they can concentrate on the endpoint (VPN server) and look for the recipient of the traffic (VPN client) and then correlate the new IP address with the one before hopping happened. This holds only when one hosts uses VPN in a network. When more hosts use VPN, the search space is already minimized and an attacker can extract the fingerprints of only those hosts and apply Hostbuster to detect which one is which.

4.7 NMTD Guidelines

In this section, we provide countermeasures to cope with a passive attacker that conducts a host profiling attack such as Hostbuster. They serve as guidelines to improving the settings in the deployment of NMTD.

In the case of the periodic hopping strategy, an attacker has a limited amount of time to carry out the passive phase (i.e. traffic collection and identification) followed by the active phase (i.e. vulnerability scanning or host exploitation), as

exemplified in Figure 4.1. In particular, the active phase does not trigger a hop, thus it can last until the next scheduled hop. Regarding the passive phase, our host profiler considers macroflows that include flows within a time span equal to the active timeout (i.e. 500 seconds). For this reason, our approach does not work when hopping frequency is higher than seven times per hour, which is the strategy that should be adopted to strengthen NMTD deployments. Nevertheless, since frequent hopping requires a significant amount of bandwidth and computational resources, recent work on NMTD proposes coping with this problem by triggering host identifiers hopping only in case of an attack (i.e. reactive NMTD), such as Denial of Service (DoS) or network scanning. However, an NMTD that only applies the reactive hopping strategy is not secure against host profiling since, without triggering a hop, an attacker would have a long period of time with fixed host identifiers. During this time, they can improve host identification accuracy or even update their model. Since the aforementioned strategies alone do not suffice to counter possible attacks, we recommend the use of a hybrid approach that is a combination of high-frequency periodic hopping and reactive hopping. Given our host profiler, the hopping period should be set at a minimum of eight times per hour. As an optional feature, such a period can be considered to be random (considering the above recommendations) to be less predictable for an attacker.

Another proposed guideline is concerned with identifiers involved in hopping. Changing the IP addresses alone would still allow the attacker to perform vulnerability scanning, while resuming it in-between hops. In order to limit their abilities, one should consider additionally changing port numbers. This will force an attacker to start the active phase anew every time a hop happens.

Summarizing, we suggest that an effective NMTD strategy has to consider a hybrid NMTD strategy with:

(i) A frequent, periodically triggered hopping (lower than our active timeout).

(ii) Hopping triggered when an attack has been detected (e.g. port scanning, DDoS).

(iii) Diverse identifiers changed by hopping (i.e. IP addresses and ports).

Currently the most valuable approach to implementing our guidelines is the use of virtual identifier hopping that can be performed with an SDN in order to limit the management effort. However, the main shortcoming of an SDN network is the cost of OpenFlow-enabled switches. In fact, SDN-based NMTD experiments in the literature are mostly done using simulation software (e.g. Mininet, OMNet++), a single switch, or the GENI testbed [115].

4.8 Summary

In this chapter, we evaluated state-of-the-art NMTD hopping strategies in terms of their susceptibility to host profiling attacks. We designed a proof-of-concept tool that identifies hosts based on their network traffic. Moreover, our proposal only relies on patterns of bytes exchanged between hosts, which additionally makes it resilient against port and protocol hopping.

We evaluated our proposal in a thorough set of experiments to underline its effectiveness in identifying a host and its robustness over the course of three months. The performance remains comparable even considering up to 200 hosts. To the best of our knowledge, Hostbuster is the first proposal that tackles Network-based Moving Target Defence by reidentifying hosts using passive flow-level network traffic monitoring. Given the results of our tool, we suggest adopting a hybrid NMTD strategy that consists of periodic hopping of at least eight times an hour along with reactive hopping. Moreover, we recommend to complement IP address hopping with port numbers.

As future work, we are going to further improve our approach to reduce the amount of data necessary for training and building a host profile, in order to assess an optimal periodic hopping duration as a trade-off between security and resource consumption.

5
Malware Detection in Large-Scale Network Traffic

Contents

5.1 Introduction and Background 74
5.1.1 Contributions 76
5.2 Related Work 77
5.3 Methodology 79
5.3.1 Flowset Creation 79
5.3.2 Feature Engineering 81
5.3.2.1 Feature Extraction 81
5.3.2.2 Feature Selection 82
5.3.3 Malware Detection 83
5.4 Evaluation 84
5.4.1 Dataset Specification 84
5.4.2 Malware Type Discrimination 89
5.4.3 Suspicious Traffic Identification 94
5.5 Discussion 94
5.6 Conclusion 98

In recent years, we witness the spreading of a wide variety of malware. Such software uses Internet connections to steal private information, harm the integrity of systems and infect new victims. Malware targets not only online services, but also traditional personal computer and mobile users. Detecting malicious software has become infeasible on a packet-by-packet basis since the rapid growth of traffic

in the last years. In order to cope with these shortcomings, security analysts need reliable techniques which are able to carry out this task on the Internet scale.

In this chapter, we address this challenge by investigating malware behaviours and designing a method to detect them relying only on network flow-level data. In our analysis we identify malware types with regards to the way they achieve their malicious purposes and their impact on a network. Leveraging this knowledge, we propose a machine learning-based and privacy-preserving method to detect malware on large-scale networks. We evaluate our results on two malware datasets (MalRec and CTU-13) containing traffic of over 65,000 malware samples as well as one month of network traffic from the University of Oxford containing over 23 billion flows. We show that despite the coarse-grained information provided by network flows and the imbalance between legitimate and malicious traffic, MalAlert can distinguish between different types of malware with the F1 score of 90%.

5.1 Introduction and Background

Many malware detection methods were proposed that aim to extract network traffic behaviours and consequently infer (implicitly) the maliciousness of their origin by applying packet-level techniques, such as examining headers or contents of packets. Using these techniques, researchers are able to extract significant insights from packet ports or payloads and identify suspicious traffic by matching it against well-known malware signatures. However, this is not feasible when ports are spoofed [116] or packets are encrypted [47], which is a popular practice nowadays, as mentioned in Section 2.3.1. In addition to that, auditing packet-level network traffic in enterprise-level settings is demanding in terms of required resources. This is caused by a vast volume of traffic among hosts on such networks. To alleviate this problem, researchers investigated techniques that extract statistical data from constructs that infer information from sequences of packets that are considered as time-series, such as network flows (i.e. logical groupings of packets that share protocol; and respective port and IP address pairs). As a result of this, it is possible

to keep track of traffic in terms of aggregations of number of packets and packets' sizes per flow, which also saves a significant amount of storage space. The added benefit of flows is its privacy-preserving nature due to the lack of information about what has been sent (payload). While these are clear advantages of using flow-level data, there also are a few negative aspects. Contrary to packet-level data that contain abundance of information from metadata of each layer of the protocol stack, which allows for a fine-grained analysis, flow-level analysis is more limited due to its scarcity of considered data. Consequently, malware detection using such restricted data is more challenging. It is a trade-off between the total space taken by the representation of network traffic and the amount of information within it. Nevertheless, network flows make traffic capture feasible in real-world scenarios and despite their constraints it is possible to detect malware using them.

Network traffic analysis based on network flows allowed researchers to conceive methods that take into account only statistical information about network traffic (i.e. which hosts communicated, on which ports, how long, how much information was exchanged). It is a common practice that the first step of these methods consists of grouping network flows to form a higher level of abstraction. The aim of this is to be able to extract features from such groupings, which could not be meaningful enough on a single flow basis. The subsequent steps depend on the malware types that a method targets, which include SSH compromise detection [25, 26], stealth malware detection [31], detection and impact of sampling on port scans [24] and botnet detection, which are discussed in more detail in Section 5.2.

In enterprise settings, it is important to be able to deal with the amount of traffic that is generated. Due to its sheer number, it is not possible to perform malware detection instantly in a one-step fashion. While network flows solve the problem of traffic capture, inputting them to malware detection methods leads to results that are not definitive. Such results have to be treated as a coarse-grained alert, which shows in what direction to continue to obtain more details about the attack or detected malware [117]. This is especially important for Security Operations Centers (SOC), which handle traffic from multiple companies, where such alerts show which

more fine-grained source of information (e.g. security logs of concrete hosts) to consult to gain insight about the intrusion. Another aspect important for SOCs is the amount of false positive alerts, which increases by the amount of traffic processed.

In this chapter, we propose *MalAlert*, a system that identifies malware in flow-level network traffic. MalAlert is capable of distinguishing between different types of malware in its detection process, such as adware, ransomware, viruses, worms, trojans and botnets. To do so, MalAlert first aggregates network flows based on both communicating IP addresses. Such aggregations are referred to as *flowsets*. Subsequently, from network flow fields (i.e. data captured in a flow, e.g. duration or source port) we extract a total of 441 statistical features in each flowset, and divide them into groups. We use the relative mutual information (RMI) metric to assess the most representative features. The top ones are then selected to form a feature vector for a flowset that is referred to as a *fingerprint*. Fingerprints are used by the random forest classifier to predict which flowsets contain traffic generated by one of the aforementioned malware types.

5.1.1 Contributions

MalAlert makes the following contributions:

(i) We present an effective approach to aggregate network flows into flowsets and extract statistical fingerprints that preserve valuable information within those flowsets, with the specific aim of detecting malware. Such a fingerprint is composed of statistical features that are, as shown by our results, based only on a number of bytes transmitted. Thus, our fingerprinting is IP address- and port-agnostic which preserves the privacy of users and makes it more robust against port spoofing used by modern malware. The fingerprints extracted in the evaluation are also network quality- and throughput-invariant since they do not rely on features related to the time domain such as flow inter-arrival time or duration. Moreover, our approach is able to aggregate even thousands

of network flows into a single flowset, while preserving information that they provide with regards to their maliciousness.

(ii) We propose a system that relies on statistical fingerprints of flowsets to identify potential malware traffic on a large-scale, real-world network. MalAlert identifies the most reliable statistical features to discriminate between legitimate and malicious traffic. In order to do so, we also take into account the impact in terms of memory and computational cost. From the result of our analysis, we can reach the F1 score of as high as 0.94 only relying on fingerprints of five statistical features. As an application of our proposed system, we identify 0.11% suspicious flowsets, from the 23-billion-flow University of Oxford dataset, that are classified as malicious.

5.2 Related Work

Malware is a relevant threat to computer and network security. Malware developers and security researchers are engaged in a battle of wits that has been lasting for several decades. In order to shed light on such threats, researchers propose several methods to outline the categories of malware according to its goals and behaviours [118, 119]. Moreover, many researchers focus on identifying the presence of malware leveraging their behaviour in terms of network traffic. Since malware programs rely on network communications to a varying degree, only certain types of malware can be robustly detected by analyzing their network traffic. In particular, the vast majority of researchers aim to discover and analyse botnets, whose operation and expansion depend on connectivity to the Internet. In what follows, we present an overview of proposed malware detection techniques using network traffic from the literature.

Botnets are malicious networks coordinated through command and control (C&C) channels that are commonly used to carry out attacks e.g. Distributed Denial of Service (DDoS). Researchers focus their efforts to detect the existence of such networks in order to prevent attacks before they happen. Antonakakis

et al. in [102] presented an extensive study on the properties and behaviour of a Mirai botnet. This botnet targets mobile devices that belong to the paradigm of Internet of Things (e.g. IP camera, VOIP phone, video recorder) and turn them into bots exploiting their poor security measures. This study offers a solid base to understand the behaviour of a botnet in the IoT era.

In the literature, the topic of detecting a wide range of malware with regards to their type by a single method has not been covered as extensively as botnet detection, which is shown in Section 3.3.2. Bartos et al. in [32] propose one of the most relevant and similar methods to our approach. However, the authors consider web proxy logs and not network flow traffic, which allows them to extract more fine-grained features at the expense of rarer availability of proxy log data. Perdisci et al. [120] propose a malware detection system that extracts network signatures of malware based on their HTTP traffic. This system relies on a clustering algorithm to group similar malicious network behaviours. Similarly, Rafique et al. in [121] propose another clustering-based malware detection method that inspects HTTP traffic. However, both these systems do not work when packets content is not accessible and malware uses spoofed and not standardized port numbers to communicate. Alahmadi et al. in [103] present MalClassifier, a system to automatically classify malware families based on their network traffic. This system identifies distinctive malware network flows and it measures the similarity between a sample and such distinctive flows using an ad hoc metric. Another similar method, named CHATTER, is proposed by Mohaisen et al. in [122]. Compared with MalClassifier, CHATTER requires an even more fine-grained analysis of packets since it needs to use DPI to extract HTTP requests. While both these methods build behaviour profiles of malware families, they heavily rely on the order of packets arrival which depends on the quality of the network, thus rendering those methods less robust in real-world environments.

Comparing our system with the state of the art, we can identify several elements of novelty. When our system receives network traffic, one of the first steps it performs is grouping of flows. In the past different strategies were used for this task [31]. One such strategy is to group flows that occur in a specified time window [123].

Additionally, one might consider rules that flows have to fulfill to be part of a grouping. Other methods consider them to be the IP address of a server [41] or both IP addresses constituting a flow [32]. Similarly to the latter, we group flows based on both source and destination IP addresses, but regardless of their role (i.e. server or client) or used ports. We refer to such groupings as *flowsets*. Instead of dividing the time domain into fixed periods and then grouping flows for each of those periods, our approach uses the timeout value to indicate which flows are part of a flowset. While different strategies have been used to extract knowledge from aggregated network flows, we are the first to rely only on information provided by data features of network flows (i.e. amount of sent bytes) to detect and discriminate between multiple malware types. For a detailed comparative analysis of related work with regards to MalAlert the reader is referred to Table 6.2.

5.3 Methodology

In this section, we discuss the methodology that we use in our analyses. First, we explain in detail the concept of a flowset and the procedure to build it by aggregating network flows. Then, we describe how features are extracted and narrowed down to form a final fingerprint from a flowset. Finally, we present our method to identify network traffic generated by malware through a classifier and we explain how the classification analysis is carried out.

5.3.1 Flowset Creation

In this chapter, we increase the level of abstraction further by moving from flows to a new concept of a flowset. It is an aggregation of all network flows that are exchanged between two hosts (identified by their IP addresses) within a specific interval of time. Such an interval of time is determined by a *timeout* parameter, which specifies how long of a period a flowset covers. The timeout value is applied when attempting to add a flow to a flowset. Timestamp of the first flow in a flowset

is subtracted from a candidate flow's timestamp and if the value is smaller than the timeout, the candidate flow is added to that flowset. A flowset contains flows with the same IP address pair irrespective of which one is the source or destination. For example, if host A is the source and host B is the destination in a flow, and in the second flow host B is the source and host A is the destination, both of these flows are put in the same flowset provided that they satisfy the timeout criterion. For convenience, we put first the IP address that belongs to the subnetwork that is contained in the considered dataset, that is assuming that the dataset was collected from the edge gateway, which means that a flow is between local and remote IP addresses. It is also worth noting that flows with different source or destination ports are still considered in the same flowset.

Flowsets are constructed in a way that allows capturing malicious traffic and grouping them together even in longer periods of time, which is given by the selected timeout value. If a flowset was aggregated based on one IP address only, the malicious flows of attacks that generate traffic less frequently could be "lost" in the dominating amount of legitimate flows. Even though it is common in related work that groupings containing fewer than a specified number of flows (e.g. less than five flows [32]) are not considered, our method poses no such limits. Several malicious behaviours produce only one or two network flows. As an example, an infected host notifies the C&C that it is ready to be remotely controlled (i.e. callback phase) [124], producing a single flow. As another example, a worm that downloads a malicious payload produces one network flow. Other approaches that filter out groupings of flows that are less than some given threshold, would not consider these scenarios, which in the end could be vital for successful malware detection and identification.

In order to set a reasonable value for the flowset timeout, we consider two aspects: the typical malware behaviour and the very nature of our flowset definition. It is known that malware often stays “silent” for a certain period of time before it starts its malicious behaviour [124]. Thus, having a short timeout we would risk to not capture those activities in the same flowset. Due to our definition of a flowset there are more such groupings (because our grouping condition is based on both

IP addresses), therefore we have to have a longer timeout value to maximize the number of flows per flowset.

5.3.2 Feature Engineering

Feature engineering is composed of two subtasks: feature extraction and feature selection. While the former aims to generate a vector of features that can represent a flowset, the latter identifies a subset of features that are considered as the most representative for malware classification, which form a fingerprint.

5.3.2.1 Feature Extraction

Features extracted directly from a group of flows, that belong to a flowset, are not sufficiently informative to make such flowset distinguishable from others. In order to do so, a flowset has to be converted to a fixed-length collection. MalAlert solves this by extracting 441 features that are calculated from the flows in each flowset. Those features are divided into logical groups based on the flow fields they are derived from: *time*, *port* and *data* feature groups. The time group consists of inter-arrival time of flows (the difference between consecutive flows' timestamps) and duration flow field. The port group includes protocol (e.g. 6 which translates to TCP), source and destination port (e.g. 80 which translates to HTTP) flow fields. The data group contains the number of flows in a flowset and the number and size of packets flow fields. In each feature group, a flow field-based feature corresponds to a number of statistical features.

Feature groups are not calculated in a same way. Time and data groups are formed of features that are calculated from the underlying flow fields, which are treated as numeric values. In a flowset, such a feature is calculated by taking the value of its corresponding flow field in each flow and adding it to a collection. Later in the process, a number of statistical features will be extracted from each collection. This method would not work for the creation of a port group. Treating its flow fields

as numeric (i.e. integer-encoded) values would not contribute to their real meaning. For example, because a port is a categorical variable with no ordinal relationship, the order of or distance between port 80 (HTTP) and port 443 (HTTPS) does not explain their difference in terms of a service they represent. Therefore, the port feature group is calculated in a one-hot encoding [108] fashion by creating a collection of frequencies of flows' respective field values.

For each collection, we calculate a number of statistical features depending on the flow field they were derived from. For collections based on number and size of packets, inter-arrival time and duration we extract the following statistical features: minimum, maximum, mean, mean absolute deviation (MAD), kurtosis, skewness, standard deviation (SD), variance and quantiles represented as 10th to 90th percentiles in steps of ten. The collection based on protocol contains all the mentioned statistical features and the number of unique frequency values. Collections based on source and destination ports expand this array of statistical features by: frequency of each of the ten most used ports in the training dataset, aggregated frequency of ports smaller than 49152 excluding the most used ones (i.e. well known and registered ports [125]), and aggregated frequency of ephemeral ports i.e. ports greater than 49152 and smaller than 65535. Additionally, all three feature groups are extracted on a subset of flows that belong to a flowset in terms of their direction with regards to the local IP address on which the flowset was created. In total there are three directions: incoming, outgoing and both of them together.

5.3.2.2 Feature Selection

To avoid the phenomenon of the curse of dimensionality, we choose the most informative features through a selection process. We evaluate their importance through the relative mutual information (RMI) metric. This feature selection method allows us to reduce both the computational cost and the memory required to store flowset feature vectors by eliminating the majority of features that do not

contribute to the classification. RMI is given by the following formula:

$$RMI(X,Y) = \frac{MI(X,Y)}{ME(Y)}$$
$$MI(X,Y) = ME(Y) - CE(Y|X)$$

where MI is mutual information, ME is marginal entropy, CE is conditional entropy, X is a 2-dimensional array with flowsets as rows and features as columns; and Y is an array of flowsets' labels. RMI outputs a score for each feature that is in the range from 0.0 to 1.0. We select the top features ordered by their RMI scores, whose number is determined by a trade-off between their informativeness and space occupied by all chosen features. We define a fingerprint of a flowset X as the vector that comprises only the selected features extracted from X. The use of RMI to select most informative features is inspired by our method from the previous chapter, where it was followed by other procedures. It is reiterated here for the sake of completeness.

5.3.3 Malware Detection

The target of MalAlert is to detect malware and distinguish between their types. While malware traffic datasets offer a clear ground truth, this is not the case for real-world traffic. To cope with this shortcoming, we design our method to be trained on malware datasets so that we can find out extra parameters that we could subsequently apply to malware detection on real-world datasets. In order to assess the optimal parameters in terms of feature groups and number of features to consider, we use cross-validation. For each iteration of cross-validation, we train a random forest classifier, which operates under the closed-world assumption. In order to make the open-world assumption valid for our classification task, we have to provide the classifier with not only the specific target classes that we want to classify, but also with samples from an additional class as a counterexample. In the case of samples of different classes that present very similar network behaviour, the classifier still outputs a classification, but with a low confidence. Malware detection problem specifically aims to reduce false positives, thus we can set a threshold on

confidence to obtain the expected precision (i.e. classification *confidence* threshold). This approach allows us to focus only on samples for which the confidence is above such threshold and ignore the others. As an example, with a confidence threshold set at 0.7, we will consider only the samples whose classifications have the consensus of at least 70% of the random forest estimators. It is worth noting that our methodology so far considers only malware traffic, hence it cannot be applied to real-world network traffic as it is. For this reason, we have to identify optimal set of parameters to retrain our classifier considering the whole malware dataset as a training set with an additional class representing benign network traffic.

5.4 Evaluation

We evalute our proposal using network traffic generated by malware from the MalRec and CTU-13 datasets; and real-world network traffic from the University of Oxford. We perform two experiments. Firstly, we train our approach on malware datasets and evaluate its performance in detecting malware types on a thorough set of parameters. Secondly, we train our model with best performing parameters to detect suspicious traffic in real-world settings using the University of Oxford dataset. We implemented our analyses using the C# and Python programming languages as well as scikit-learn [112] library for the latter. We ran our experiments on a server with 32 CPU cores and 512GB of RAM.

5.4.1 Dataset Specification

The network traffic data involved in the first experiment comes from two sources: MalRec [126] (a dataset of network traffic collected from more than 65,000 malware samples over a two-year period) and CTU-13 [123] (a labeled dataset of botnet, normal and background traffic). MalRec includes VirusTotal reports of binaries that generated network traffic in the dataset. This allows us to divide the samples into malware families that they are representatives of, by using the AVClass [127]

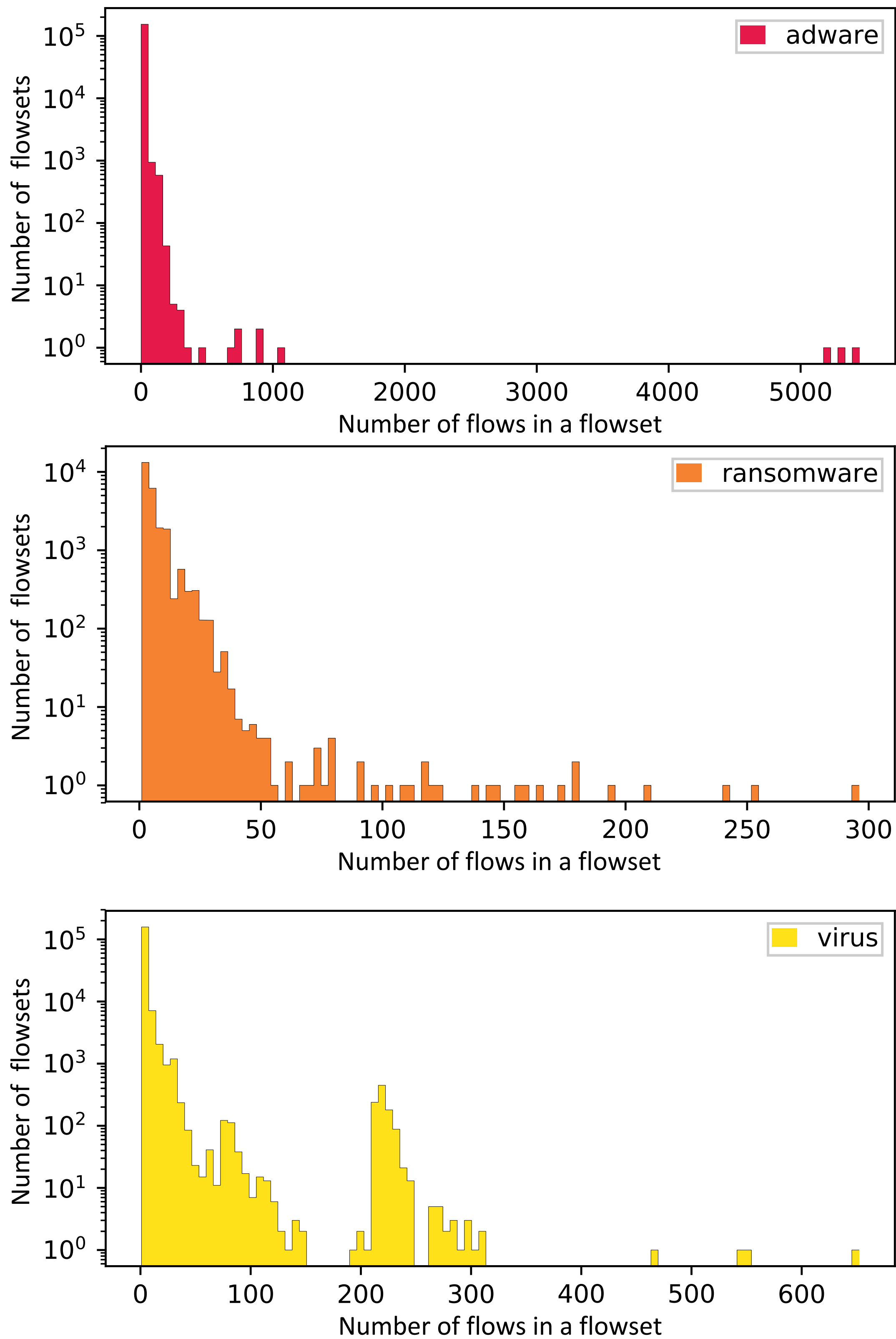


Figure 5.1: Number of flows per flowset in the malware dataset.

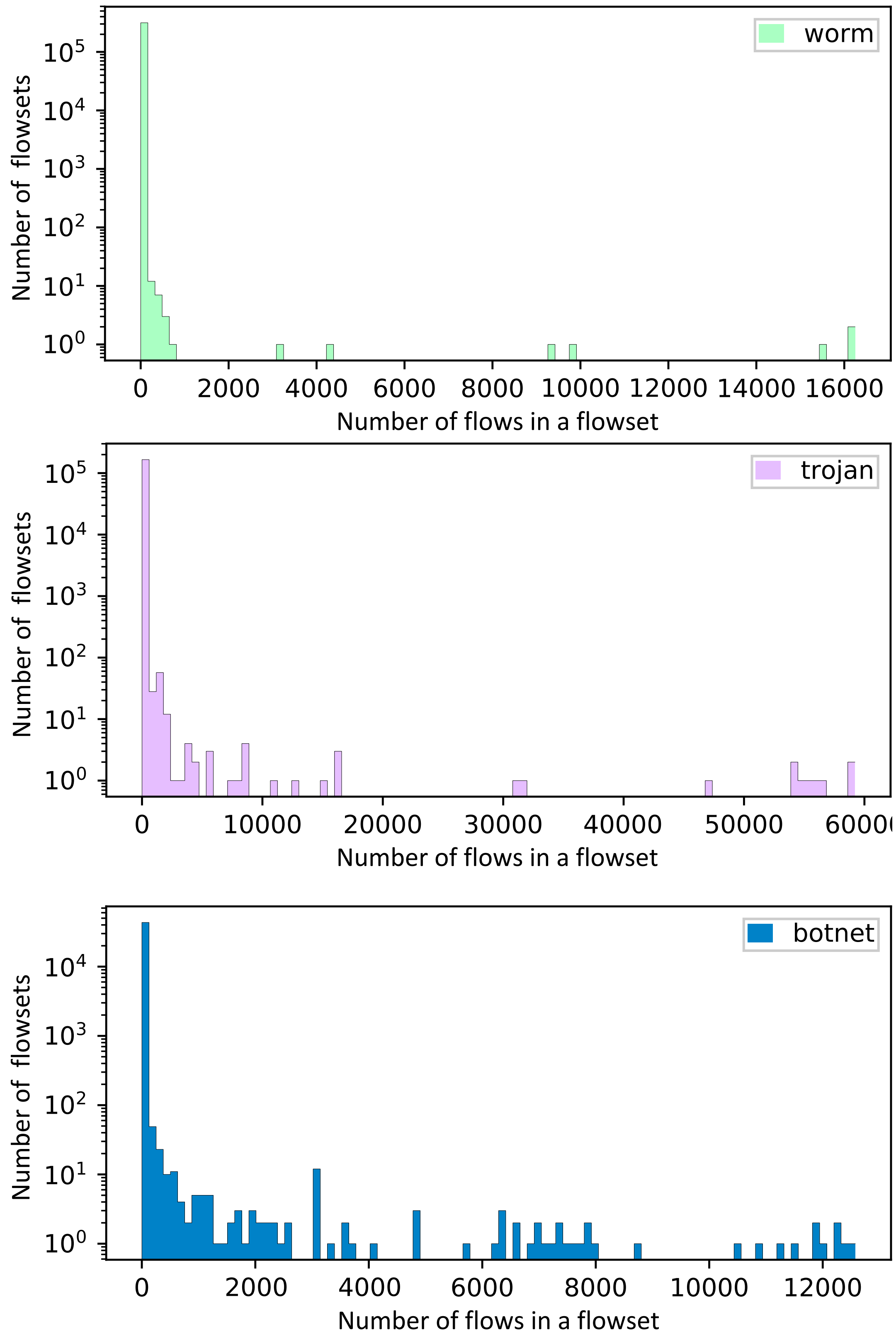


Figure 5.1: Number of flows per flowset in the malware dataset (continued).

Category	Malware Samples	Flows	Flowsets
Adware	11,385	876,556	155,415
Ransomware	2,746	127,605	25,023
Virus	8,126	733,115	171,462
Worm	14,632	1,096,511	314,584
Trojan	4,236	1,378,822	165,767
Botnet	13	547,444	43,617
Other	24,197	3,832,755	545,629
Total	65,335	8,592,808	1,421,497

Table 5.1: Malware datasets details.

malware labeling tool. It outputs a mapping of each malware sample to a family name. We aggregate the number of samples for each family to choose the top 25 with regards to the number of flows. The remaining families are combined into a category referred to as *other*, which consists of 24,197 malware samples. The main reason to include it as a separate category, is to comply with the requirements of the open world assumption (see Section 5.3.3) and to avoid forcing classification. For each considered top family, we manually determine what malware type it represents to elicit five main ones: *ransomware*, *adware*, *worm*, *virus* and *trojan*. The list is expanded by the *botnet* type, which comes from traffic labeled as botnet in the CTU-13 dataset. Based on empirical analysis, we observed that a timeout of 60 minutes is a reasonable trade-off that maximizes the number of flows in a flowset (as discussed in Section 5.3.1) and minimizes the number of flowsets containing a single flow.

Figure 5.1 shows the distribution of number of flows in a flowset per malware type. Considering the timeout value of 60 minutes, only a small percentage of flowsets contain less than five flows. It is worth noting that for worm and botnet types there are flowsets that contain a large number of flows (more than 10,000). This is mostly due to their operational nature, which heavily depends on network communication. The statistics of all considered malware types are shown in Table 5.1 with regards to their number of samples, flows and flowsets. The number of botnet samples is smaller than those of other malware types, because they come from the CTU-13 dataset which contains long captures of network traffic of selected botnet families. In the end, they result in a comparable number of flows. However,

Rank	Direction	Field	Feature	RMI Value
1	**all**	**bytes**	**quantile 0.6**	**0.48551360**
2	**all**	**bytes**	**quantile 0.7**	**0.48336127**
3	**all**	**bytes**	**quantile 0.9**	**0.48315632**
4	**all**	**bytes**	**mean**	**0.47853297**
5	**all**	**bytes**	**quantile 0.8**	**0.47439112**
6	all	bytes	quantile 0.1	0.47121966
7	all	bytes	quantile 0.4	0.46989292
8	all	bytes	quantile 0.2	0.46782586
9	all	bytes	quantile 0.5	0.46584895
10	out	bytes	mean	0.45470081
11	all	bytes	quantile 0.3	0.45032017
12	out	bytes	quantile 0.1	0.44662071
13	out	bytes	quantile 0.4	0.44419630
14	out	bytes	quantile 0.2	0.44400321
15	out	bytes	quantile 0.9	0.44318897
16	out	bytes	quantile 0.6	0.44194841
17	out	bytes	quantile 0.7	0.43949804
18	out	bytes	quantile 0.5	0.43718424
19	out	bytes	quantile 0.8	0.42707932
20	out	bytes	min	0.42454684
21	out	bytes	quantile 0.3	0.42039694
22	all	bytes	max	0.40275439
23	out	bytes	max	0.38747635
24	all	bytes	min	0.38417834
25	all	bytes	SD	0.30444006
26	all	bytes	MAD	0.29327809
27	all	packets	quantile 0.9	0.20194104
28	all	packets	quantile 0.7	0.20004838
29	all	packets	quantile 0.6	0.19980995
30	all	packets	quantile 0.3	0.19915117
31	all	packets	mean	0.19832032
32	all	packets	quantile 0.8	0.19731838
33	all	packets	quantile 0.4	0.19656650
34	all	packets	quantile 0.1	0.19337399
35	all	packets	quantile 0.2	0.19299861
36	all	destPort	otherPorts	0.18393328
37	all	packets	quantile 0.5	0.18384195
38	out	duration	quantile 0.9	0.16498142
39	out	duration	quantile 0.7	0.16381702
40	out	duration	mean	0.16198295

Table 5.2: Features with highest RMI values (selected features in bold).

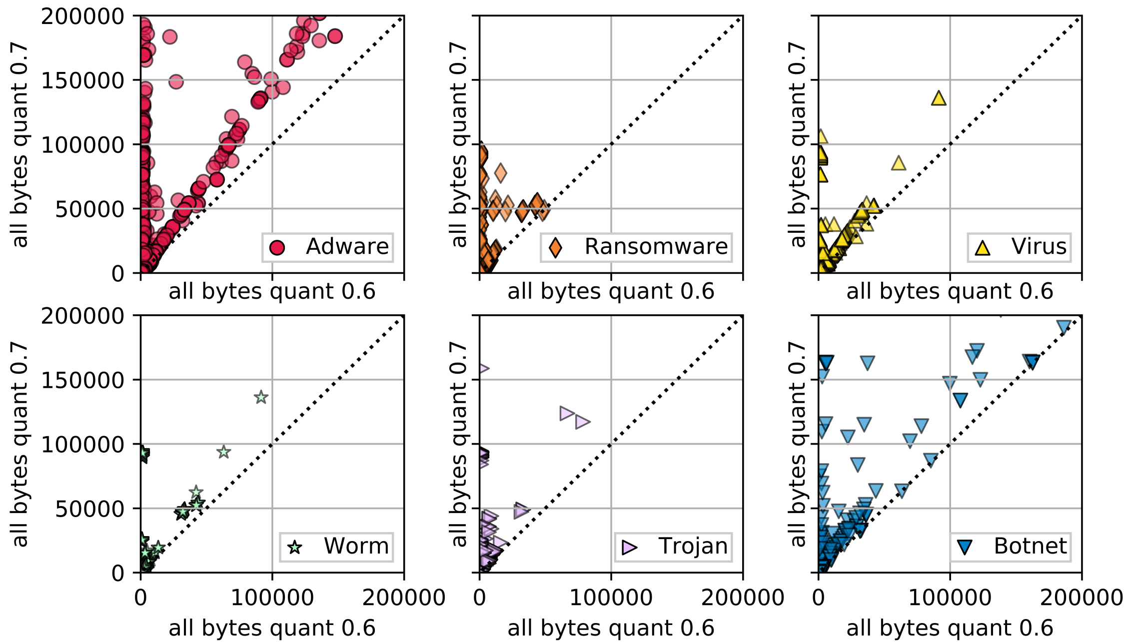


Figure 5.2: Distribution of the top two features projected on a two dimensional space.

their final number of flowsets is smaller due to larger flowsets (flowsets with more flows) as compared to other malware types.

In the second experiment, apart from the malware dataset we take into account the University of Oxford dataset, which comprises of network flow data spanning a one-month period and contains 23,628,240,386 flows that translate to 2,770,033,992 flowsets.

5.4.2 Malware Type Discrimination

In this experiment, we use stratified 5-fold cross-validation on malware datasets varying two parameters: the feature group and the number of top features to consider. For each iteration, we train a random forest classifier with 50 estimators using balanced weights to cope with disparity between the number of samples across different classes.

The first part of this analysis aims to identify which are the most reliable feature groups to discriminate among malware types. As we mentioned in Section 5.3.2, we consider three feature groups according to their domain: time, ports and data

Type	Precision	Recall	F1 score
Other	0.89 (±0.05)	0.45 (±0.12)	0.59 (±0.12)
Adware	0.32 (±0.01)	0.63 (±0.01)	0.43 (±0.01)
Ransomware	0.08 (±0.00)	0.77 (±0.01)	0.15 (±0.00)
Virus	0.76 (±0.01)	0.59 (±0.02)	0.66 (±0.01)
Worm	0.78 (±0.11)	0.64 (±0.10)	0.70 (±0.10)
Trojan	0.58 (±0.23)	0.57 (±0.26)	0.58 (±0.25)
Botnet	0.92 (±0.01)	0.96 (±0.00)	0.94 (±0.00)

Table 5.3: Malware types classification results on 5 fold cross-validation.

features. Moreover, we consider the case of all the features together as an additional group. For each feature group, we rank the features by their informativeness score given by the RMI. We study the performance of classification varying the number of features to assess the optimal number of features that is a good trade-off between their number and classification performance. The results show that the data feature group performs best, which leads us to select the top five features from this group (see Table 5.2). It can be observed that, despite the fact that data features group also contains features based on number of packets, none of them appear among the selected top five. Moreover, none of the features from the other groups occur in the top 35 features as given by the RMI. This makes our method immune to port and protocol spoofing, and to changes in network quality. As a further analysis, in Figure 5.2 we examine the top two features with regards to considered malware types. We can observe that the second feature alone (all bytes quantile 0.7) is enough to discriminate most of adware fingerprints, and a combination of both allows to distinguish most other types.

Once the best features are selected, we further study the individual classification performance for each malware type. As shown in Table 5.3, we achieve an F1 score as high as 94% for botnets, and for other types much higher than the random guess, apart from ransomware. We have to notice that while the recall of adware and ransomware samples is good, the precision is much lower due to the high false positive rate. As reported by the confusion matrix in Figure 5.3, many samples from other types have been wrongly classified as adware or ransomware. This is

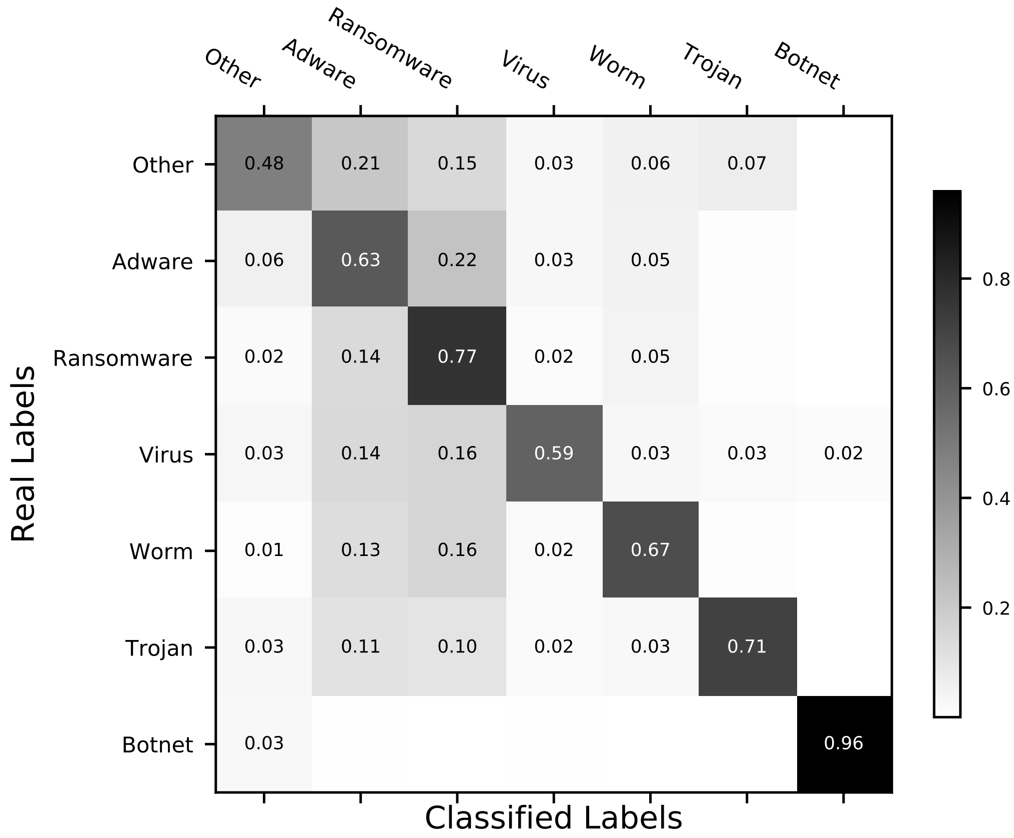
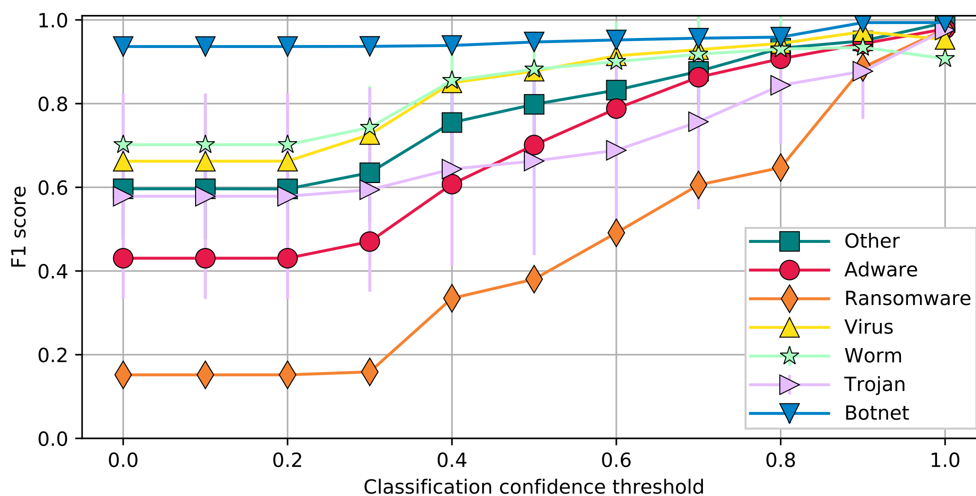


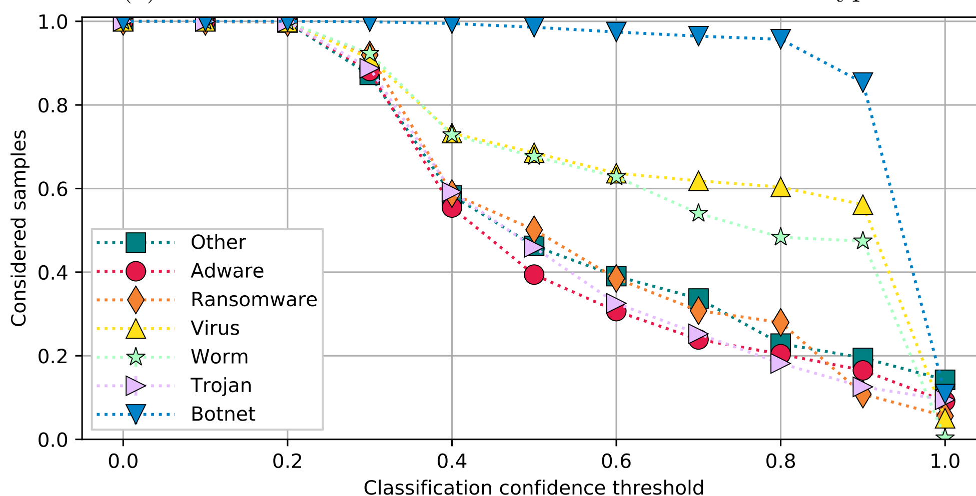
Figure 5.3: Confusion matrix of malware categories classification.

due to the fact that some types of malware can share the same network behaviour while carrying out the malicious activity. These samples cannot be considered as distinctive for a malware type.

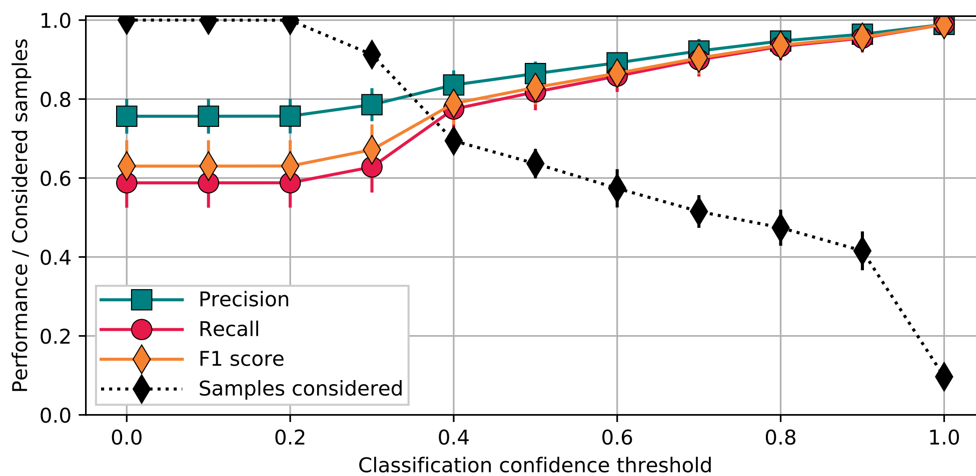
In order to cope with overlapping behaviours between different malware types, we explore the notion of classification confidence threshold to filter out samples that cannot be discriminated. Figure 5.4 presents the performance of our classifier varying the classification confidence threshold. This increase of classification performance comes at a price of the reduced number of samples that are considered as distinctive. This means that some types of malware require more samples to be successfully identified. Figures 5.4a and 5.4b further analyse the impact of confidence threshold on different types of malware in terms of the F1 score and percentage of considered samples, respectively. The overall classification performance and percentage of considered samples is shown in Figure 5.4c. The confidence threshold can be adjusted for each type of malware to achieve the desired F1 score. As an example,



(a) Performance in terms of F1 score for each malware type.



(b) Proportion of considered samples for each malware type.



(c) Overall performance and proportion of considered samples.

Figure 5.4: Performance and proportion of considered samples varying the classification confidence threshold.

Confidence threshold = 0.5

Type	Precision	Recall	F1 score	Samples
Other	0.93 (±0.03)	0.71 (±0.11)	0.80 (±0.08)	0.46
Adware	0.64 (±0.01)	0.77 (±0.02)	0.70 (±0.01)	0.39
Ransomware	0.24 (±0.01)	0.90 (±0.01)	0.38 (±0.01)	0.50
Virus	0.94 (±0.01)	0.82 (±0.02)	0.88 (±0.01)	0.69
Worm	0.86 (±0.14)	0.92 (±0.03)	0.88 (±0.09)	0.68
Trojan	0.60 (±0.23)	0.75 (±0.20)	0.66 (±0.22)	0.46
Botnet	0.92 (±0.01)	0.97 (±0.00)	0.95 (±0.00)	0.99

Confidence threshold = 0.7

Type	Precision	Recall	F1 score	Samples
Other	0.95 (±0.02)	0.82 (±0.12)	0.88 (±0.08)	0.34
Adware	0.85 (±0.00)	0.88 (±0.01)	0.86 (±0.01)	0.24
Ransomware	0.44 (±0.00)	0.95 (±0.01)	0.61 (±0.00)	0.31
Virus	0.98 (±0.01)	0.89 (±0.01)	0.93 (±0.01)	0.62
Worm	0.88 (±0.15)	0.98 (±0.01)	0.92 (±0.10)	0.54
Trojan	0.70 (±0.25)	0.85 (±0.13)	0.76 (±0.21)	0.25
Botnet	0.93 (±0.01)	0.99 (±0.00)	0.96 (±0.00)	0.96

Confidence threshold = 0.9

Type	Precision	Recall	F1 score	Samples
Other	0.98 (±0.01)	0.92 (±0.06)	0.95 (±0.03)	0.19
Adware	0.95 (±0.00)	0.94 (±0.01)	0.94 (±0.00)	0.17
Ransomware	0.82 (±0.01)	0.97 (±0.01)	0.89 (±0.01)	0.11
Virus	0.99 (±0.01)	0.96 (±0.01)	0.97 (±0.01)	0.56
Worm	0.90 (±0.16)	0.99 (±0.00)	0.93 (±0.10)	0.47
Trojan	0.82 (±0.16)	0.95 (±0.04)	0.88 (±0.11)	0.13
Botnet	1.00 (±0.00)	0.99 (±0.00)	0.99 (±0.00)	0.85

Table 5.4: Malware types classification performance on 5 fold cross-validation and proportion of considered samples with three classification confidence thresholds.

we have to consider thresholds of 0.4 for adware and 0.7 for ransomware to detect such malware types with the F1 score higher than 0.6. Table 5.4 shows the classification performance in terms of the F1 score for fixed confidence thresholds of 0.5, 0.7 and 0.9. The values in the *Samples* column mean the percentage of all fingerprints for a given type, for which a given classification confidence is achieved.

5.4.3 Suspicious Traffic Identification

Using the insights obtained from the analysis of malware dataset, we apply the classifier trained on the malware dataset to the University of Oxford dataset. Giving as input a sample, we are able to identify traffic that is similar to malware with a certain confidence. Figure 5.5 shows the number of samples per hour detected in the University of Oxford network that resemble behaviours of various malware types considering different values for the classification confidence threshold. There are two y-axis scales in each subfigure: the left y-axis shows the absolute number of samples, while the right one shows the percentage of samples of a given malware type. Similarly to the terminology in previous sections, a sample is equivalent to a fingerprint, whereas a fingerprint is a vector of selected features from an underlying flowset. We identify the F1 score of 0.9 to be a reliable value that reduces the amount of false positives. In order to achieve it, the corresponding confidence thresholds for adware and ransomware are 0.8 and 0.9, respectively. This results in the average of 100 suspicious samples per hour for adware and less than 10 per hour for ransomware. When we consider viruses and worms, which share the same confidence threshold of 0.7, we observe single occurrences during the course of one month. For trojans and botnets, the respective confidence thresholds are 0.95 and 0, which translates to less than 10 and tens of thousands occurrences of malicious samples.

5.5 Discussion

Malware detection and classification into types using only network flow data is known to be a hard problem. This is attributed to the coarse-grained nature of such data, challenges with obtaining ground truth in real-world datasets, and the scarcity of available datasets due to them containing private and often confidential information.

Similarly in our work, we face those challenges and attempt to overcome them using a machine learning-based method. Due to not being able to get and vouch for

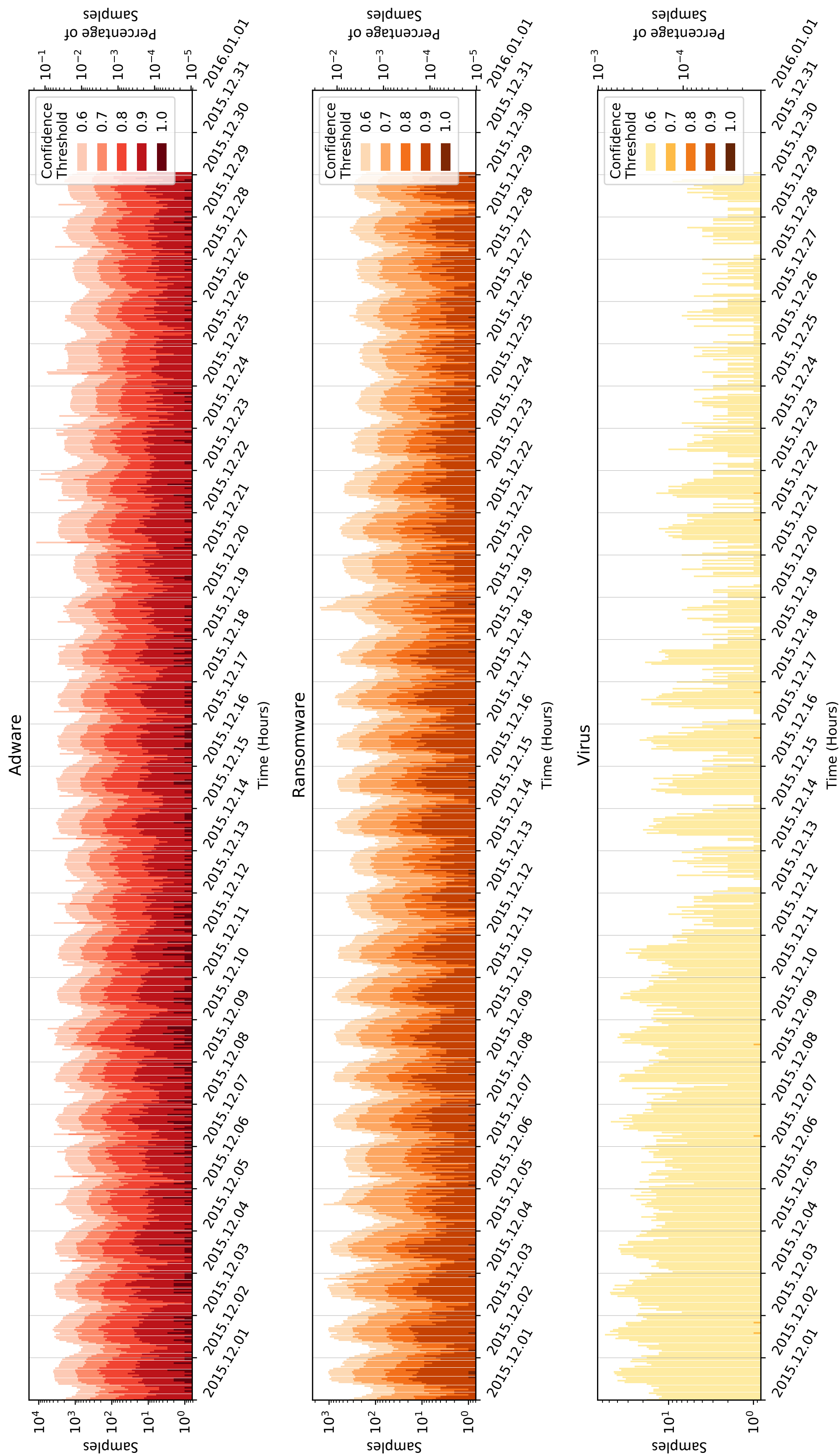


Figure 5.5: Samples (i.e. fingerprints) from the University of Oxford dataset, identified as generated by various malware types.

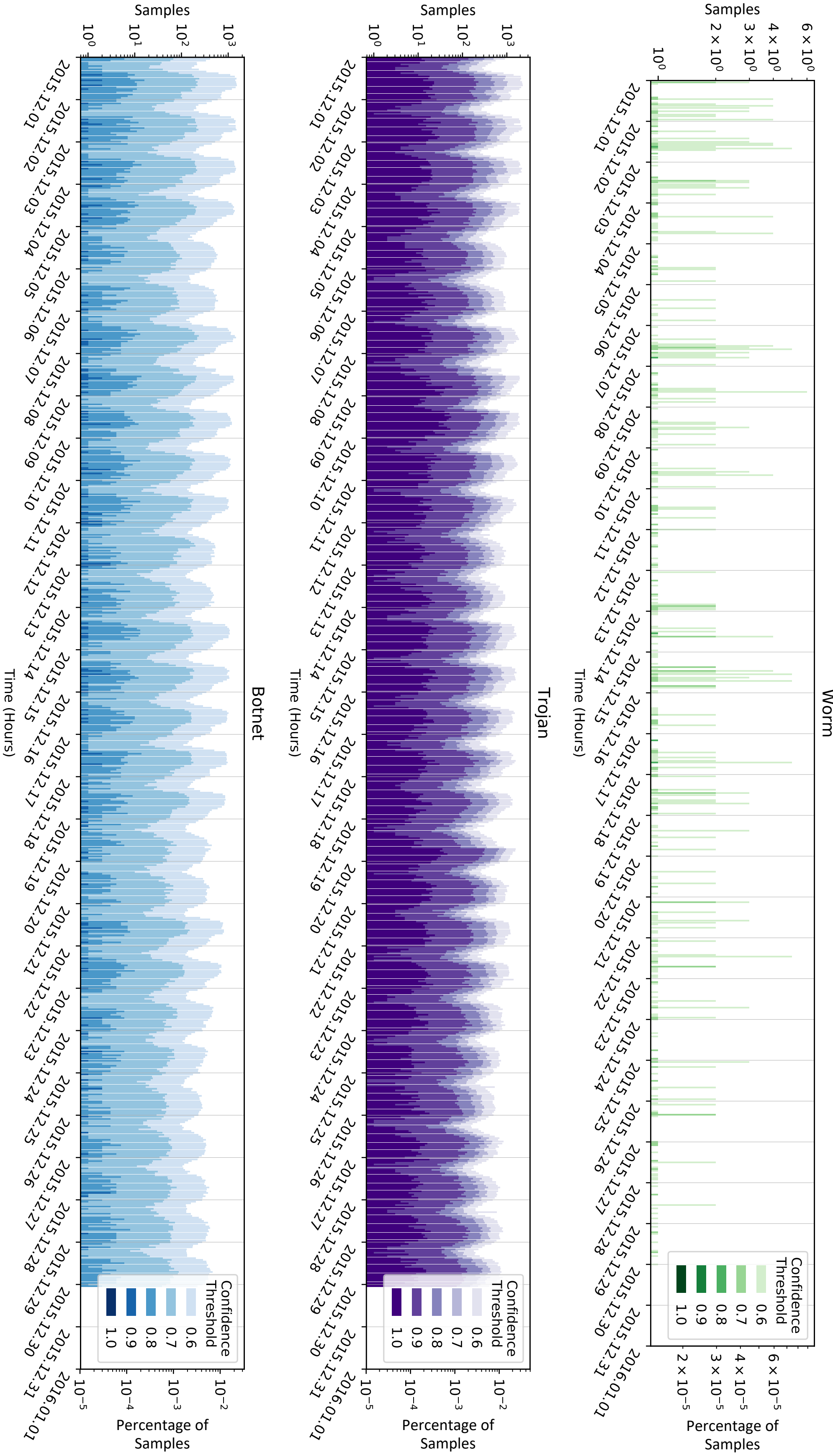


Figure 5.5: Samples (i.e. fingerprints) from the University of Oxford dataset, identified as generated by various malware types (continued).

the ground truth in the University of Oxford dataset, we resort to training the model on known malware datasets that also contain diverse benign traffic. The limitation of our method is the assumption that the benign traffic is representative enough of a wide variety of normal traffic on real-world networks. However, despite the limited knowledge of ground truth in University of Oxford dataset, the presented results for the confidence threshold of 1.0 (and to a lesser extent confidence threshold of 0.9) show flowsets whose features match exactly the ones produced by malware. Therefore, while ground truth would be important to fully evaluate results with lower confidence thresholds, the highest-threshold results could be used directly by security analysts as an indication of malicious activity.

In order to construct the malware dataset and specifically to be able to evaluate malware type detection of MalAlert, we manually divide them into five malware types (however we do not do so for the sixth botnet type as we know all samples from the CTU-13 dataset represent botnet malware). This means that each family given by the AVClass that has a large number of flows is compared with antivirus vendor reports. From a set of reports the most common malware type is extracted in order to create a mapping between a malware family and a malware type it belongs to. It may be viewed as a similar process to AVClass', however there is no tool that offers this functionality for malware types. Another limiting factor and justification for the lack of such a tool might be the ambiguity with regards to malware taxonomy on the type level, which is further discussed in Section 6.7.1.

In the design of MalAlert, we aimed to avoid injecting traffic across different datasets due to the concern of maintaining the order of events as well as their integrity. That is, traffic captured on one network (often also isolated from the Internet or other devices on a local network) could exhibit behavioural traits that would not be generated if the same malware was run on another network. However, the injection avoidance comes at a price of not being able to accurately judge the classification performance with applying the model on the University of Oxford dataset. We overcome this with the help of a confidence threshold, which is an additional mechanism to improve the certainty of obtained results, at a cost of fewer

data points that can be considered. To avoid the shortcomings stemming from lack of ground truth in real-world datasets, we examine an alternative approach that favors traffic injection (but restricted by certain conditions) in Chapter 6.

5.6 Conclusion

Evaluation and comparison of malware detection methods on real-world networks are challenging tasks because of privacy concerns, heterogeneity of networks and lack of ground truth [128]. In this chapter, we proposed a method to tackle such challenges. We illustrated how our proposal is able to detect suspicious traffic with certainty high enough to be blocked automatically.

In the first experiment, we showed that flows grouped into flowsets preserve their informativeness in such a way that we are able to distinguish between malware types - most of them with high (e.g. trojan) and some of them with not sufficiently high confidence (e.g. ransomware). However, for certain types of malware (that do not generate significant amount of network traffic) confidence thresholds require to be tuned properly. The method then aims to point the administrators in the right direction to find the source of the malicious behaviour, which they can examine more closely using other sources of information (e.g. application and security logs, packet-level data that can be captured for high-priority network resources parallel to network flow capture to form a two-stage system [22]). In the second experiment, we detected suspicious flowsets in the real-world traffic of a large-scale network, which resulted to be in the order of magnitude 10^{-6} per hour which is reasonable given the rarity of malware events. Contrary to the large amount of false positive alerts received by security operation centers, MalAlert identifies a small number of suspicious flowsets with high confidence.

6

Fine-Grained Malware Detection Using Enhanced Network Flows

Contents

6.1 Introduction 101
6.1.1 Contributions 102
6.2 Background 103
6.2.1 Challenges of Network-Based Detection 104
6.3 Related Work 105
6.3.1 Machine Learning Applied to Malware Analysis 105
6.3.2 Network Flows 106
6.3.3 Entropy of Packets 107
6.3.4 Detection of Malware from Benign 107
6.3.5 Family Classification 108
6.3.6 Comparison to Other Proposals 109
6.4 System Architecture 111
6.4.1 System Overview 112
6.4.1.1 Flow Extraction and Encoding 114
6.4.2 Denoising and Classification 115
6.5 Datasets 116
6.5.1 Noise Injection 121
6.6 Evaluation and Results 122
6.6.1 Clean Samples Classification 123
6.6.1.1 Phase 1: Binary Classification 124
6.6.1.2 Phase 2: Type Classification 125
6.6.1.3 Phase 3: Family Classification 126
6.6.2 Noisy Samples Classification 129
6.6.2.1 Phase 1: Binary Classification 130
6.6.2.2 Phase 2: Type Classification 131
6.6.2.3 Phase 3: Family Classification 133

6.6.2.4 Phase 1: Binary Classification of Unseen Malware 134
6.6.3 Tier Comparison 135
6.7 Discussion and Limitations 136
6.7.1 Malware Taxonomy 136
6.7.2 Network Flows 137
6.7.3 Noise Injection 138
6.7.4 Unseen Type Classification 138
6.7.5 Sandboxes 138
6.7.6 Evasion 139
6.8 Future Work 140
6.9 Summary 141

Economic incentives encourage malware authors to constantly develop new, increasingly complex malware to steal sensitive data or blackmail individuals and companies into paying large ransoms. Just for 2017, the worldwide economic impact of cyberattacks is estimated to be between 445 and 600 billion USD, or 0.8% of global GDP [129]. Traditionally, one of the approaches used to defend against malware is network traffic analysis, which relies on network data to detect the presence of potentially malicious software. However, to keep up with increasing network speeds and amount of traffic, network analysis is generally limited to work on aggregated network data, which is traditionally challenging and yields mixed results.

In this chapter, we present MalPhase, a system that was designed to cope with the limitations of aggregated flows. MalPhase features a multi-phase pipeline for malware detection, type and family classification. The use of an extended set of network flow features and a simultaneous multi-tier architecture facilitates a performance improvement for deep learning models, making them able to detect malicious flows ($>$ 98% F1) and categorize them to a respective malware type ($>$ 93% F1) and family ($>$ 91% F1). Furthermore, the use of robust features and denoising autoencoders allows MalPhase to perform well on samples with varying amounts of benign traffic mixed in. Finally, MalPhase detects unseen malware samples with performance comparable to that of known samples, even with mixed in benign flows to reflect realistic network environments.

6.1 Introduction

In the last decades, researchers proposed many different network traffic analysis approaches which can, with varying degrees of success, identify ongoing malware communication in the network. Based on the level of granularity of the information they require, these approaches can generally be categorized as *packet-level* network analysis and *flow-level* network analysis. Packet-level network analysis approaches use detailed network traffic data such as individual packets or HTTP connections in order to classify network streams [130]. Packet-level approaches can generally reach very high performance, as the use of detailed network traces enables the detection of finer differences between benign and malware traffic, that are otherwise lost when using aggregated network data (e.g. NetFlows [12, 131]). However, the use of detailed network traces renders these approaches ill-suited to real life applications, where capturing such traces at line speed is challenging and expensive. Flow-level network analysis approaches on the other hand use aggregated traffic flow information, which can be more readily logged for analysis. However, classification based on flows is considerably more difficult, with traditionally mixed results due to the reduced amount of information available. Approaches that rely on network flows generally have reduced scope and are limited to malware detection only [41], malware type classification with few families [12] or family classification only for a specific malware type [131].

Using one of the largest dataset of malware traffic to date (see Section 6.5), in this chapter we revisit the topic of traffic analysis with aggregated flows, building on the method proposed in the previous chapter. Contrary to MalAlert that employed smaller datasets and a manual feature selection method, large amount of malware network data that is available today from sources such as VirusTotal [15], combined with the ability of neural networks to extract robust features from such a large dataset, can allow to overcome the limitations of flow-based analysis. We propose MalPhase, a multi-phase system that is designed to cope with the limitations of flow-based analysis and is able to detect malware traffic, as well as classify it to a specific malware type and malware family, which is an extension of the abilities of

MalAlert that was limited to malware type detection only. MalPhase uses a multi-tier classification system that is responsive to short term bursts of malware traffic, as well as slower, more regular malware network activity. Our evaluation demonstrates that MalPhase achieves extremely high performance in malware detection that is comparable or better than state-of-the-art ($> 98\%$ F1-score), while also broadening the scope of classification to malware type ($> 93\%$ F1-score) and malware family ($> 91\%$ aggregated F1-score). Finally, MalPhase uses a combination of denoising autoencoders and Deep Neural Network (DNN) classifiers that make it resilient to high levels of noise injected in the malware traffic, as well as being able to detect unseen malware samples with performance comparable to that of known samples.

6.1.1 Contributions

In this chapter we make the following core contributions:

(i) We design and implement MalPhase, a multi-phase system for malware detection and classification based on network flows. MalPhase is able to detect a large set of malware families from network flows, as well as classify it to a malware type and even specific malware family.

(ii) The MalPhase system can work both as an offline analysis tool, as well as online on live data. This is enabled by its multi-tier design which allows it to work on different timescales in parallel.

(iii) We evaluate MalPhase on a large, standardized malware dataset, obtained from a single source. To the best of our knowledge, this is one of the largest datasets of labeled malware traffic to date (~ 1 billion flows). We assess MalPhase performance in a real world setting composed of both clean malware traffic as well as noisy traffic (i.e. malware traffic with mixed in benign flows), assessing the robustness of our system to real life conditions. Moreover, we also evaluate the ability of MalPhase to generalise and detect unseen malware samples, even with the presence of noise.

6.2 Background

The sub-field of malware network traffic analysis typically performs analysis on datasets of either malicious traffic mixed with benign, with the purpose of discerning which network indicators and properties represent malicious activity; alternatively, other research observes traffic from known malware samples. In enterprises, there is significant investment in the establishing and maintaining security operations centres that leverage state-of-the-art research in this field to allow security analysts to identify previous cyber-attacks or attacks in progress [132] from network traffic and other log sources.

Host-based Intrusion Detection Systems (HIDS) are also used for the detection of malware. These methods can take the form of signature-based analysis, wherein the output of a hash function is compared against a key-value store. Static host-based analysis, is also used to identify and classify malware [133]. Static analysis and reverse engineering of binaries is a time-consuming and complex task that does not scale, in comparison to forensic analysis of malware's network traces. Polymorphism and packers have made signature and static-based analysis less effective. In response to this, malware analysis has adopted dynamic analysis, that looks at the behaviour of malware. Dynamic behavioural analysis of malware by detonation in sandboxes began with the Anubis platform. This led to a number of different papers that looked at the behaviour of malware [134–136]. Dynamic analysis via sandboxes is now prolific in the security community. The combination of host-based and network-based intrusion detection methods is what is referred to as a hybrid intrusion detection system.

This field has evolved from detection of intrusions to detection and classification of particular malware samples. Large-scale measurement studies have also been performed to evaluate the network behaviour of malware [137]. Other research has looked at network forensic artefacts of malware delivery mechanisms [138] and bulletproof hosting [139]. Our research is concerned with the analysis of network traffic of malware, rather than aggregated artefacts such as DNS domains.

Two data types are typically used for the analysis of malware network traffic. Firstly, there are packet captures (pcaps), the most basic data type used for computer communications over a network. Packet captures are not often used for analysis at scale, due to the amount of storage required, and the ubiquity of encryption which negates the usefulness of storing packet data. The second data type, network flows (from here on, *flows*), are aggregations of packets that provide a summary of a given communication without the storage requirements of packet captures. The key features are the protocol in use, source and destination IP addresses and ports, as well as number and sizes of contained packets. In our experiments, we use bidirectional flows augmented with round-trip time of packets and packet payload entropy in both directions. Critically, flows are a common method of collecting and analysing data at scale in large organisations while preserving the privacy of network data, hence they are more accessible than packet captures.

6.2.1 Challenges of Network-Based Detection

There are a variety of problems that are inherent to using network artefacts to identify malware. Aviv and Haeberlen [128] identify a number of challenges with detection of malware from network traffic. *Realism* and *Representativeness* are challenges that raise the issue of the scale and environment of testing in academic research not representing realistic implementation scenarios. Our experiments aim to account for this via the injection of benign traffic, to more accurately represent real-world networks, for which the majority of traffic is benign. *Generality* is concerned with the fact that many botnet detection methods simply assess one botnet family. We mitigate this by looking at over 40 different families spread across 5 malware types. Another challenge not covered by the paper is that malware authors are also aware of network forensic capabilities and have adapted to this. Many malware families started using Fast-Flux domains [140] wherein the domains being used rapidly switch. Fast-flux networks use domain generation algorithms to coordinate this and research focused on detecting use of these Fast-flux networks by malware [141].

Year	Work	Malware Hashes
2007	BotFinder [142]	188
2010	Hsu et al. [143]	12,629
2010	Perdisci et al. [120]	25,720
2012	DISCLOSURE [41]	37,687
2013	BotSuer [40]	2,143
2013	FIRMA [121]	15,850
2014	Nazca [144]	43,380
2014	Chatter [122]	2,699
2016	Bartos et al. [32]	7,000
2017	Lever et al. [137]	26,800,000
2018	Deng et al. [145]	999
2021	MalPhase	13,920,730

Table 6.1: Malware sample size in related work.

6.3 Related Work

We enumerate several different proposals that approach the subject of network-based detection or classification of malware, resultant from execution of malware samples. We list the number of samples that each paper leverages in Table 6.1. These proposals do not all use flows and are covered in more detail later in related work. Apart from Lever et al. [137], the dataset used in our study is the largest to date, with several orders of magnitude more malware samples than comparable related work. It is worth noting, however, that [137] does not address the issue of automated malware detection and classification.

6.3.1 Machine Learning Applied to Malware Analysis

Machine learning as applied to the field of network intrusion detection and malware classification is documented comprehensively. These algorithms are applied to packing [146] and other areas of malware research. The most common use case for machine learning in malware research is that applied to the analysis of malware network traffic and intrusion detection. This is so prolific that Sommer and Paxson

described challenges and recommended best practices for using machine learning for the use case of network intrusion detection [147].

There are different approaches taken to using machine learning for network traffic analysis. Some exploratory research investigates the use of deep learning to facilitate network traffic analysis to identify malware, Marín et al. [148] limited this analysis to classification of 3 separate families. Shibahara et al. [133] also make use of deep learning in their paper, but the analysis is based on detailed network traffic data, such as DNS and HTTP protocol data. In [149] Zhu et al. describe a natural language processing-based method to learn malware behaviour from academic literature and automatically extract meaningful features for malware detection. While this work belongs to the same general area of malware detection, the proposed system is not a network traffic analysis-based approach and uses a much wider range of features.

One of the most notable recent work related to malware classification based on network traffic is a paper from Bartos et al. [32]. In their work, the authors proposed a new system to detect unseen malware samples based on statistical representation of features learned from "bags of samples" – an abstraction which represent a set of web proxy logs that are strictly related to each other. While the goal of this work is close to our own, its scope, the level of detail of the features used for classification, as well as how this goal is achieved differ from ours. The method proposed by the authors extracts traffic features from web proxy logs, which provide a richer set of information compared to flows, at the cost of increased capture complexity and scarcer availability. Moreover, the system in [32] is limited to detection of malware and does not perform type nor family classification.

6.3.2 Network Flows

Various proposed systems apply machine learning techniques to network flows in order to detect traffic belonging to malicious actors. Disclosure conducts analysis of flows to detect botnet command and control (C&C) servers, regardless of the

protocol it employs [41]. While their work offers a thorough evaluation on two real-world networks, it is specifically tailored to one specific malware type - botnet. Piskozub et al. [12] process network flows to identify malicious traffic in addition to malware type classification. However, they have a limited dataset, use only clean malware traffic and do not perform family classification. Gezer et al. [150] propose a flow-based machine learning approach to detect the banking trojan TrickBot. The approach proposed by the authors heavily differs from ours in term of features, architecture, as well as in the reduced scope of the paper.

6.3.3 Entropy of Packets

Whilst to the best of our knowledge, entropy of packet payloads has not been considered as a feature in flow-based methods preceding our work, BotHunter [36] is a system that conducts packet-level analysis and uses entropy as an indicator of maliciousness. Do et al. [151] have also used entropy to identify synthetic attacks. Entropy of underlying binaries has been reliably used as an indicator of maliciousness [152, 153]. Although recent research has shown that this assumption may be flawed [154], regarding packers. This is similar to network forensics, wherein it can be assumed that entropy is an indicator of encryption and therefore maliciousness.

6.3.4 Detection of Malware from Benign

In [130], Ahmed et al. propose a method to detect malicious software via analysis of network packets. While detection of malicious activity in network traffic is also part of the scope of our work, the approach proposed by the authors is aimed only at detecting malware, rather than also classifying it to type and family. Moreover, [130] depends on detection of executable files in packet payloads and therefore requires deep packet inspection, which makes it less practical than our flow-based approach. Our approach of injecting benign traffic to more accurately simulate real world

network traffic is also adopted by other researchers. Zarras et al. and Lamprakis et al. create simulated HTTP traffic to detect HTTP-based malware and APT malware respectively [155, 156]. However, their methods are tailored to a specific malware family, communication protocol or malware architecture (C&C channels), and are not designed to detect or classify a variety of malware. Moreover these approaches are based on the analysis of detail network data such as web request graphs and HTTP-level analysis.

6.3.5 Family Classification

The field of malware network traffic analysis also has a number of proposals that approach the problem of malware family classification. Perdisci et al. [120] propose an unsupervised clustering method to group similar malware families and automatically generate signatures for detection. One of the key differences between this work and our own is the reliance of the authors on individual packet content in the form of HTTP request/response fields. As previously discussed, such detailed analysis of network traffic is challenging in a real world scenario because it is expensive and hard to scale. Moreover, the system proposed by the authors uses fields that can easily be altered by malware authors, such as URLs, which reduces the robustness of the clustering. Sebastián et al. also address the problem of malware family classification with the AVClass tool [127], which automatically provides a family label for a malware sample starting from a set of antivirus labels.

Firma [121] clusters traffic into particular families and then generates IDS signatures for those families. An aspect that separates our system from other work is the breadth of families we are able to identify. For instance Firma only classifies 3 distinct families, and uses packet captures rather than flows. A subset of papers that identify families of malware, identify botnets from benign traffic. BotFinder [142] is an exploratory research system that processes NetFlows or packet captures to extract the presence of botnet traffic of 6 specific families. Gu et al. BotHunter [36] is another well-known system that conducts packet-level analysis to enable bot

identification, based on a stage of botnet behaviour. Similarly, BOTection [131] is a machine learning-based system that uses network flows to detect and classify botnet families, building on earlier work on behavioural analysis on network flows by the same authors [103]. BotMiner [38] is an approach to detect botnet based on unsupervised clustering and cross-cluster correlation that is able to detect the presence of botnet malware in network traffic. All these approaches heavily differ from ours for multiple reasons. First, these proposals are limited to the detection — and in some cases classification — of only botnet malware. The scope of our work is broader, since the detection and classification capabilities of our system are not restricted to a single malware type. Furthermore, several of these proposals only perform botnet detection, and not classification. Those that perform classification as well, do it on a limited number of unique families (up to 17 for BOTection, as compared to 38 in our case). Finally, our system differentiates from most previous proposals in the area by being able to reliably detect and classify different families and types even with the injection of noise, in the form of benign flows interleaved with the malicious flows (see Section 6.5.1).

6.3.6 Comparison to Other Proposals

Finally, in Table 6.2 we present a systematic comparison to previous work ordered by publication date. It is worth noting that the majority of approaches are tailored only to detection of botnets and their C&C servers, hence they are not malware type-agnostic. A number of approaches are designed to work only with HTTP protocol, which used to be employed in the majority of malware (mostly botnets). However, nowadays it should not be taken for granted. Older proposals show simplicity by using basic methods, such as clustering, which we do not consider to belong to machine learning-based methods (i.e. those contain more advanced approaches such as random forest or deep learning). None of the systems in previous work offers malware type classification, instead they focus on malware family classification. This could be due to the fact that establishing a coarse-grained taxonomy of malware, such as types, is not well-described in the literature and it

Work	Malware Type-Agnostic	Multiple Application-Layer Protocols	ML-Based	Malware Type Classification	Malware Family Classification	Unseen Malware Detection	Mixed Malicious and Benign Traffic	Content-Agnostic \ Immune to Encryption	Data Type
BotHunter [36]		✓				✓	✓		NIDS logs
BotSniffer [37]						✓	✓	✓	NIDS logs
BotMiner [38]		✓				✓	✓	✓	Flows & NIDS logs
Perdisci et al. [120]	✓				✓	✓			Packets
BotFinder [142]		✓			✓		✓	✓	Flows
DISCLOSURE [41]		✓	✓				✓	✓	Flows
BotSuer [40]		✓						✓	Flows
FIRMA [121]	✓	✓							Packets
Chatter [122]	✓	✓	✓		✓				Packets
Nazca [144]	✓					✓	✓		Packets
Bartos el al. [32]	✓		✓			✓			Web proxy logs
MalClassifier [103]	✓	✓	✓		✓			✓	NIDS logs
BOTection [131]		✓	✓		✓	✓	✓	✓	NIDS logs
MalAlert (Chapter 5)	✓	✓	✓	✓				✓	Flows
MalPhase (Chapter 6)	✓	✓	✓	✓	✓	✓	✓	✓	Flows

Table 6.2: Comparison of related work to our proposed systems: MalAlert and MalPhase.

would require a precise set of malware types as well as ways to map families into those types. The interpretation of "unseen malware detection" criterion is different for those methods that are ML-based – we require methods fulfilling this criterion to have ground truth about evaluated malware that is deliberately withheld in the testing phase. Hence methods claiming to detect unseen malware based on training a model on malware traffic and testing on a real-world data without ground truth

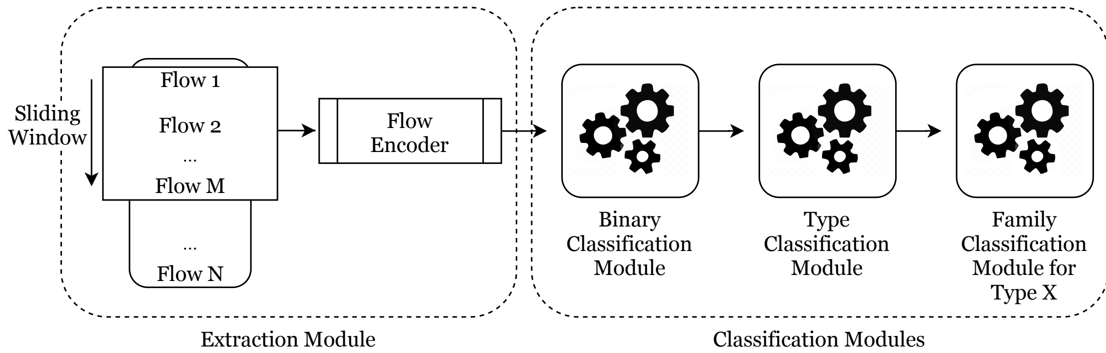


Figure 6.1: MalPhase pipeline. Flows from an individual source are encoded and ran through a pipeline of three classifiers: (i) a binary benign/malicious classifier; (ii) a malware type classifier and (iii) malware family classifier for the given malware type.

are not truly able to detect unseen malware according to our requirements. Previous approaches use different types of network data as input. As explained in Chapter 2, the types are divided into packets and flows, with NIDS logs being closer to flows and web proxy logs being closer to packets in their level on detail.

As seen in the comparison table, our first proposal (MalAlert) fulfills most of the criteria and explores malware type classification. Our second proposal (MalPhase) improves on related work and explores a more complete set of objectives, by offering detection based on desired granularity (from binary malware detection, through malware type classification to malware family detection) while working with data type that is privacy-preserving and holds the least information as compared to others.

6.4 System Architecture

MalPhase is a multi-tier, multi-phase system that combines supervised and unsupervised machine learning techniques to provide fine-grained malware classification from aggregated network flow data. Rather than analysing traffic network-wide, MalPhase works on a per-host basis and supports both real-time traffic analysis as well as analysis of stored network data. As depicted in Figure 6.1, the MalPhase pipeline is comprised of a flow extraction and encoding step, followed by three classification phases:

1. **Binary Classification Phase.** In the first phase, the network flows undergo a *denoising* step aimed at extracting features linked to malicious activity, while filtering out features arising from benign traffic. The denoised flows are then input to a binary DNN classifier which categorizes them as either *benign* or *malicious*.

2. **Malware Type Classification Phase.** Flows categorized as malicious in the binary phase are further input to a multiclass DNN classifier which identifies the *type* of malware that generated the flows (e.g. worm, ransomware). Consistent with other categorisations in the literature [12, 157], in MalPhase we consider five types of malware: adware, ransomware, trojan, virus, and worm.

3. **Malware Family Classification Phase.** In the third and final phase, malicious flows are fed to a type-specific DNN classifier (e.g. ransomware classifier if the flows are classified as belonging to a ransomware in phase 2), which further categorizes them into the specific malware family that generated the traffic.

We designed the architecture of MalPhase classification modules to provide robust classification that is resilient to noisy inputs, due to the use of denoising autoencoders, as well as to different distributions of benign and malicious traffic, thanks to the use of a multi-tier system. In the following sections, we discuss in detail the different components of MalPhase.

6.4.1 System Overview

The MalPhase pipeline is comprised of two main modules: (i) the extraction module and (ii) the classification modules. The extraction module is responsible for extracting flow features from a window of m flows, encoding and concatenating them before forwarding the compacted features to the classification module. The classification modules are responsible for denoising the encoded flows, removing

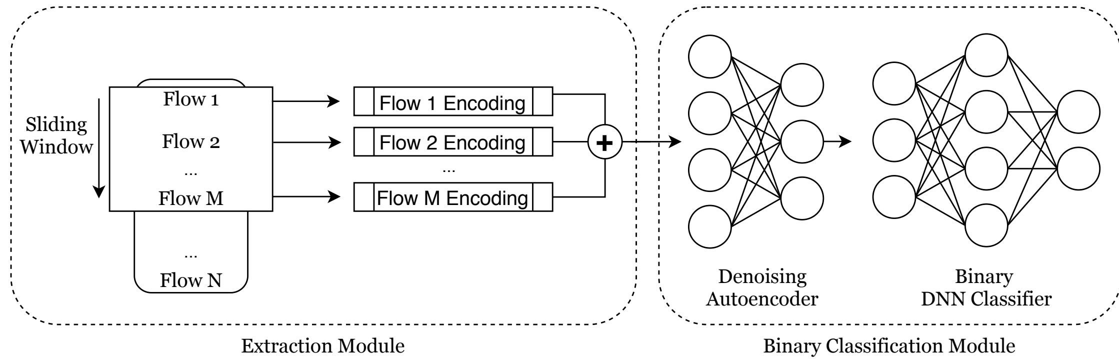


Figure 6.2: MalPhase detailed architecture. A sliding window extracts M flows, which are encoded and compressed in a compact representation. The encoded flows are then ran through a denoising autoencoder to filter-out benign traffic. The remaining malware flow features are then passed on to the binary classifier, and successively to the type and family classifiers if the sample is malicious.

features related to benign traffic, and for classifying the flows with the appropriate classifier based on the current phase (binary, type or family). A detailed illustration of this process is depicted in Figure 6.2. As previously discussed, MalPhase's pipeline uses three classification modules, each handling a different phase of classification: (1) benign/malicious (binary) classification; (2) malware type classification and (3) malware family classification. The MalPhase pipeline is designed to work on a per-host level, rather than at a network-wide level. This allows MalPhase's classification modules to deal with much less noise in the form of benign traffic generated by other machines, which would make it much harder to detect and classify malware traffic.

The MalPhase pipeline is replicated and structured in an N-tier system: each tier runs a full copy of the pipeline on a different group of flows: lower tiers work on short sequences of flows, while higher tiers process increasingly longer sequences of flows, as illustrated in Figure 6.3. The classification modules of each tier are fine-tuned to work on a specific number of flows and require different amounts of malware flows to accurately classify samples, with higher tiers requiring more flows then lower tiers. We design this tier system to allow MalPhase to monitor both short-term and long-term behavioural changes in flows: lower tiers capture short bursts of malicious flows, while higher tiers capture slower, more regularly-occurring malicious flows, similarly to [158]. Moreover, the tiered architecture of MalPhase supports both real-time traffic analysis and analysis of stored flow data. Since lower

tier classification modules operate on shorter flow windows, they provide faster response times to malware activity compared to higher tiers. On the other hand, higher tier classification modules are much more resilient to noise and are more precise than lower tier modules with non-burst malicious flows, as we will see in Section 6.6.2.

6.4.1.1 Flow Extraction and Encoding

The extraction module of MalPhase uses multiple sliding windows, one for each tier, to extract a subset of flows from a larger sequence. Each individual window is then encoded before being forwarded to the tier's classification module, as illustrated in Figure 6.3. The encoding step filters the raw network flows, excluding or transforming features that could bias the classifier. For instance, most malware are coded to communicate with a fixed set of IP addresses to download additional malicious code, or to exfiltrate stolen data to the malware author. If a classifier was trained with IP address as a feature, it is likely that the model would assign a heavy weight to it and become biased, since IPs tend to be unique per malware family. While such a classifier would probably exhibit high performance on the testing portion of the datset it was trained on, it wouldn't be able to maintain such performance if a new version of the same malware family were to use different IPs for communication. For these reasons, in the encoding phase we only use features that are harder to spoof and linked to the behaviour of the malware itself. We use features that are included in the standard definition of a network flow, extended by the round-trip time and Shannon entropy of packet payloads. In particular, we extract the following flow features:

- flow duration
- round-trip time
- IP protocol used
- connection towards local or public IP (boolean flag)

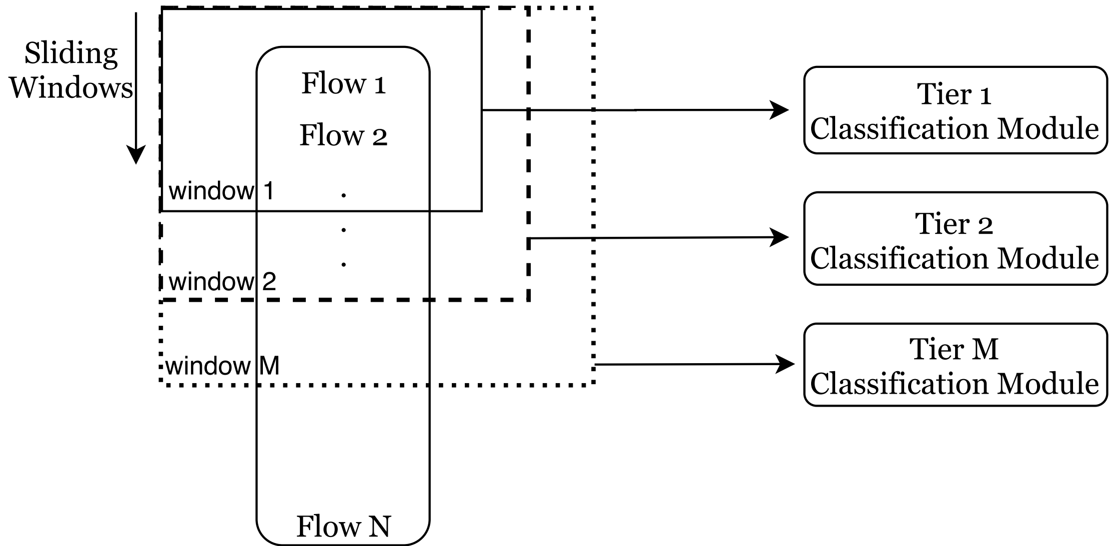


Figure 6.3: MalPhase tier system. MalPhase flow features are extracted using M sliding windows of increasing size, and successively fed to a window-specific Classification Module.

- destination port
- number of packets sent throughout the flow lifespan
- number of bytes sent throughout the flow lifespan
- number of packets received throughout the flow lifespan
- number of bytes received throughout the flow lifespan
- sent packet payload entropy
- received packet payload entropy

6.4.2 Denoising and Classification

In a real world setting, the traffic generated by an infected host will include both malicious flows and flows generated by benign programs at the same time. MalPhase is designed to be resilient to varying amounts of benign flows (called *noise*) in the network traffic, thanks to the use of a denoising autoencoder [159]. Denoising autoencoders are a type of autoencoder that is trained to remove noise from samples

belonging to a given distribution. Denoising autoencoders are notably used in image classification, where they are able to filter surprising amounts of noise and reconstruct the underlying image [160, 161]. Similarly, in MalPhase each tier uses a denoising autoencoder to filter benign noise and extract clean malware features, which are then passed to the DNN classifier of the appropriate phase. It is worth noting that the autoencoder does not simply remove benign flows from a given sample, leaving only malware flows. Rather, it compresses a sample in a robust, compact representation that preserves malware-related flow features, while filtering out most benign-related flow features. It is this compact representation that is then used by the DNN classifiers. In MalPhase, the autoencoders are trained by taking as input noisy malicious traffic samples, obtained by randomly injecting benign flows in a purely malicious sample, and learn to output the original clean sample without noise. At inference time, only the encoder part of the autoencoder is used, which is what provides the compact feature representation discussed above (see Figure 6.2).

6.5 Datasets

We make use of various datasets to train, test and evaluate our system. We acquire these datasets from public and private sources and seperate them into malicious and benign. In contrast to benign datasets, malicious datasets (Table 6.3) contain hashes of malware binaries (from here on, *hashes*).

ClamAV is an open-source antivirus engine, which features a database of malware hashes to check against. We harvest most of its released hash lists to form the ClamAV dataset. Ember is an open dataset of hashes of malicious Windows executables created as a benchmark for machine learning models. It contains malicious, benign and unlabeled hashes. For our needs, only malicious hashes are taken as Ember Malicious dataset.

The GT Malware dataset is a daily feed of flows that come from a number of newly observed malware by the Georgia Tech Information Security Center. The period considered in our analysis ranges from 1st September 2018 to 30th November 2019.

Access	Dataset	Hashes	VirusTotal Pcaps	AVClass Pcaps	Flows
public	ClamAV [162]	3,447,223	1,289,442	1,214,760	229,045,355
public	Ember Malicious [163]	800,000	453,467	436,166	26,568,413
quasi-restricted	GT Malware [164]	12,739,759	10,288,165	10,080,539	658,835,290
public	MalRec [126]	66,292	40,731	37,776	12,273,572
public	MalShare [165]	3,410,439	1,313,244	1,274,333	32,311,660
mixed	Misc.	1,955,923	937,690	648,570	28,784,436
public	VirusShare [166]	3,147,748	2,339,362	2,182,300	43,810,096
public	VX Underground [167]	405,803	95,023	87,629	1,489,459
	Total (unique)	13,920,730	15,369,738	14,644,135	996,712,027

Table 6.3: Overview of malicious datasets.

Access	Dataset	Flows
public	CTU [168]	546,490
private	Research Server A	70,319,185
private	Research Server B	5,938,924
	Total	76,804,599

Table 6.4: Overview of benign datasets.

From this dataset we only use the published malware hashes, for which we download pcaps from VirusTotal, but do not use the associated network flows that are already part of this dataset. This is done for consistency reasons, as we use VirusTotal captures for all hash sources in order to work on a standardized dataset. MalRec is a small dataset created as a result of a running malware on a dynamic analysis platform, which was collected over a two-year period. MalShare is a community-driven open repository of malware samples, which features over 3 million hashes.

Similarly, VirusShare and VX Underground are virus sharing websites that aim to help security researchers analyse selected strains of malware. The Miscellaneous (Misc.) dataset is created as an aggregation of hashes from a number of other public sources, too small to be considered as separate datasets. As mentioned previously, we only collect malware hashes from the listed malware datasets.

The last row in Table 6.3 represents the sum of unique amounts of hashes, pcaps and flows. This is due to the fact that there are duplicate hashes across datasets, which then propagate to duplicated pcaps and flows. It is worth noting that the increase of total VirusTotal Pcaps with regards to total hashes is due to the fact that malware is often run in more than one sandbox and in those cases for one hash, there is more than one pcap.

We check hashes of each malicious dataset against the VirusTotal database and download the corresponding antivirus reports, which list detection results from over 70 antivirus vendors, as well as network packet captures (pcaps) provided by their API. While these pcaps are a result of malware being detonated in a number of dynamic analysis sandboxes, not all of them are available on VirusTotal. This is due to the lack of binary files for a given hash uploaded by users to VirusTotal.

Having this in mind, the number of VirusTotal pcaps in Table 6.3 is in some cases significantly smaller than the total number of hashes in a dataset. Following this trend, the number of usable pcaps is further reduced by feeding the VirusTotal antivirus reports to AVClass [127] - a malware labeling tool capable of computing malware family names from a number of labels given by antivirus engines. Since not all pcaps belong to a known family (to antivirus engines) or there is no clear consensus between antivirus labels, they are labeled as singletons and filtered out in the dataset creation process.

As the last step, we convert pcaps to flows by using Yet Another Flowmeter (YAF) [169] - a suite of flow metering tools. The parameters used, ensure that a reasonable number of flows is obtained (--idle-timeout 30 --active-timeout 300), and additionally compute entropy of packet payloads (--max-payload 2048 --udp-payload --entropy). The resulting flows are in a bidirectional format, meaning that there are separate fields for forward and reverse directions of transferred number and sizes of packets, as well as their payload entropies. What sets our described malicious dataset creation method apart is the fact that it comes from a single source, which makes it more standardized than the regular approach of combining different datasets captured under varying conditions and settings. By eliminating such variables, we improve the quality of the dataset, which translates to higher quality of trained models.

Benign network traffic is the second part of our dataset. As shown in Table 6.4, it comprises of a number of normal captures from the publicly-available CTU dataset from the Malware Capture Facility Project, and network traces from our research servers, spanning from 6th December 2019 to 18th July 2020 for server A and from 10th March 2020 to 18th July 2020 for server B. In the case of CTU, flows are created from pcaps, and in the case of our research servers we capture the flows directly, by using YAF with identical parameters to those outlined for the malware datasets. This ensures that all resulting flows in both malicious and benign datasets are homogeneous by being a product of the same tool used with matching parameters.

Type	Hashes	Flows	Families (hashes %, flows %)
adware	1,117,311	26,198,428	directdownloader (0.31%, 7.12%), downloadguide (24.96%, 25.14%), hotbar (15.74%, 11.4%), inbox (7.41%, 3.96%), installcore (13.63%, 17.42%), playtech (4.88%, 6.17%), softcnapp (30.85%, 23.77%), softonic (1.75%, 1.72%), techsnab (0.43%, 3.26%)
ransomware	1,459,889	357,716,434	cerber (0.89%, 12.03%), deshacop (0.01%, 0.09%), sage (0.01%, 0.21%), virlock (98.73%, 87.41%), wannacry (0.35%, 0.23%)
trojan	2,063,369	276,427,311	bublik (0.87%, 2.15%), byfh (0.31%, 0.36%), cycbot (0.11%, 0.52%), delf (16.58%, 10.66%), mudrop (0.92%, 0.97%), ramnit (2.79%, 0.3%), razy (2.6%, 0.78%), scar (3.7%, 0.55%), shiz (2.57%, 9.36%), ulise (3.5%, 0.77%), unruy (18.19%, 2.21%), upatre (32.1%, 66.62%), vtflooder (6.08%, 2.75%), zbot (7.73%, 1.42%), zusy (1.88%, 0.5%)
virus	192,924	5,745,615	pioneer (19.15%, 15.29%), sality (71.67%, 76.32%), viking (9.17%, 8.37%)
worm	351,987	208,691,955	allaple (22.51%, 93.56%), drolnux (4.77%, 0.17%), mydoom (23.91%, 4.63%), socks (30.99%, 0.93%), warezov (8.67%, 0.45%), windef (9.12%, 0.23%)
Total	5,185,480	874,779,743	

Table 6.5: Overview of families selected for the evaluation.

Family	Hashes	Flows
autoit	126,036	1,698,697
banload	17,141	609,371
fareit	49,445	709,474
goldun	496	390,957
upantix	11,368	511,520
virut	1,338,470	24,676,037
Total	1,542,956	28,596,056

Table 6.6: Overview of unseen families.

The Malicious flows, that we obtain by converting pcaps of known malware families (as identified by AVClass), are processed further by filtering out flows that are related to sandbox artefacts (i.e. traffic that was present in each packet capture, not related to the activity of malware). The resulting flows are assigned to family-based groups. We select those families that generate a sufficient number of flows for our classifiers. In the end, there are 38 malware families (Table 6.5), which we arrange into respective malware types by consulting reports from antivirus vendors. Additionally, we choose six malware families for the evaluation of MalPhase on unseen samples (Table 6.6). With regards to composition of the final malware dataset, we attempt to abide by the experimental best practices outlined by Rossow et al. [170] concerning malware experiments.

6.5.1 Noise Injection

In order to test the effectiveness of the denoising autoencoder approach and the resilience of the MalPhase architecture to noise, we created a noisy dataset by injecting benign flows in our original malware dataset. In the original, clean dataset, each malicious sample is comprised only of malware flows. In the noisy dataset we add to each malicious sample varying amounts of benign flows in random positions, simulating a more realistic network trace. In particular, given a malicious sample with N malware flows, we add to it varying amounts of benign flows, increasing the size of the sample. We chose this method of injection for two reasons:

(i) It allows us to provide a more direct comparison between evaluation on clean samples and on mixed samples, since they both include the same amount of malware flows.

(ii) Adding benign flows on top of the existing malware flows is consistent with evasion techniques used by current malware [142].

The injected benign flows (from here on, referred to as *noise*) are randomly sampled from a set of 76 million benign flows from our research servers and the CTU dataset. We use this noisy dataset in all experiments in Section 6.6.2.

6.6 Evaluation and Results

In this section, we present our evaluation of MalPhase under three main sets of conditions, and aim at answering the following questions:

(i) Given samples with either only benign or only malicious flows (*clean samples*), what is the performance of each of MalPhase phases?

Evaluating MalPhase in ideal conditions allows us to define a baseline performance for the system. Whilst results on clean samples are not representative of real-world performance, they are indicative of the usefulness of MalPhase in a closed testing environment, such as a sandbox test for an unknown malware family, for instance.

(ii) How resilient is MalPhase to samples with mixed benign and malicious traffic (*noisy samples*)?

In a real-world setting, a network traffic sample will contain both benign and malicious flows interleaved with each other. Evaluating the performance of MalPhase on noisy samples provides a better understanding of the system's performance in real life, where it is expected to be resilient to a high degree of noise generated by the various benign applications running beside the malware.

(iii) How well does MalPhase perform on unseen malware families?

Given that new malware families emerge everyday, it is important that MalPhase can detect families that were not part of the training set, even when noise is injected.

In Section 6.6.1 we investigate question (1) and evaluate MalPhase using only clean samples, with no benign traffic mixed in. In Section 6.6.2 we investigate the resilience of MalPhase to noisy samples, using mixed benign and malicious flows. In Section 6.6.2.4 we study how well MalPhase binary classification generalizes to unseen malware samples. In Section 6.6.3 we analyse the robustness of different tiers of MalPhase to noise injection. For all experiments in this section, each sample for a given MalPhase tier contains the following number of malicious flows:

- Tier 1: 10 malicious flows.
- Tier 2: 20 malicious flows.
- Tier 3: 30 malicious flows.
- Tier 4: 40 malicious flows.

For all the experiments in this section, we trained the classification modules using the families presented in Table 6.5. In order to avoid excessive class imbalance, only a subset of the total flows were used for the families with a very large amount of flows. For these families, we randomly sample the flows from the set of overall flows.

6.6.1 Clean Samples Classification

In this section, we present the performance results of MalPhase classifiers under ideal conditions, with clean samples only. Clean samples are windows of traffic containing either only benign or only malicious flows. These conditions are representative of a controlled malware analysis environment, where programs are isolated and tested individually (e.g. on a specifically-crafted VM).

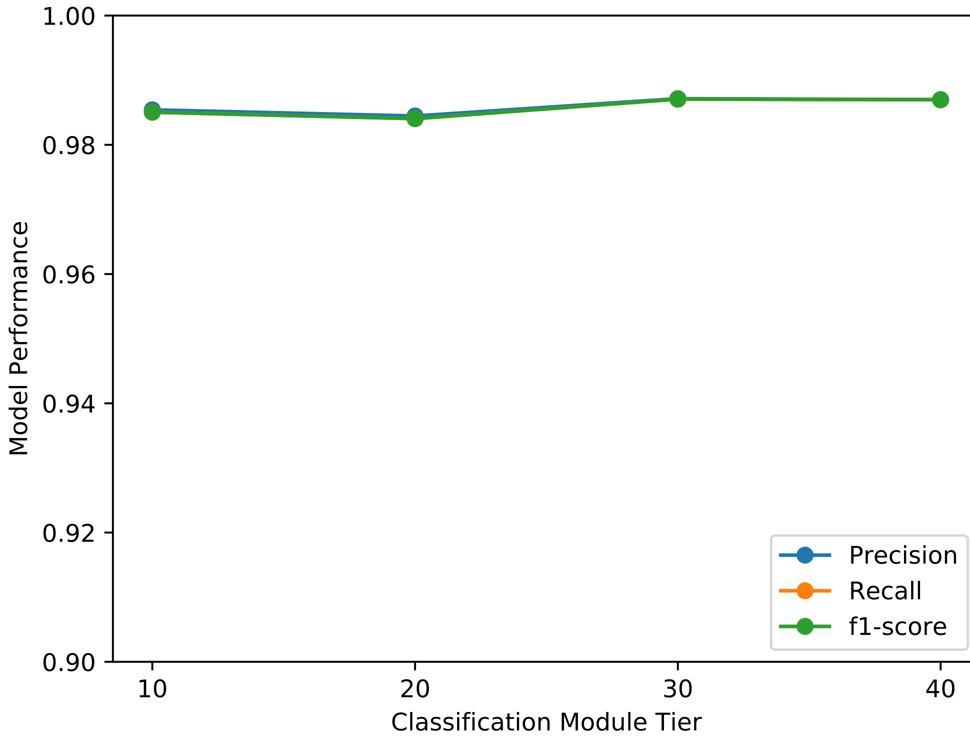


Figure 6.4: Precision, recall and F1-score for different tiers of the Binary Classification Modules.

6.6.1.1 Phase 1: Binary Classification

In Figure 6.4, we illustrate the performance of the Binary Classification Module of MalPhase on clean samples. The binary classifier exhibits very high performance in terms of the F1-score for both classes across all window sizes, with minor improvements as the window size increases.

The negligible performance difference between low and high tier classification modules is somewhat expected in this particular setting, as the evaluation is carried only on clean samples. The tiered architecture of MalPhase is designed to detect both short-term burst of malware flows, thanks to the low-tier classifiers, as well as constant, more regularly-occurring malware flows with high tier classifiers. Since samples for this evaluation contain either benign-only or malware-only flows, all malware samples can be effectively considered as bursts of malware traffic, that is quickly detected by classifiers of any tier. Moreover, higher tier classifiers are

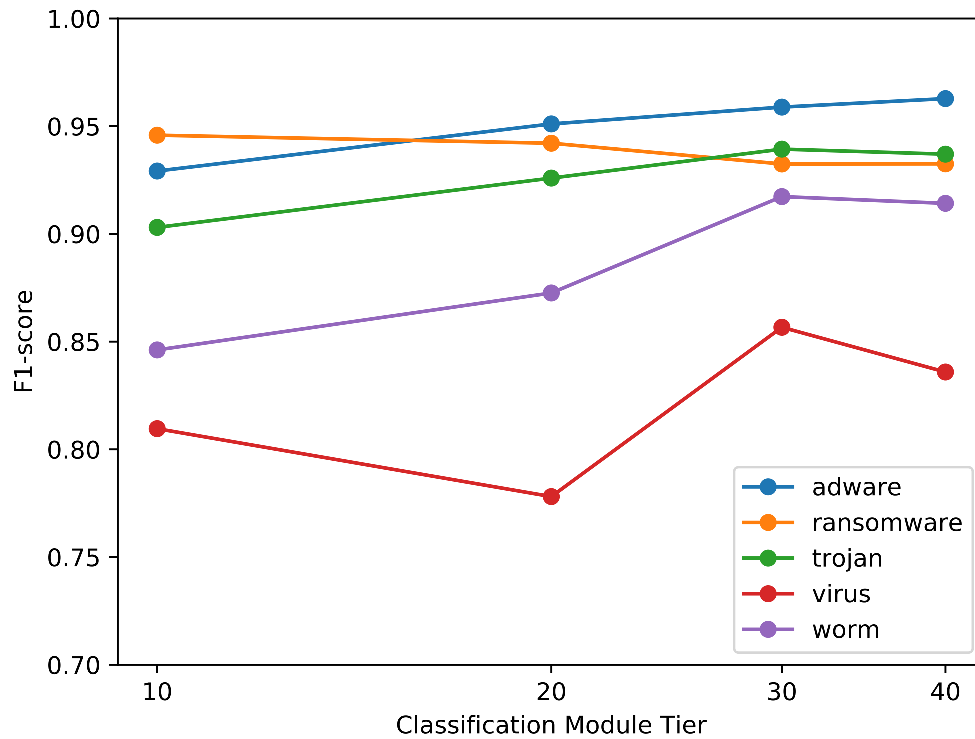


Figure 6.5: F1-score for different tiers of the Malware Type Classification Module.

designed to exploit longer term correlations between malware flows compared to lower tier classifiers, allowing them to more precisely classify samples that exhibit similar short-term patterns. However, when samples from different classes present very distinct behaviours — as is the case with benign and malicious network traffic — the models are able to easily distinguish them with only few flows, resulting in lower tier classification modules performing as well as higher tier ones.

As we will see in Section 6.6.2 and Section 6.6.3, when considering noisy samples higher tier classifiers tend to preform better than lower tiers as the noise ratio increases.

6.6.1.2 Phase 2: Type Classification

The Type Classification Module immediately follows the binary classification, taking malicious samples as input and categorizing them as one of five malware families: adware, ransomware, trojan, virus and worm. Figure 6.5 presents the performance

of MalPhase's type classifier in terms of F1-score for each malware type. Overall, all classes have very good performance, with F1-score >= 80% in all cases, above 90% for trojan, ransomware and adware and with a weighted average F1-score of $\sim$ 91%. As could be expected, classification performance can vary considerably between different types. In particular, the performance of ransomware classification is much higher than in the previous chapter (Figure 5.4). While this could be attributed to the improved method that MalPhase presents, it is also due to the size of datasets used in MalAlert and MalPhase. In the former the ransomware type constitutes only 1% of flows of the whole malware dataset (Table 5.1), whereas in the latter case ransomware traffic is the largest group in the malware dataset constituting over 40% of all flows (Table 6.5). The classifier performs noticeably worse on identification of viruses and worms compared to the remaining types. This behaviour can be partiallly explained by the relative under-representation of these two classes in the dataset: we had three usable virus families, accounting for $\sim$ 5.8% of the training data, and six usable worm families, accounting for $\sim$ 11% of the training data. Another contributing factor is that assigning a type to a malware is not as clear-cut as it might appear. In recent years, malware has become increasingly complex, resulting in samples that implement behaviours from different malware types. This fact can contribute heavily to the misclassification of under-represented classes, since given a sample with cross-type behaviour, the classifier is more likely to classify it as the most represented class.

6.6.1.3 Phase 3: Family Classification

The last phase of MalPhase uses multiple type-specific classifiers to categorize the malware family, hence the results are averaged for all families belonging to a given type and displayed as types. This is to make the results of the experiment more readable. In Figure 6.6, we show the performance for each of the five family classifiers in terms of F1-score, which is different from the previous experiment in Section 6.6.1.2 where there was only one classifier that classified broadly to malware types which

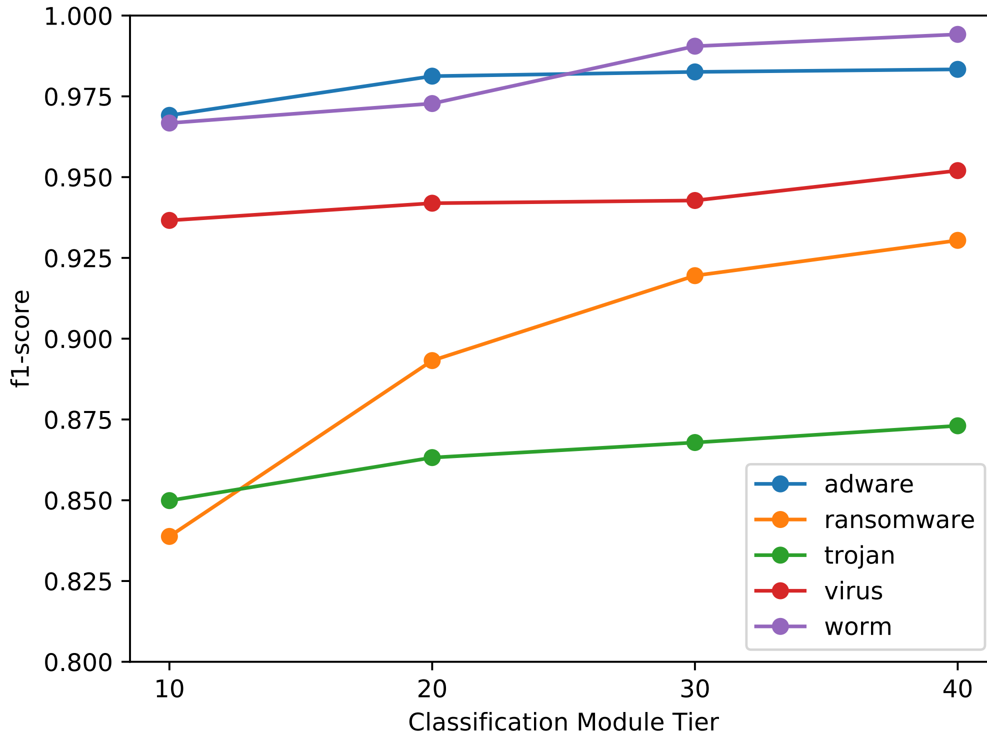


Figure 6.6: F1-score for different tiers of the five Malware Family Classification Modules.

where its classes. As we can see, most malware families are easily distinguished regardless of the classification module tier. The exception is ransomware families, with performance increasing almost by 10 percentage points going from window size 10 to window size 40. This trend is indicative of the fact that on shorter flow sequences, different ransomware families exhibit behaviours that are very similar. However, as more flows are considered, the behaviours become increasingly divergent, resulting in increased performance for higher tier classification modules.

What is worth noting is the performance of malware families the belong to the virus malware type, which is much higher than in Phase 2. This is mostly attributed to the amount of families that constitute the virus type. It contains only three families with one containing over 75% of flows for this type.

Interestingly, classification performance for adware families is extremely high, with F1 consistently above 97% regardless of the fact that we have 9 adware families in our dataset. On the other hand, malware families belonging to the trojan type

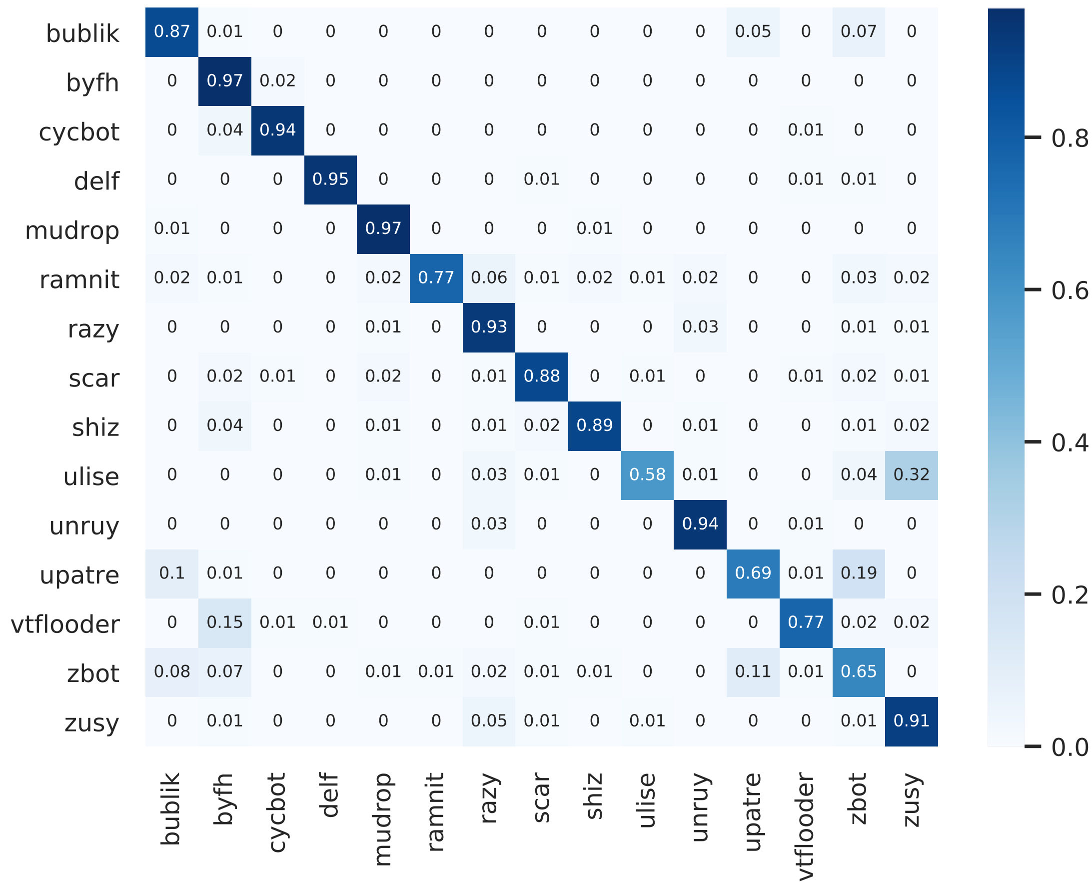


Figure 6.7: Confusion matrix of tier 1 family classifer for trojan family.

appear to be harder to reliably classify for all classification modules, regardless of tier, with F1 hovering around 86%. Upon closer inspection, there are three families which are consistently misclassified by our model with recall between $60 - 70\%$ (see Figure 6.7): Ulise, Upatre and ZBot. ZBot is an alias for the famous ZeuS botnet [171] which compromises computers using the ZeuS trojans. Upatre is a trojan downloader which, upon installation, downloads and executes additional malware on the target machine. One malware family that can be downloaded by Upatre is, in fact, ZBot itself [172], which explains why the model tends to consistently misclassify samples of these two families. It's interesting to note that the number of instances in which Upatre is classified as ZBot and vice versa are rather close. This can be attributed to the fact that both classes are approximately equally represented in our dataset, resulting in similar samples classified approximately

half of the time as one class and half of the time as the other.

Ulise appears to be a "traditional" trojan, which when downloaded on a target machine sits in the background stealing sensitive user data. Zusy, on the other hand, is an atypical piece of malware that uses a vulnerability in Office as a vector to infect a machine and steal user banking data. While having a similar goal, the two families are distinct and there doesn't appear to be any link between the two. Upon closer inspection, however, we can notice that in some cases certain antivirus engines mislabel Ulise samples as Zusy [173, 174]. Since our pipeline relies on antivirus labeling and on the AVClass tool to assign a sample to a family (see Section 6.5), it seems likely that some samples were mislabeled due to incorrect antivirus or AVClass labeling. This effectively results in our model being trained on some Ulise samples that are labeled as Zusy, lowering the ability of the classifier to distinguish the two classes.

It is worth noting that during our initial dataset processing and malware family extraction, we did not notice these correlations between different classes. While investigating our apparent model "misclassification", we decided to study the correlation between these families in greater depth. This is a compelling indication of the usefulness of MalPhase, which allowed us to understand that some families (e.g. Upatre) were actually using other families as components (ZBot), resulting in very similar behaviours.

6.6.2 Noisy Samples Classification

This section presents the performance of MalPhase after we inject varying amounts of noise in our dataset (see Section 6.5.1). In this dataset, each malware sample is a noisy sample, containing a mix of interleaved malicious and benign flows. Evaluating MalPhase with noisy samples provides a closer approximation to real-life system performance, where malware traffic is interleaved with traffic from benign applications. It also gives a better understanding of the robustness of the different classification phases. We create noisy samples by randomly adding benign

flows in a window of malware flows, with varying ratios of benign-to-malicious traffic (*noise ratio* from here on).

For instance, for the tier 1 classifiers that we designed to work with at least 10 malicious flows, each sample is comprised of 10 malicious flows in addition to a varying number of noise flows. For these experiments, we use noise ratios ranging from 0.2 (i.e. benign flows are 20% of malicious flows) up to 2 for type and family classification (i.e. benign flows are 200% of malicious flows), and up to 8 for binary classification (i.e. benign flows are 800% of malicious flows). Lower noise ratios are representative of bursts of malware traffic, while higher noise ratios simulate a malware that slowly and consistently performs network operations. For all experiments in this section, noise used during the training phase is sampled from a different benign dataset than noise used during testing phase. We do this, to make sure that the results were not affected by the model overfitting on the distribution of the specific noise dataset used.

Assessing MalPhase performance with noisy samples allows us to evaluate the effectiveness of our denoising autoencoder approach in extracting robust malware-related features to use for classification, as well as providing a much better understanding of the expected performance of MalPhase in a real-life setting.

6.6.2.1 Phase 1: Binary Classification

In Figure 6.8, we show the classification performance of the Binary Classification Module for tier 3, using noisy samples with varying noise ratio. The performance of the classifier decreases in a relatively stable manner between noise ratio 0.2 and 1, with recall and F1 going from $\sim 93\%$ down to $\sim 83\%$ and precision from $\sim 93\%$ down to $\sim 85\%$. For higher noise ratios the decrease in performance becomes less marked, with F1 decreasing to $\sim 80\%$ at noise ratio 2 and further down to $\sim 77\%$ at ratio 4. This behaviour indicates most benign and malicious samples are distinct enough that even with a four-fold increase in noise ratio (from 1 to 4) the model can still produce very good results, with only a 6% performance decrease. Beyond

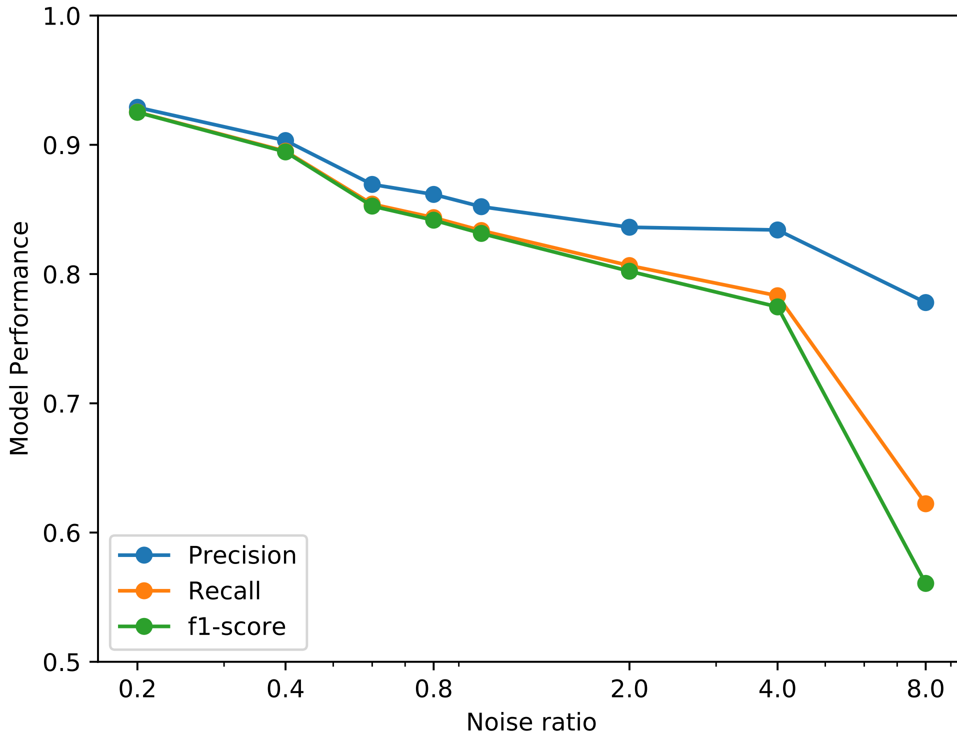


Figure 6.8: F1-score for the tier 3 Binary Classification Module, with varying ratio of benign-to-malicious traffic. Noise ratio plotted on a logarithmic scale.

this point, we see that further increases in noise ratio have considerably larger impacts on performance, with F1 decreasing from 77% down to 56% at noise ratio 8. This behaviour indicates that the autoencoder begins to struggle to filter out the benign noise, impacting the ability of the classifier to reliably tell apart benign and malicious samples. This general behaviour holds across all tiers, with higher tiers performing consistently better than lower tiers, as we will see later in Section 6.6.3.

6.6.2.2 Phase 2: Type Classification

In Figure 6.9, we present the performance of the tier 3 Type Classification Module. The classifier maintains good performance on the ransomware, trojan and worm classes up to noise ratio 1, and acceptable performance for higher noise ratios. The adware and virus classes appear to be much more sensitive to injected noise: while

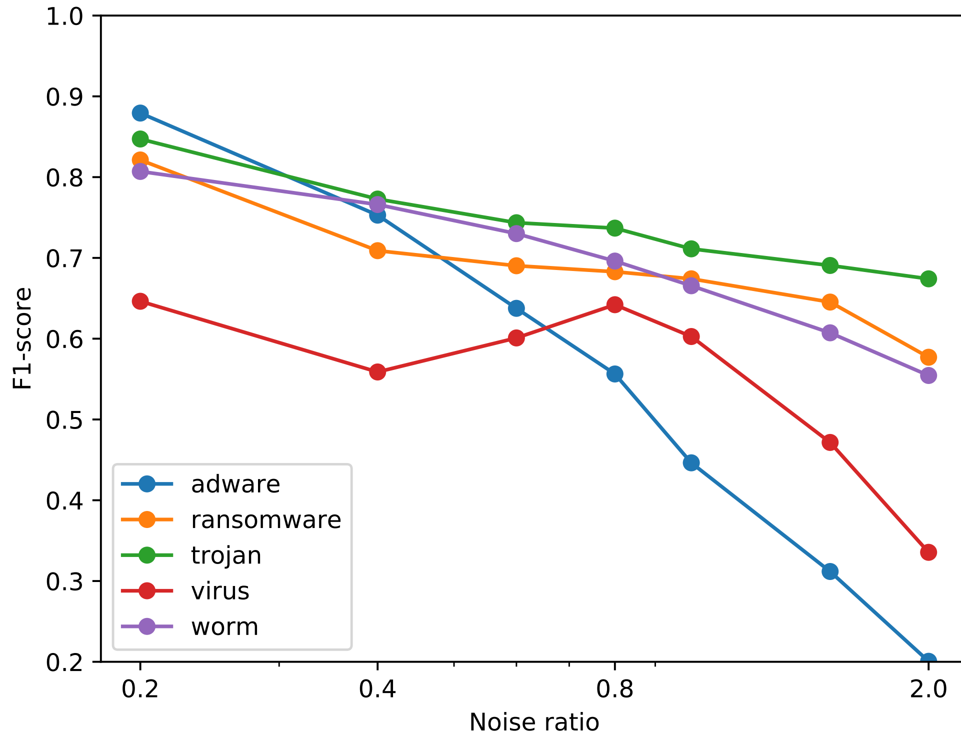


Figure 6.9: F1-score for the tier 3 Type Classification Module, with varying ratio of benign-to-malicious traffic. Noise ratio plotted on a logarithmic scale.

virus remains relatively stable for noise range $[0.2, 1]$, its performance is much lower compared to that on clean samples (see Figure 6.5) and rapidly decreases for higher noise ratios. Adware performance on the other hand decreases much more sharply, losing over 30 points between noise ratio 0.2 and 0.8, and decreasing even further for higher noise ratios. This result was somewhat expected for the virus class, as it is the least represented class in our dataset with just three families. On the other hand, adware is the second most represented class in the dataset with 9 families, therefore we were not expecting the classifier to perform so poorly on it. When we look at the confusion matrix for adware, we see that it is often classified as trojan. This indicates that adware traffic is close to trojan traffic from the point of view of network flow features, and that some noise is sufficient to confuse the feature vectors enough that our model is unable to classify adware consistently.

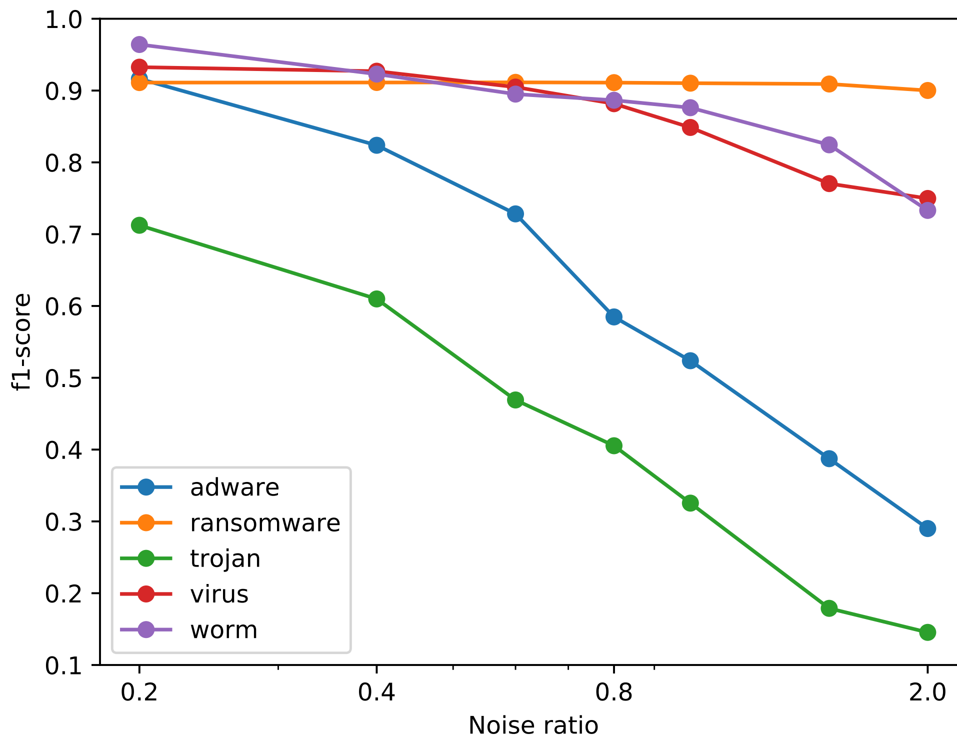


Figure 6.10: F1-score for the five tier 3 Family Classification Modules, with varying ratio of benign-to-malicious traffic. Noise ratio plotted on a logarithmic scale.

6.6.2.3 Phase 3: Family Classification

In Figure 6.10, we show the performance of the various Family Classification Modules for tier 3. The classifiers for worm, virus and ransomware are performing the best, with F1 scores around 90% when the noise ratio below 1, and consistently above $\sim 75\%$ for when the noise ratio up to 2. Trojan and adware classifiers on the other hand are much more sensitive to noise, with performance decreasing uniformly in the noise ratio interval $[0.2, 2]$. This behaviour is unsurprising, given that adware and trojan are the malware types for which we have the most families, which makes the job of the family classifier considerably more difficult. It is easy to see how, when considering ten or more target classes, any amount of noise in the input sample can severely impact the performance of the classifier.

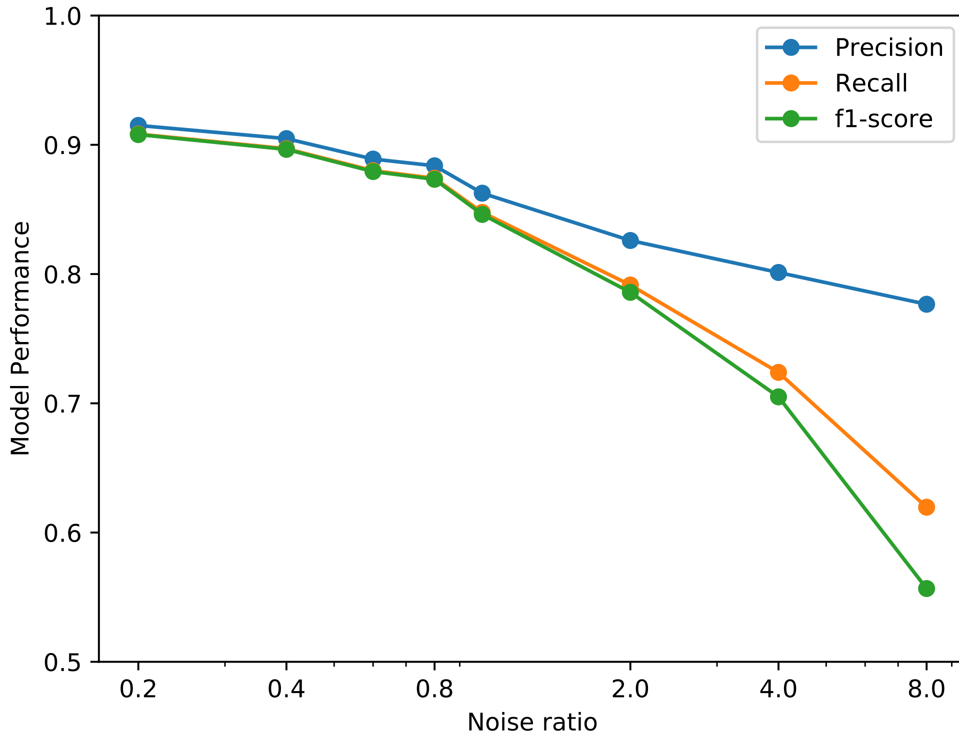


Figure 6.11: Precision, recall and F1-score on unseen malware families for the tier 3 Binary Classification Module, with varying ratio of benign-to-malicious traffic. Noise ratio plotted on a logarithmic scale.

6.6.2.4 Phase 1: Binary Classification of Unseen Malware

In this section, we examine the performance of our malware detector, the Binary Classification Module, on unseen family samples. Unseen samples are families that the classifier did not see during the training phase. Such experiment provides a better understanding of the performance the classifier would have on *new, unknown malware families.* All experiments in this section are performed with noisy samples, as described in the previous section. We are therefore testing our detector in the worst case: network traffic from an unknown malware with mixed in benign traffic. For these experiments, we use the families described in Table 6.6. As we can see in Figure 6.11, the performance of the tier 3 Binary Classification Module on unseen samples is comparable to that presented in Figure 6.8 on known samples in the noise ratio interval $[0.2, 2]$, with F1-score in the range $[91\%, 78\%]$. For noise ratio

above 2, the spread in performance between known and unseen samples begins to increase, with a performance delta of 7 points at noise ratio 4, before decreasing again at noise 8 where performance on know and unseen samples is comparable. This behaviour indicates that higher levels of noise have a larger impact on unseen sample classification compared to known samples. However, this penalty eventually evens out at extremely high noise levels, where the classifier performs comparably on both known and unseen samples. Furthermore, the small difference between results on known and unseen samples suggest that improvements to performance for known noisy binary classification would most likely carry over to noisy unseen sample classification as well.

6.6.3 Tier Comparison

In our last experiment, we assess the robustness to injection of noise for different MalPhase tiers. As discussed in Section 6.4, the tier system of MalPhase is designed to capture different malware behaviours: lower tiers require less malware flows to correctly classify samples compared to higher tiers, with tier 1 requiring just 10 malware flows compared to 40 for tier 4. This makes lower tiers suitable for the detection of quick bursts of malware traffic amidst some benign flows. Higher tiers, on the other hand, monitor a larger time horizon compared to lower tiers and are designed to be robust to higher ratios of benign-to-malicious traffic. As we can see in Figure 6.12, all tiers show very good performance at noise ratios between 0.2 and 1, with the tier 1 classifier barely falling under the 80% F1-score mark. As the noise ratio increases, lower tier classifiers reach a point where performance begins to decrease sharply. Conversely, performance for higher tier classifiers decreases more gradually and reaches much higher F1-score at the maximum noise ratio. This behaviour is consistent with the rationale behind the multi-tier design of MalPhase, and shows that higher tier classifiers are more robust to noise than lower tiers, which are better suited for the detection of quick bursts of malware traffic.

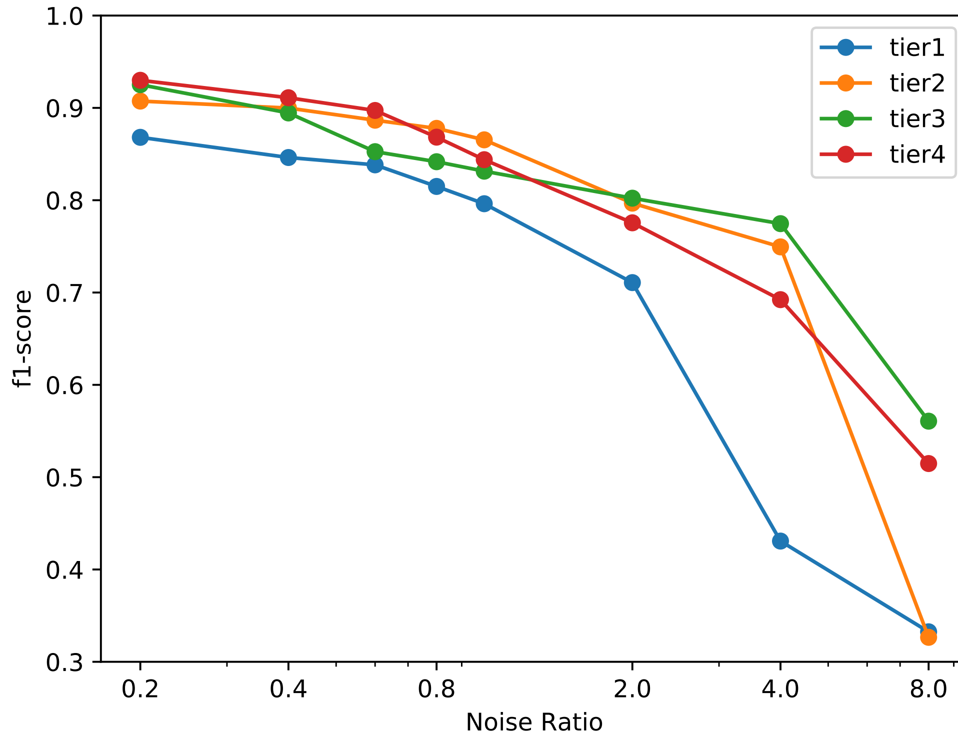


Figure 6.12: F1-score for Binary Classification Modules of different tier, with varying amount of noise ratio. Noise ratio plotted on a logarithmic scale.

6.7 Discussion and Limitations

In this section, we discuss the proposed system and outline current challenges. We talk about limitations of MalPhase with regards to its design, and the availability of data.

6.7.1 Malware Taxonomy

We found the lack of clearly defined malware taxonomies to be one of the obstacles in creating a robust malware dataset. A number of papers define ways to categorise malware. There is a general consensus that different strains or mutations of malware are classified into a family, while type (or class) is a more general category based on malware modus operandi. To add complexity to the process, sometimes one of those three terms (family, type and class) is confused for the other one. For instance, Rhode et al. [175] use the term 'family' when they talk about types;

AVClass [127] uses the term 'class' for its output, which is a malware family. The more high-level the categorisation, the more liberty and flexibility is assumed by paper authors. Some of them adopt similar malware types to ours, while combining malware with attacks in their taxonomy [22], or provide a simple taxonomy that divides malware into simple and self-reproducing [176]. Grégio et al. [34] assume yet another type taxonomy, treating viruses, worms and trojans as a single type. Additionally, antivirus vendors provide their own fixed set of malware types, which varies depending on the specific company. The most comparable type taxonomy to ours is the one presented by Cisco [157], with the addition of adware to the list of ransomware, viruses, worms and trojans. A notable and deliberate omission on our part, is the botnet type. Its main characteristic is the way in which it interacts with its owner, from whom it takes new commands. The term 'botnet' does not define the way of operation of malware or its behaviour (i.e. there are worm botnets, trojan botnets or virus botnets), which was the main criterion in our selected type taxonomy. Having said this, botnets are still a valid way of defining certain malicious programs, however we feel they belong to a separate taxonomy.

6.7.2 Network Flows

MalPhase uses bidirectional flows, which contain all standard features such as protocol, source and destination IP addresses and ports, packet sizes and counts. The flows are augmented with additional features: round-trip time of packets, and packet payload entropy. While the calculation of Shannon entropy requires access to packet payloads, it is still privacy-preserving. Due to those additional features our method does not operate on default flows (e.g. NetFlows) with the same classification performance. Additionally, we do not use IPv6 traffic in our datasets and evaluation, however including them would not require significant changes to our system.

6.7.3 Noise Injection

Noise injection provides a useful indication of the robustness of the classification approach and its applicability in a real world setting, but it also has some limitations. Whilst the noise is sampled from real world benign traffic, randomly injecting it in a window of malicious flows does not necessarily preserve the temporal properties that the benign data would have in the real world. Moreover, there is no guarantee that the benign traffic in our dataset captures the true distribution of benign traffic in general. However, given the objective impossibility of obtaining large amounts of real world traffic with per-flow labeling, noise injection is an acceptable compromise to assess the resilience of MalPhase.

6.7.4 Unseen Type Classification

While our evaluation shows that MalPhase is able to detect unseen malware samples with high performance, we were unable to replicate the same result for type classification. This limitation stems from an insufficient amount of families left after dataset processing (see Table 6.5). It is possible that the amount of available families is insufficient for the model to learn a type representation that can generalize to unseen samples. Furthermore, the problem of an insufficient amount of families is compounded by the fact that the distribution of malware families is not uniform across types. We hypothesise that with additional training data from other malware families, MalPhase could achieve satisfactory results in unseen type classification. However, a higher number of families per type would also likely impact the performance of the classifier in the family classification task.

6.7.5 Sandboxes

While dynamic analysis of malware is a separate field that offers most detailed results of the activity of a binary, there are important factors that make an analysis on a large set of malware valuable. In an ideal scenario, a large set of malware

binaries should be manually examined by a number of reverse engineers who could attest to their maliciousness or lack thereof. This is to overcome the scenario where malware detects automatic tools, such as sandboxes, it is used in and ceases its malicious behaviour. When it comes to collecting malware artefacts of a certain type (network traffic in our case), it is crucial to limit the variation of data added by the use of different tools. For example, the use of a number of sandboxes would only add uncertainty with regards to collected network traffic, as malware could operate in one of them and detect the other one. However, even running malware in one sandbox would be prone to capturing malware traffic as malicious (because it is coming from a malware binary) while malware has not activated its malicious operation due to the mentioned sandbox detection or the fact that C&C servers are no longer active. The latter could be partially overcome by analysing the date when a malware binary was first seen by the antivirus community to determine the likelihood of its current active operation in the wild. In MalPhase, we avoid the problem of labeling all malware traffic as malicious by determining which network flows in the whole malware dataset are repetitive and subsequently filtering them out. Such detection is done using a simple algorithm that counts the occurrences of each network flow and examines what the percentage is of those repetitive flows in each binary. In the future, one could extend this and employ more advanced methods such as machine learning to arrive at the most optimal data filtering technique. As all of the used malware network traffic data is collected from sandboxes, our dataset is biased towards malware that does not employ anti-VM techniques [177, 178]. Moreover, since network traffic was captured in the sandbox only for a short length of time, malware that deploys delayed execution may not leave any forensic artefacts, and therefore be excluded from our dataset.

6.7.6 Evasion

Many malware variants today make use of evasion techniques in order to avoid detection. One such technique consists of injecting benign traffic to obfuscate the

malicious traffic generated by the malware itself [142, 145, 179]. As highlighted in Section 6.6.2, MalPhase shows good robustness to injection of benign traffic. However, injection of high amounts of benign traffic decreases the performance of the classifiers, with a more pronounced impact on type and family classification. Another class of evasion attacks that could be used by malware to avoid detection is mimicry. In [180] the authors show how a ransomware sample is able to mimic the behaviour of benign processes and completely evade detection. While the methods proposed by the authors were not directed at evading network detection, it appears feasible that a similar approach could be adapted to work at a network level.

6.8 Future Work

The current iteration of MalPhase uses a feed-forward autoencoder architecture, which accepts a fixed-size input and generates a feature vector for the DNN classifier. While our evaluation shows that this simple architecture can produce good results, it would be interesting to assess the performance of other architectures. In particular, since an encoded malware flow in a sequence can be seen as a word in a sentence, it would be interesting to more thoroughly explore recent natural language processing advances, such as GPT2 [181] and BERT [182], and apply some of the concepts to malware detection and classification.

Another avenue of research for future work relates to scalability. Currently, many of the families in our dataset could not be used for training as they did not generate enough flow data. While DNNs can generally surpass the performance of traditional machine learning architectures with enough training data, traditional architectures can reach comparable or even higher levels of performance when fewer training samples are available. It would be worth revisiting the architecture of MalPhase using different machine learning algorithms that require less training data, which would likely unlock a much larger number of families for training and testing. In this regard, one of the challenges would be to identify a suitable replacement for the denoising autoencoder.

6.9 Summary

We present MalPhase, a multi-phase system for the detection and classification of malware network traffic, and evaluate it on one of the largest datasets of malware flows to date. MalPhase is capable of detecting malware from aggregated network flow data, as well as classifying it into specific malware type and family. We show that MalPhase performs comparably or better than current state-of-the-art approaches on clean malware sample detection ($>$ 98% F1), while also broadening the scope of classification to malware type ($>$ 93% F1) and family ($>$ 91% aggregated F1). Furthermore, we evaluate and demonstrate the robustness of MalPhase classification to the injection of benign flows interleaved with malware network traffic. Finally, we show that MalPhase is able to detect unseen samples with performance comparable to that of known samples, even in the presence of noise.

7

Network Flow Analysis System and Binary File Format

Contents

7.1 Introduction 144
7.1.1 Contributions 146
7.2 Related Work 146
7.2.1 Packet-Level Traffic Collection 147
7.2.2 Flow-Level Traffic Collectors 147
7.2.2.1 Storage Formats 148
7.2.2.2 Indexing Methods 149
7.3 CompactFlow Format Design 150
7.3.1 CompactFlow File Header 151
7.3.2 Flow Binary Representation 154
7.4 Evaluation 157
7.4.1 Average Flow Size Experiment 160
7.4.2 Big Data Flow Storage 161
7.4.3 Port Analysis 161
7.4.4 CompactFlow Compression Experiment 161
7.5 Discussion 163
7.6 Summary 165

Network traffic monitoring has become fundamental to obtaining insights about a network and its activities. This knowledge allows network administrators to detect anomalies, identify faulty hardware, and make informed decisions. The increase of the number of connected devices and the consequent volume of traffic

poses a serious challenge to carrying out the task of network monitoring. Such a task requires techniques that process traffic in an efficient and timely manner. Moreover, it is crucial to be able to store network traffic for forensic purposes for as long a period of time as possible.

In this chapter, we propose CompactFlow, a hybrid binary format for efficient storage and processing of network flow data. Our solution offers a trade-off between the space required and query performance via an optimized binary representation of flow records and optional indexing. We experimentally assess the efficiency of CompactFlow by comparing it to a wide range of binary flow storage formats. We show that CompactFlow format improves the state of the art by reducing the size required to store network flows by more than 24%.

7.1 Introduction

In recent years, we have witnessed an astonishing evolution of networks in terms of complexity, variety, and versatility. An increasing number of devices have started to embed networking capabilities and to require Internet connection to provide their full functionalities. Hence, guaranteeing the connectivity of such devices has become fundamental to the operation of the entire networking infrastructure. In order to carry out this task, network administrators have to be provided with reliable tools to monitor traffic flowing through a network. In addition to that, administrators have to be able to investigate past events by retrospectively analyzing the state of a network at any given point in time. For this reason, it is necessary to archive network traffic in a fast and space-efficient way.

Monitoring networks at the granularity of packets offers perfect visibility of their state but also requires overwhelming computing resources and storage space to be devoted. While packet-level approach may have been possible in the early days of networking, it is infeasible in modern networks due to the increasing number of interconnected devices and the volume of data produced by them. Moreover, the ubiquitous adoption of encryption in network communication to protect user privacy has made packet-level traffic capturing obsolete since encrypted payloads do not

provide any meaningful information. Due to these limitations, network monitoring has shifted toward a network flow as a more coarse-grained representation of traffic data. A network flow comprises information of a communication from a temporal perspective as a five-tuple: protocol, source and destination IP addresses and ports. Differently from packet-level data, flows capture only metadata, such as the overall number and size of exchanged packets.

Flow exporters are devices in the flow creation process that capture and assign network packets to flows based on their five-tuple and within a temporal interval. Once flows are created, they are sent to a flow collector using a given export protocol. A flow collector is a device in charge of storing flow data for future use. The most popular export protocol is Cisco's NetFlow [6], which inspired the creation of the open standard IPFIX [7].

Over the years, a number of flow collectors have been proposed by networking companies and researchers. The main goal of such devices is to rapidly collect and store flows in such a way that avoids blocking the next oncoming flows. More importantly, they have to adopt a storage format that is efficient in terms of the size required and indexing to process future queries. Network administrators are constrained by the space available to store network traffic, thus older traffic has to be periodically deleted. For this reason, a space-optimized format saves storage space, which allows for keeping network traffic of longer periods for retrospective analysis. Unfortunately, our investigation of open-source flow collectors showed that they use an inefficient flow representation in their formats, even among the ones that favor storage efficiency over processing speed.

In this chapter, we present CompactFlow - a binary format to represent network flows that favors storage and processing performance while supporting indexing. In particular, the CompactFlow format relies on dynamic field sizes and is based on a linked list to store the contents of flows. This accounts for a significant reduction of storage size. In fact, experimental results show that CompactFlow files are on average almost 3 times smaller than the ones using binary formats of other

flow collectors, and 24% smaller than the ones using the binary format of the state-of-the-art System for Internet-Level Knowledge (SiLK) collector [183].

CompactFlow can be considered a hybrid binary format since it allows for customization according to administrators' analysis purposes:

(i) It supports additional indexing methods to increase the speed of repetitive queries.

(ii) It is possible to choose which flow fields or which specific values of a flow field to index.

Unlike database-based approaches, our solution allows for high-speed saving of flows without the risk of dropping them or resorting to sampling since the indexing can be done after successful storage. The design principles of CompactFlow join two best practices of storage (binary files) and querying (indexing), to have a robust system for network monitoring and processing of cybersecurity events.

7.1.1 Contributions

CompactFlow makes the following contributions:

(i) We present a binary file format to store network flows using less space than state-of-the-art approaches. Our format supports popular indexing methods to allow faster data processing in the security context.

(ii) We perform a thorough analysis of all open-source network flow collectors and a popular data serialization library by analyzing their binary formats.

7.2 Related Work

In the last two decades, many approaches have been proposed to monitor network traffic. This effort has been necessary to carry out management and security analyses on networks, such as identification of anomalies or failures, and detection

of attacks. Most of these analyses cannot be done in real-time, hence network traffic has to be stored in persistent memory in order to make it available when needed. For this reason, it is necessary to store and query network traffic efficiently. A first important distinction between storage approaches is related to the granularity of traffic collection: packet- and flow-level.

7.2.1 Packet-Level Traffic Collection

Collecting network data at packet-level provides fine-grained information about traffic but it requires fast dedicated equipment. Desnoyers et al. in [184] propose Hyperion, a system that relies on a log-structured file system that is optimized for writing data streams to store packet-level network traffic. This system indexes data stream segments via distributed multi-level Signature indexes. The authors claim to be able to write and index up to 1M and 200K packets per second, respectively.

Maier et al. in [185] propose to focus only on the part of the packet stream that may be interesting for a network intrusion detection system (NIDS). Hence, they present the TimeMachine system which applies a cut-off heuristic (i.e. it only considers the first N bytes) to reduce the size of the data stream to store.

Fusco et al. in [186] present PcapIndex which extends *Libpcap* by supporting rapid packet filtering via COMPAX compressed bitmap index [187]. By doing this, PcapIndex reduces the disk overhead and the response time of queries.

Unfortunately, the aforementioned methods are not suitable for large-scale networks since they do not scale on the number of devices connected. Moreover, such fine-grained information would require an overwhelming storage capacity.

7.2.2 Flow-Level Traffic Collectors

In order to cope with the shortcomings of packet-level traffic collection, the networking community has moved toward the collection of traffic information by aggregating packets into flows. In what follows, we present various solutions available to collect, store and access network flows.

Collectors	Storage Formats				Bidirec-tional
	Database		Flat Files		
	Row-based	Column-based	Binary format	Text format	
Argus [188]	✓		✓	✓	✓
flowd [189]			✓		
IPFIXcol [190]		✓			✓
nfdump [191]			✓		
pmacct [192]	✓			✓	✓
SiLK [183]			✓		
Vermont [193]	✓				✓

Table 7.1: Comparison of open-source flow collectors.

7.2.2.1 Storage Formats

Network flow collectors adopt several solutions to store flow-level network traffic in persistent memory. Such solutions can be divided according to the way they structure and index the data [56]. A popular data structure to store flows is a database. The advantage of databases is that such data structures automatically handle the information storage and indexing via a Database Management System (DBMS). Traditional Database Management Systems, such as MySQL and PostgreSQL, store the information by rows (*row-based databases*). In our case, a row represents an entire flow (i.e. all its fields). Two examples of flow collectors that use a row-based database to store network traffic are Vermont and pmacct. Regarding the queries, row-based databases offer good flexibility but they have poor performance in terms of data retrieval and new flow insertion time. Moreover, a row-based database is not storage-efficient since it requires considerable indexing.

For this reason, column-based databases have been proposed for network flow storage. Rather than to consecutively store entire flows, column-based databases store them in columns by flow fields. Examples of column-based databases are MariaDB ColumnStore [194] and bitmap indexing methods (e.g. FastBit [195], and COMPAX [187]). Indexing by columns decreases data retrieval time for queries

while maintaining good flexibility and moderate insertion time. In particular, FastBit is an order of magnitude faster than MySQL [196]. IPFIXcol is a collector that relies on FastBit. It supports IPFIX, bidirectional flows, and variable length fields. Unfortunately, the main shortcoming of column-based databases is poor performance in retrieving flows in their entirety. Moreover, such databases still have to maintain a reference to a specific flow (i.e. index) for each flow field resulting in overhead in storage size.

Another solution that aims to reduce storage space is to rely on *flat files*. A flat file typically stores data sequentially and does not embed any hierarchy nor indexing by default. For this reason, flat files do not offer query flexibility but they occupy much less space than a database [197]. Data in flat files can be represented in a text or binary format. Despite the portability of a flat file in text format, representing data in a binary format further reduces the storage size and the query response time. Examples of flow collectors that save network traffic in a binary format are Argus, flowd, nfdump, and SiLK.

In Table 7.1, we report several open-source network flow collectors and we compare them according to storage formats supported and whether they can represent bidirectional flows. It is worth noting that some collectors can use more than one storage format (e.g. Argus, and pmacct) and that the majority use flat file formats.

To perform a thorough comparison between our proposal and the state of the art, we analyse the binary formats used by collectors that allow storing network flows in flat files (i.e. Argus, flowd, nfdump, and SiLK). In Section 7.4, we show that our compact format outperforms all of them in terms of space efficiency.

7.2.2.2 Indexing Methods

Flat file formats are optimal for network flow storage because they save space and have a negligible computational overhead in inserting new flows. However, the limitation of this format is that it does not provide integrated indexing of the flows,

thus it lacks in query performance and flexibility. To cope with this shortcoming, researchers propose solutions to build indexes which offer a low retrieval time and require little storage. Typically, storing network flows from a flow exporter consists of two aspects: writing the flows to a flat file and building indexes of those flows. For this reason, most solutions rely on the multi-processing capabilities of modern computers [198, 199].

In the literature, researchers use different data structures to organize flow (or query) indexes [200]. For example, TelegraphCQ [201] stores indexes and results of queries via a modified version of PostgreSQL. Three other examples, GigaScope [202], MIND [203], and FloSiS [199] arrange indexes into trees, multi-level hashing tables, and Bloom filters, respectively. However, the most popular and best-performing approach leverages bitmap indexing. As an example, Reiss et al. [204] and Chen et al. [205] applied the concept of bitmap indexing (i.e. FastBit [195]) to improve the performance of TelegraphCQ [201] and TimeMachine [185] (applied on network flows), respectively. More recently, Xie et al. in [206] present Index-trie, a novel data structure to index flows that combines trees and bitmaps. Our CompactFlow format is designed to be hybrid. This means that it is able to support a variety of flow indexing methods.

Several approaches also propose fast compression/decompression algorithms to be applied to both stored flow data and indexing data structures to further reduce the storage size. Fusco et al. propose NET-Fli [187] and RasterZip [207] systems that compress on-the-fly flow data streams via compression algorithms, e.g. Lempel-–Ziv-–Oberhumer (LZO) and Run-Length Encoding (RLE) [208]. Unfortunately, even the fastest compression algorithms generate computational overhead which translates to increased processing time.

7.3 CompactFlow Format Design

Network flow data is comprised of information relating to communication between two hosts on a network. Every flow consists of core fields (traditionally called a

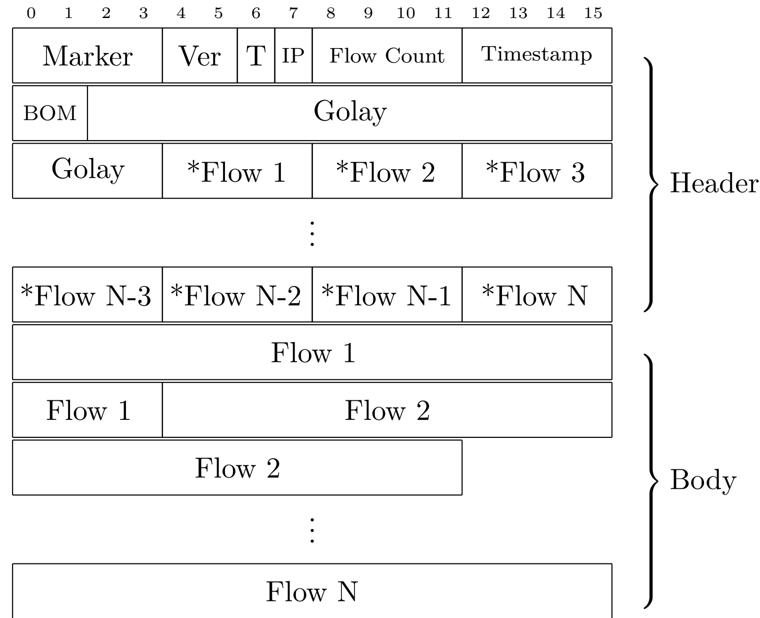


Figure 7.1: CompactFlow file format (in bytes).

five-tuple) and additional fields that contain volumetric and temporal information pertaining to communications. A five-tuple includes protocol, source IP address, source port, destination IP address and destination port. The basic additional fields of a flow are the timestamp of the first packet, the overall duration, the number of packets and the total size of packets.

In this section, we introduce CompactFlow, a new format specifically designed to provide more efficient storage and fast processing of network flow data. Our proposal relies on a new binary file format, which supports both unidirectional and bidirectional flows. In Figure 7.1, we show the general structure of a CompactFlow file. In what follows, we describe all of its components and discuss our design choices.

7.3.1 CompactFlow File Header

A CompactFlow file is structurally divided into the header and body. The header stores information about the format (i.e. binary file marker, format version, and byte order) and contained flows (i.e. type, IP version, number of flows, and timestamp) that are later encoded in the body of the file. The header of a CompactFlow file contains a *Marker* as a first field by which the format can be recognized. We

designed its value to be 0x00434600, where the inner bytes represent characters 'C' and 'F' and the outer bytes are non-character values to avoid being misinterpreted by applications for text files. The *Ver* value represents a version of the CompactFlow specification with the first byte being major and the second minor versions (e.g. 0.3). The T value stands for the type of flow in terms of its direction (unidirectional or bidirectional). Since CompactFlow supports both IPv4 and IPv6, the version of IP is given by the *IP* parameter. By design, IPv4 and IPv6 flows are stored in separate files. The *Flow Count* value stores the total number of flows contained in the file. Instead of storing complete timestamps in each flow, we only save the *Timestamp* (down to a precision of an hour) in the header since each file represents up to one hour of traffic. This allows for each flow record to store only the added time to that timestamp to reconstruct it fully. The *Byte Order Mark* (BOM) is used to clarify that the CompactFlow file uses big-endian encoding. This decision is justified by the fact that big-endian order is used by default in network communications, in fact to such an extent that it is often referred to as the network order.

All fields described above constitute mandatory data in the header. Since those fields carry high importance to the remainder of the file, we use the *extended binary Golay code* (G_{24}) to detect and correct errors in them in the case of corruption. Such code allows us to recover 3 bits for each 12-bit word at a cost of doubling the size of data. Fortunately, we can afford it since the header constitutes a minimal, almost negligible, percentage of the size of the whole file.

Our proposal uses dynamic field sizes to store flows, thus flow records can vary in size. This means that given a current flow it is not possible to know in advance where the next one starts. Typically this is not a problem in the context of network security analysis since the way to process flows is to sequentially traverse each one to get to the ones of interest [183], or to build more complex network behaviour profiles [12]. However, if CompactFlow is used in a network administration context, the types of workflows could require running the same queries to extract flows with a fixed set of parameters (e.g. given IP addresses). One could speed up such queries by indexing data of interest and then accessing it. This is described as *random*

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

$	@	s IP	d IP	time Δ	pkts #	bytes #	s prt	d prt	duration	pr	F
1	**2**	**4**	**4**	**3**	**1-4**	**1-4**	**0/2**	**0/2**	**0-3**	**0/1**	**0/1**

(a) Unidirectional flow.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40

$	@	s IP	d IP	time Δ	s pkts #	s bytes #	d pkts #	d bytes #	s prt	d prt	duration	pr	s F	d F
1	**3**	**4**	**4**	**3**	**1-4**	**1-4**	**0-4**	**0-4**	**0/2**	**0/2**	**0-3**	**0/1**	**0/1**	**0/1**

(b) Bidirectional flow.

$ = flow size field @ = control field s = source d = destination prt = port field pr = protocol field F = TCP Flags

Figure 7.2: Binary schema of CompactFlow records (in bytes).

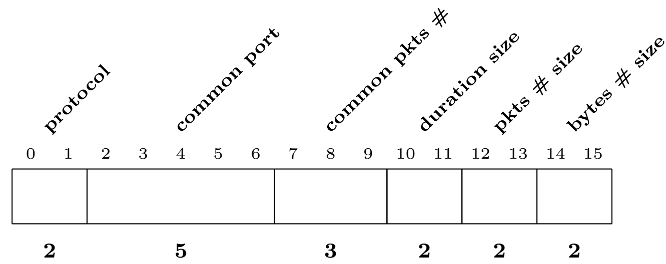
access and to enable this, CompactFlow precomputes an array of 4-byte pointers (offsets from the beginning of a file) to each flow record. Additionally, one can opt to use one of the indexing methods reported in Section 7.2.2.2. It is worth noting that this step is optional and such an array is not contained in the format by default. Overall, the header without an array of pointers takes 36 bytes of space.

7.3.2 Flow Binary Representation

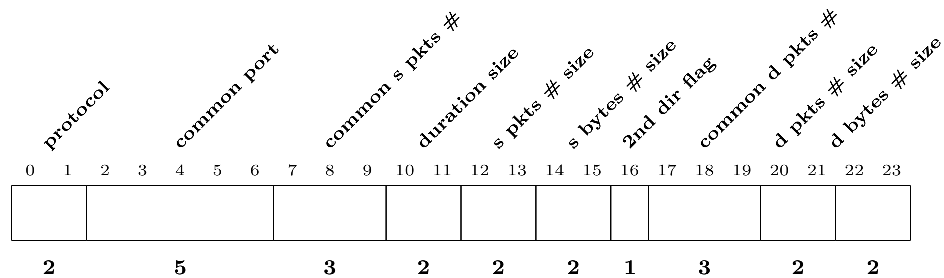
Flow records feature dynamic field sizes that are adjusted to the size of data that needs to be accommodated. The use of dynamic fields makes the flow more compact in cases where field values to be stored are small. As depicted in Figure 7.2a, every flow record contains the following fields: flow size (in bytes), control, source IP address, destination IP address, start time of the flow in terms of added milliseconds to the timestamp in the header, total number of packets, and total number of bytes. Optionally a flow can include a protocol, source and destination ports in the case of TCP or UDP protocols, duration and TCP flags. The length and position of variable-size fields is given by interpreting bits in the control field, described in Figure 7.3.

We noticed that some values are repeatedly used in flows. Saving the full values of such fields each time would require additional bytes per flow, which quickly build up if the number of flows is in the order of billions per day. For example, according to our observations the majority of traffic uses ICMP, TCP or UDP protocols. In order to save space, we use 2 bits (bits 0 to 1) of the control field, that allow for the storage of 4 values, to encode them with the fourth value meaning that another protocol is used which implies the existence of the protocol field in the binary data.

Port values (non-ephemeral) are encoded in a similar fashion using 5 bits (bits 2 to 6). They are only consulted if the protocol is either TCP or UDP (in other cases the port fields do not exist in the binary format). Encoded port values are specific to a given production network, hence they should be determined beforehand. Using those 5 bits, we can encode 32 values. The value of 0 means that neither source port nor destination port belong to the list of most frequently used ports. The value of 1



(a) Unidirectional control field



(b) Bidirectional control field

Figure 7.3: CompactFlow control fields (in bits).

is not used. The remaining 15 and 15 values mean that the source or destination port respectively is in the list of common ports. Each common port list hit saves 2 bytes of space. Since we observed that a significant number of flows constitute requests without a response which translate to a small number of packets, we use the next 3 bits (bits 7 to 9) to store small packet numbers. The value of 0 has a special meaning - the number of packets if different from a list of common small packet numbers. The remaining values from 1 to 7 are used to represent packet numbers, which results in 1 byte of saved space per flow. The next 2 bits (bits 10 to 11) in the control field store the length of the duration field. The value of 0 denotes that the duration is 0 and the respective duration field does not exist in the binary representation. The duration field can support non-standard values of up to 4.5 hours (with millisecond precision), even though the default active timeout value in the NetFlow 9 export protocol is only 30 minutes. This makes the format more robust towards changes of default values in flow exporters. The sizes of packets and bytes fields take 2 bits each (bits 12 to 13 and 14 to 15) to denote values up to 4 bytes. Summarizing, the size of a unidirectional flow can vary from 16 to

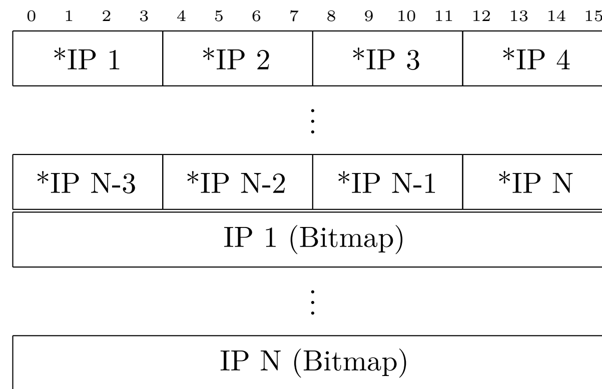


Figure 7.4: Bitmap index of IP addresses.

30 bytes. Each unidirectional CompactFlow file can store up to 143,165,576 flow records (unsigned integer using 4 bytes divided by a maximum size of a flow - 30), if one chooses to compute the array of pointers to each flow record.

CompactFlow also supports bidirectional flows (Figure 7.2b). The bidirectional format differs by the addition of packet and byte counters as well as TCP flags for the other side of the communication. It is not always the case that those counters exist, e.g. the communication might comprise only a request with no reply. In such scenarios, the destination counter values are not captured, hence the size of those counters can be equal to 0. For that reason, the size of a bidirectional flow is larger than its unidirectional counterpart and can take from 17 to 40 bytes.

To increase data processing performance, CompactFlow automatically places dynamic fields that could have the size of 0 at the end of the flow. If those fields were placed throughout the flow record, then one would need to query their size by calculating the corresponding flags in the control field in order to get to fields positioned after them. By using this design, we tried to minimize this behaviour. Additionally, IPv6 flow records are supported and saved into separate files. The format for such files differs in the number of bytes allocated for each address - 4 bytes for IPv4 and 16 bytes for IPv6.

The second part of the CompactFlow framework considers flow processing techniques. Our proposal supports a variety of indexing methods. We present how to create the bitmap index, which is a state-of-the-art approach. Bitmap indexing is efficient as it uses only 1 bit per flow record to denote whether a given value of a

flow field is present in a flow. Hence, each bitmap index is an array of bits with the size equal to the number of all flow records. Figure 7.4 reports the structure of such a bitmap index that is stored in a separate file as part of the CompactFlow format. Similarly, to the header of the unidirectional CompactFlow file, it stores the pointers to the locations of bitmap arrays for each field. In the case of this figure, these are IP addresses. In order to obtain selected flows, one needs to take the positions of bits with the value of 1 and jump to the respective flows by using the array of pointers in a binary CompactFlow file. Since the size of pointers is fixed-size (4 bytes), it is easy to jump to the correct ones with negligible overhead.

7.4 Evaluation

We compare our format to most popular open-source flow collectors that support binary file storage, i.e. Argus, flowd, nfdump, and SiLK. To carry out this comparison, we analyse the binary formats of such collectors to have full understanding of their flow representation. In Figure 7.5, we show the same unidirectional flow represented using the aforementioned binary formats and our proposed CompactFlow format. The flow is shown in the plaintext format (Figure 7.5a) with color-coded field names (explained in Figure 7.5b). In our comparison we configure flow collectors to store only the fields that we consider, whenever possible. Most binary formats use fixed-length representation of each flow record, which makes the file format more straightforward to read from. Indeed, it is possible to jump by a constant number of bytes to get to the same field in the next flow. However, this feature also makes it extremely inefficient space-wise. CompactFlow is designed to achieve a trade-off between file size and processing speed. In fact, our proposed format applies a hybrid approach, in which the always-present fields are of constant length and the fields whose values can change are of variable length. This results in a compact representation that is the smallest of all presented binary formats, as shown in Figure 7.5h. It is worth noting that the protocol, destination port and number of packets are included in the control field as an optimization by our binary format

02:59:40 8.96 TCP 192.168.1.43 58769 72.163.4.161 443 3 152

(a) Sample flow

timestamp duration proto srcIP srcPort destIP destPort packets bytes TCP flags

(b) Color legend

```
01:  3320 001d 0101 0102  c0a8 0101 0201 4105
02:  c0a8 012b 48a3 04a1  0600 e591 01bb 2020
03:  031a 1805 5c55 079c  0007 62a0 5c55 07a5
04:  0006 c660 1001 0602  0003 0098 3004 0003
05:  0000 0000 0000 0000  4800 0102 0000 0000
06:  4200 0005 0000 0000  0000 0000 0000 0000
07:  0000 0000 3200 0004  0000 0000 0000 0000
08:  0000 0000
```

(c) Argus

```
01:  600a 0000 0000 38a6  5c55 07ab 0002 d4a1
02:  1806 0000 c0a8 012b  48a3 04a1 e591 01bb
03:  0000 0000 0000 0003  0000 0000 0000 0098
```

(d) flowd

```
01:  0a00 3800 0600 0000  e401 bc01 9c07 555c
02:  a507 555c 0018 0600  91e5 bb01 0100 0000
03:  2b01 a8c0 a104 a348  0300 0000 0000 0000
04:  9800 0000 0000 0000
```

(e) nfdump

```
01:  da89 1003 2a80 2300  1840 0003 0000 0000
02:  e591 01bb c0a8 012b  48a3 04a1
```

(f) SiLK

```
01:  0000 1600 2c00 2b00  2400 2000 1400 1000
02:  0c00 0a00 0800 0400  1600 0000 0023 0000
03:  bb01 91e5 9800 0000  0300 0000 44bb 25ac
04:  6801 0000 0000 0000  a104 a348 2b01 a8c0
```

(g) FlatBuffers

```
01:  1862 e1c0 a801 2b48  a304 a136 a244 9891
02:  e523 0018
```

(h) CompactFlow

Figure 7.5: Comparison of binary file formats.

(see Section 7.3.2). The sample flow in CompactFlow binary format is only 20 bytes. In what follows, we discuss and compare different flow collectors one by one.

Audit Record Generation and Utilization System (Argus) is a popular, open-source flow monitoring framework. The Argus collector provides a binary file format to store flow records, which assigns a fixed-length space for fields constituting a flow. It is worth noting that such length remains fixed even when not all fields are captured. This approach leads to wasted space, which is a fundamental factor when dealing with large data sets. In total a flow record takes 116 bytes.

Another flow collector, flowd, offers binary storage at a reduced size of 48 bytes per flow record. It uses big-endian encoding and provides no file header. It offers an option to save protocol, TCP flags, and Type of Service without being able to selectively pick each one. Additionally, there is no option to save the timestamp from when the flow started, only the timestamp of receiving the flow by the collector. It also does not provide an option to store the duration of a flow. Both of these shortcomings limit its use in real-world settings.

A slight improvement in those regards is offered by nfdump. It uses the nfcapd tool to collect flows from the exporter and to save them to binary files. It also allows for fine-grained specification of which fields to store. However, it keeps start and end timestamp from which duration field is calculated, which is not an efficient approach. It also uses little-endian encoding, which might seem strange since network order is big-endian. Moreover, the conversion between encodings might add unnecessary overhead to the collection process. The binary format of nfdump stores the header in 344 bytes, each flow record in 56 bytes and the footer in 44 bytes.

SiLK is the most optimized open-source flow collector with a state-of-the-art binary format and a set of processing tools. It provides an option to specify endianness of files and provides optimizations such as storing flow duration instead of end time or storing average amount of bytes transferred per packet. This results in the smallest binary format of all open-source tools with 24-88 bytes for the header and 28 bytes per flow record. However, it does not support bidirectional flows (even though it provides tools to match flows after they are captured) and it does

not provide functionality to index flows. Moreover, it relies on the file hierarchy, which divides traffic into internal to internal, external to external, incoming and outgoing Web and ICMP traffic. While the hierarchy can speed up select queries regarding one of those types, it also makes the binary format more complex and adds overhead for more analytic queries, such as the selection of all flows within a time period to train machine learning models.

Additionally to flow collectors, we examine a popular, fast data serialization library, FlatBuffers [209]. This library supports a large variety of extensions in different programming languages to interact with its binary data format. One of its advantages is the flexibility of what information to store. This is done by writing a schema with a structure of data to be stored. In the case of storing flow records, FlatBuffers does not perform better than most of flow collectors. It uses 64 bytes per flow record. This is due to the support of only regular types (e.g. uint8, uint16, uint32, uint64) and no custom types (such as uint24) which leads to wasted space. Additionally, in order to store a collection of records, one needs to specify another type that will serve as a container for those records.

7.4.1 Average Flow Size Experiment

Even though the comparison favors the CompactFlow format, a single flow is not representative of a larger variety of flows on networks. In fact, the unidirectional record takes from 16 to 30 bytes, which in some cases can exceed SiLK's 28 bytes per flow record. For this reason, we evaluate the average size of a CompactFlow record by considering a variety of flows from the University of Oxford. Experimental results based on the analysis of 1,802,377,030 flows show that the average size of a flow in our proposed unidirectional binary format is 21.4 bytes - a value 24% smaller than SiLK's format.

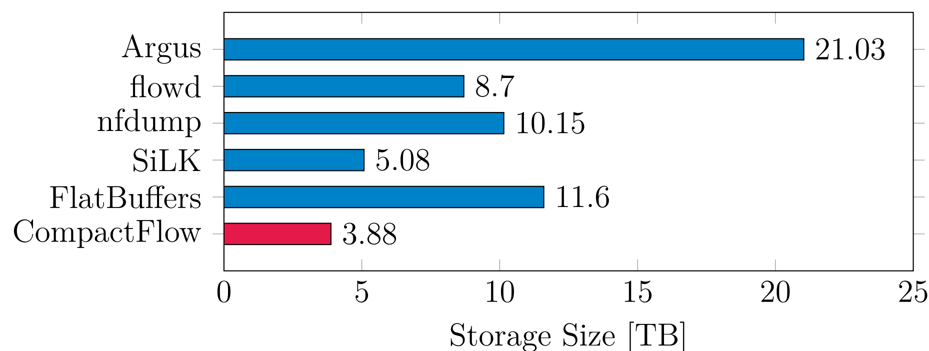


Figure 7.6: 181 million flows from the University of Oxford stored using different flow collectors.

7.4.2 Big Data Flow Storage

To get a clearer view of what different binary formats mean in the real world, we used each of them to store over five months of flow data from the University of Oxford. The results are shown in Figure 7.6. In total 181,315,995,252 flows are stored that come from three networks with over 64 thousand hosts. The most inefficient format takes over 21 TB to store the entirety of this data. While the state-of-the-art binary format of SiLK uses about 5 TB, our proposed format uses only 3.88 TB.

7.4.3 Port Analysis

The list of the most common ports that are encoded in the control field of a flow record (Figure 7.3) is created from an analysis of the most used ports in one month of the University of Oxford network traffic (shown in Figure 7.7). The ports in the list represent university's traffic and can differ in other real-world settings. For this reason, they should be treated as a suggestion and in other use cases a different list of ports can be used.

7.4.4 CompactFlow Compression Experiment

The final aspect of the evaluation is a comparison of an uncompressed CompactFlow file against three common compression algorithms (i.e. gzip, bzip2, and lzo). Such

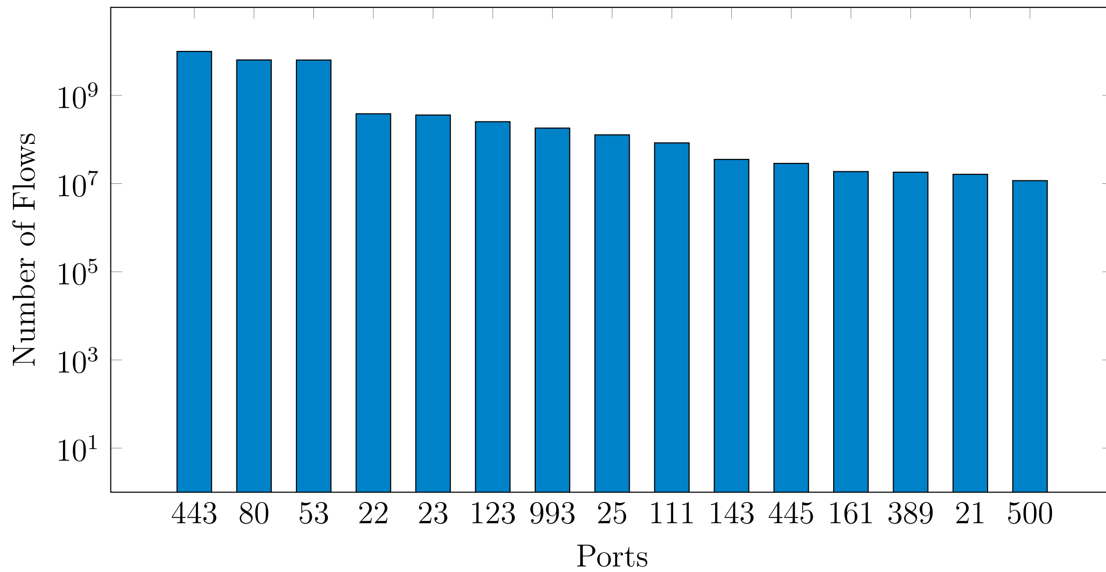


Figure 7.7: Most commonly used ports in one month of the University of Oxford dataset.

Compression Method	CPU Cores	File Size (MB)	Size Gain (%)	RAM Load Time (s)	Processing Time (s)	Total Time (s)	Time Loss (%)
uncompressed		1,425	-	3.267	0.548	3.815	-
gzip	1	791	44	1.887	12.065	13.952	366
gzip	32	792	44	1.881	7.408	9.289	243
bzip2	1	697	51	1.609	61.638	63.247	1,658
bzip2	32	696	51	1.603	3.155	4.758	125
lzo	1	1,023	28	2.325	4.818	7.143	187

Table 7.2: Comparison of compression methods used with a CompactFlow file of 67,257,407 unidirectional IPv4 flow records.

an analysis is meant to show which approach is the fastest in terms of loading the file into memory (RAM load time) and reading contained flow records (processing time). We observed that those two metrics are a trade-off between disk speed and a chosen compression algorithm. As it was shown in 2014 in the evaluation of SiLK, reading flow records was faster from a compressed binary file. This was due to the limited speed of then widely used hard disk drives (HDDs). However, we assess that this is no longer true with the rising popularity and decreasing prices of solid-state drives (SSDs). In Table 7.2, we show that the increase of disk speeds

(from 100 MB/s in [183] to 436MB/s in our analysis) results in faster processing of raw, uncompressed binary files. In order to reverse this trend, one can assign more CPU cores to speed up the decompression. We used 32 cores of a dual Intel Xeon CPU to determine that it takes 25% longer in total while reducing the file size by half. However, the prices and power requirements of such CPUs are high, which means that often they are not available to network administrators. As a result, compression is not suitable in the case of commodity hardware, which puts more emphasis on a small binary representation of flow data.

7.5 Discussion

Storing data in a binary format is more efficient than database-based methods in terms of size. In fact, the database-based methods need more space for indexing purposes, which may even take double the space required for the data [56]. They also do not allow a fine-grained control of what and when is indexed which accounts for their poor per-flow storage times. Our format provides a quicker and hybrid solution. It is also robust in case of errors. We use Golay code in the header to preserve fundamental information regarding the flows in the file, such as an hour-based timestamp, the IP version supported, or the type of flow. Secondly, the header can be optionally enriched with an array of flow pointers. In this way, it would be easy to isolate the faulty flows in case errors occur within the file body. Faulty flows can be easily detected by relying on the first two flow fields, namely *flow size* and *flow control*. As a first check, we have to verify the following condition on the *flow size* value:

$$flow_size \geq \sum_{i \in C} size(i) + 2$$

where C is a set of flow fields of constant size and $size(i)$ is the size of field i. The additional 2 is related to the two variable fields with a minimum size of 1 byte (i.e. number of packets and bytes). The condition does not comprise the other variable fields since their minimum size is 0. A second check on flow consistency could be

made on the packet and byte flow fields. Indeed, it is known that not only is the former smaller than the latter, but that the following condition is verified:

$$bytes\# \geq min_packet_size * packets\#$$

where min_packet_size is the minimum allowed packet size by the considered protocol.

Our evaluation shows that even though flow collectors use the most efficient type of data storage, binary files, they usually do so in an inefficient manner. In fact, a string representation of each flow record would take less space than in most evaluated binary formats.

CompactFlow is intentionally examined with regards to its storage requirements, while not addressing fully the evaluation of processing speed when querying data. This is due to a number of reasons. The implementation presented in this chapter is a proof of concept while binary formats of other flow collectors are open-source and widely used for a long period of time, which results in an optimized implementation tested and improved by many parties. Testing the current version of CompactFlow (which is implemented in C#) against related work (where C is the dominating implementation language) would not result in an objective comparison. Secondly, since CompactFlow is designed to support many indexing algorithms as well as a number of indexing settings (i.e. indexing only selected flow fields) it is challenging to find suitable counterparts in the related work. Usually, other binary formats either do not support indexing methods or support only a single chosen one. Moreover, while testing our binary format the results of internal processing speed evaluation, which analysed how much time it takes to process a fixed number of flows, indicated that in the worst case scenario it can take 10% more time than the most efficient framework. However, this evaluation did not take into account the speedup one could obtain by using indexing. We acknowledge that those results are within the margin of our design expectations, which mainly targeted storage performance, while also noting the available and flexible indexing support if additional processing efficiency is required.

We do not evaluate compressed sizes of different flow collectors' binary formats. Even if compressed sizes were similar, in order to process the data, one needs to decompress it - which brings us to the initial problem since the file sizes start to matter again. We show in Section 7.4 that compression slows down the processing of flows. Moreover, memory prices show no signs of decreasing, hence it is important for a format to have a minimal memory footprint.

7.6 Summary

In this chapter, we presented a hybrid binary file format to store network flow data. It not only is compact in its representation, but also supports well-known indexing approaches to speed up flow queries. To assess the performance of the CompactFlow format, we compared it to the most popular open-source flow collectors with an in-depth analysis of their binary formats. Then, we carried out an extensive comparison in terms of storage size on a real-world traffic dataset from the University of Oxford. Finally, we evaluated the impact of compression on our format in terms of file size and processing time.

8
Conclusion

Contents

8.1 Summary of This Thesis 167
8.2 Future Work 169
8.2.1 Flow Data Enhancement 170
8.2.2 Behaviour-Based Malware Taxonomy 170
8.3 Final Remarks 171

8.1 Summary of This Thesis

This thesis aims to explore the use of behaviours extracted from network traffic in order to profile network hosts. The goal of the profiling may range from identification of network hosts by uniquely fingerprinting them or it could concentrate on a subset of network traffic of a host in order to detect malicious communications. The analysis of network packets is not feasible nowadays due to the number of network-connected devices and the widely-used encryption. For this reason, the work presented in this thesis uses network flows that contain a subset of packet data, which allows the collection of summaries of all communications on a network in most real-world cases.

In Chapter 4, we show that network flow data possesses enough information to extract unique behaviours of network hosts which consequently leads to the

ability of identifying them. In the experiments, we show that Hostbuster is able to distinguish hosts on networks of up to 200 hosts with the F1 score of 77%. Additionally, the fingerprints, which contain host behaviours, are able to identify them even after three months of model training. While Hostbuster may be used to guarantee the security of networks by monitoring whether host behaviours deviate from their regular behaviours (i.e. anomaly detection), or to authenticate hosts from a biometric perspective, it can be also used by attackers. We show that a knowledgeable adversary may use Hostbuster to defy NMTD systems, which results in us proposing guidelines to strengthen deployments of such systems.

Chapter 5, adapts the method used in Hostbuster to malware detection and classification problem, which shows that it is possible not only to identify malware from network flow-level traffic, but also categorize it into malware types. The results show that MalAlert can do so with the F1 score of up to 94%. Furthermore, the evaluation on the University of Oxford dataset, containing over 23 billion flows, shows that MalAlert detects a number of malicious fingerprints from this real-world, production network, which is especially important for the confidence threshold 1.0, while other thresholds require consultation of additional sources.

The system proposed in Chapter 6 tackles a wider variety of challenges than MalAlert. It aims to show that it is possible to achieve: more fine-grained malware classification (i.e. families), malware detection from mixed network traffic that conforms to the one on real-world networks, and identify network traffic generated by unseen malware. In the case where an attack compromised the production systems which makes them unable to operate correctly, it is important to gain further insight into this threat. In order to recover from the attack, it is necessary to know the type of malware that was used to carry out the attack. Once it is established, one may discriminate the malware further into a specific family. For example, when it is known that the perpetrator used a malware sample that is part of a ransomware family, detection of the specific ransomware family might help with recovering the data (i.e. some strains of ransomware implement their own version of cryptographic algorithms which are known to be breakable), or help

with proactively deciding which parts of the network to isolate. While the goals of MalAlert are more limited, it is based on a random forest algorithm and therefore it is possible to justify its decisions. MalPhase is based on deep learning which offers better classification performance, but its models operate in a black-box fashion. It is worth noting that some critical national infrastructure employ only methods that are based on explainable machine learning methods. We were not able to apply MalPhase to the University of Oxford data, because of the requirement of additional flow data, such as packet payload entropy. Unfortunately, the dataset from Oxford contains only flow data according to the minimal definition (i.e. 5-tuple and packet counters). This poses a minor challenge for methods that work with enhanced network flow data. Enterprises would need to adopt additional network traffic data into the captured flows. However, most standards that are widely used offer this functionality (e.g. NetFlow v9), and the storage of the additional features has minimal impact on the space taken.

In Chapter 7, we propose a hybrid binary format to store network flows. CompactFlow is more efficient than comparable proposals from the literature. It's binary representation takes 24% less space than the state-of-the-art format. We store all network flows from the University of Oxford dataset using CompactFlow binary format, which we take advantage of to make our other proposals in this thesis possible.

8.2 Future Work

Throughout this thesis, we proposed a number of methods concerned with extraction of behaviours from network flow data in order to detect and classify actors of interest or other suspicious activities. They ranged from profiling network hosts to identification of malware. Each content chapter was accompanied by the discussion of the results and critical analysis of the proposed method. In this section, we talk about how the faced challenges could be attempted to be overcome as part of future work.

8.2.1 Flow Data Enhancement

In this thesis, we work only with network flows which we utilize in the presented methods. Due to the fact that network flows hold compact representation of the data, they can be captured even on large-scale networks. The majority of available network data is captured according to the classical definition of a flow (5-tuple and packet counters). The addition of supplementary data to a flow, that would be extracted from network packets, is an interesting proposal in the context of malware detection and behaviour profiling in general. With a small increase of the space taken by a flow, one could augment their informativeness with the aim of extracting more unique behaviour-related data.

In Chapter 6 we experiment with this notion of extended network flow data by adding, among others, packet payload entropy. While we do not include explicitly the experiment that evaluates the classification performance of MalPhase on standard flow data in comparison to extended flow data, we observed during the creation of this method that the inclusion of packet payload entropy alone boosts the classifiers' performance. Our experimentation with additional flow data could be viewed as quite rudimentary, as this was not the aim of our methods (mainly due to the inability to use real-world network flows, which are most often captured according to the classical definition of a flow). However, it could pose an interesting avenue of future work.

8.2.2 Behaviour-Based Malware Taxonomy

Modern malware uses increasingly sophisticated approaches to exploit its targets. Each malware campaign has a set of goals that shape the design of a malicious program. Some of those goals are shared which results in commonalities among malware, which researchers use to create a logical division of malware. The criteria for such a division may differ. For example, one could create a malware taxonomy based on their design or type of operation (i.e. targeted or widespread).

In Section 6.7.1, we observe that the difficulty with providing accurate malware taxonomies and with them being accepted by the community further exacerbates the challenge-ridden landscape of malware detection, which is especially true in the network-based context. The prime example of this is inconsistent use of the terminology in malware taxonomies (e.g. the frequent interchangeable use of "malware type" and "malware family"). Additionally, the lack of a well-known taxonomy with regards to the behaviour of malware has posed problems for us when defining malware types. In Chapter 5 we include botnet as one of malware types, while in Chapter 6 we deliberately exclude it observing that it only describes the way of operation of malicious programs and not their intention. Only the latter generate network behaviours that we are interested in.

Having mentioned the problems with current malware taxonomies, we think that a deeper analysis of malware, taking into account a variety of criteria, would lead to better taxonomies that could have a direct impact on a number of malware research themes. In our case, we identified the behaviour-based malware taxonomy to be lacking, which we find important to be tackled as future work.

8.3 Final Remarks

Historically, malware detection has been a well-studied topic that received interest from a number of researchers. Previous approaches tackle this problem using different types of data. Host-based methods are concerned with analyzing malware on a given machine, usually by inspecting a malware executable to examine its structure or extract behaviours in form of system calls. While this results in fine-grained malware data, it does not scale — it is prone to manual reverse engineering efforts and has to be done on each host. Network-based methods, which gained popularity with the development of the malware detection field, use a different type of data. Initially they captured network packets and gradually, with the increase of network traffic, methods started adapting the network flow data format. While such

data contains less information, it scales well and can be captured on centralised nodes comprising of a large number of hosts.

In this thesis, we have shown novel methods that work with network flow data to capture underlying behaviours. While our results have shown that we are able to successfully profile behaviour of hosts and malware, we had to face a number of problems that have been known in this field. As described previously in the literature, they include practices for handling of malware datasets [170], challenges in malware detection [128], and challenges of using machine learning for intrusion detection [147]. Having those challenges in mind, we would like to highlight that the field of behaviour profiling based on network traffic is a fuzzy problem. We hope that the novelty of the methods presented in this thesis helps in pushing the field forward and will serve as a foundation for future, improved techniques.

References

[1] Monowar H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita. “Network Anomaly Detection: Methods, Systems and Tools”. In: *IEEE Communications Surveys Tutorials* 16.1 (2014). Conference Name: IEEE Communications Surveys Tutorials, pp. 303–336.

[2] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. “BLINC: multilevel traffic classification in the dark”. In: *ACM SIGCOMM Computer Communication Review* 35.4 (Aug. 2005), pp. 229–240. URL: https://doi.org/10.1145/1090191.1080119 (visited on 08/28/2020).

[3] S. Zander, T. Nguyen, and G. Armitage. “Automated traffic classification and application identification using machine learning”. In: *The IEEE Conference on Local Computer Networks 30th Anniversary (LCN'05)l*. ISSN: 0742-1303. Nov. 2005, pp. 250–257.

[4] Vincent F. Taylor et al. “Robust Smartphone App Identification via Encrypted Network Traffic Analysis”. In: *IEEE Transactions on Information Forensics and Security* 13.1 (Jan. 2018). Conference Name: IEEE Transactions on Information Forensics and Security, pp. 63–78.

[5] Mauro Conti et al. “Analyzing Android Encrypted Network Traffic to Identify User Actions”. In: *IEEE Transactions on Information Forensics and Security* 11.1 (Jan. 2016). Conference Name: IEEE Transactions on Information Forensics and Security, pp. 114–125.

[6] B. Claise. *RFC3954 - Cisco Systems NetFlow Services Export Version 9*. Tech. rep. Internet Engineering Task Force (IETF), 2004. URL: https://tools.ietf.org/html/rfc3954.

[7] B. Claise, B. Trammell, and P. Aitken. *RFC7011 - Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information*. Tech. rep. Internet Engineering Task Force (IETF), 2013. URL: https://tools.ietf.org/html/rfc7011.

[8] Bruce Schneier. *Secrets and Lies: Digital Security in a Networked World*. Wiley, 2000.

[9] Xabier Ugarte-Pedrero, Mariano Graziano, and Davide Balzarotti. “A Close Look at a Daily Dataset of Malware Samples”. In: *ACM Transactions on Privacy and Security* 22.1 (Jan. 2019), 6:1–6:30. URL: https://doi.org/10.1145/3291061 (visited on 08/28/2020).

[10] Anna L. Buczak and Erhan Guven. “A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection”. In: *IEEE Communications Surveys Tutorials* 18.2 (2016). Conference Name: IEEE Communications Surveys Tutorials, pp. 1153–1176.

[11] Michal Piskozub et al. “On the Resilience of Network-based Moving Target Defense Techniques Against Host Profiling Attacks”. In: *Proceedings of the 6th ACM Workshop on Moving Target Defense*. MTD’19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 1–12. URL: https://doi.org/10.1145/3338468.3356825 (visited on 09/17/2020).

[12] Michal Piskozub, Riccardo Spolaor, and Ivan Martinovic. “MalAlert: Detecting Malware in Large-Scale Network Traffic Using Statistical Features”. In: *ACM SIGMETRICS Performance Evaluation Review* 46.3 (Jan. 2019), pp. 151–154. URL: https://doi.org/10.1145/3308897.3308961 (visited on 08/28/2020).

[13] Michal Piskozub et al. “MalPhase: Fine-Grained Malware Detection Using Network Flow Data”. In: ASIA CCS ’21. Hong Kong: Association for Computing Machinery, 2021. URL: https://doi.org/10.1145/3433210.3453101.

[14] Michal Piskozub, Riccardo Spolaor, and Ivan Martinovic. “CompactFlow: A Hybrid Binary Format for Network Flow Data”. en. In: *Information Security Theory and Practice*. Ed. by Maryline Laurent and Thanassis Giannetsos. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 185–201.

[15] *VirusTotal*. https://www.virustotal.com/gui/.

[16] D.E. Denning. “An Intrusion-Detection Model”. In: *IEEE Transactions on Software Engineering* SE-13.2 (Feb. 1987). Conference Name: IEEE Transactions on Software Engineering, pp. 222–232.

[17] N. Brownlee, C. Mills, and G. Ruth. *RFC2722 - Traffic Flow Measurement: Architecture*. Tech. rep. Internet Engineering Task Force (IETF), 1999. URL: https://tools.ietf.org/html/rfc2722.

[18] J. Rajahalme et al. *RFC3697 - IPv6 Flow Label Specification*. Tech. rep. Internet Engineering Task Force (IETF), 2004. URL: https://tools.ietf.org/html/rfc3697.

[19] Ralph Koning et al. “CoreFlow: Enriching Bro security events using network traffic monitoring data”. en. In: *Future Generation Computer Systems* 79 (Feb. 2018), pp. 235–242. URL: http://www.sciencedirect.com/science/article/pii/S0167739X17305952 (visited on 08/28/2020).

[20] Bingdong Li et al. “A survey of network flow applications”. en. In: *Journal of Network and Computer Applications* 36.2 (Mar. 2013), pp. 567–581. URL: http://www.sciencedirect.com/science/article/pii/S1084804512002676 (visited on 08/28/2020).

[21] N. Hoque et al. “Network attacks: Taxonomy, tools and systems”. en. In: *Journal of Network and Computer Applications* 40 (Apr. 2014), pp. 307–324. URL: http://www.sciencedirect.com/science/article/pii/S1084804513001756 (visited on 08/28/2020).

[22] Anna Sperotto et al. “An Overview of IP Flow-Based Intrusion Detection”. In: *IEEE Communications Surveys Tutorials* 12.3 (2010). Conference Name: IEEE Communications Surveys Tutorials, pp. 343–356.

[23] Julina Zhang et al. "Comparing unsupervised learning approaches to detect network intrusion using NetFlow data". In: *2017 Systems and Information Engineering Design Symposium (SIEDS)*. Apr. 2017, pp. 122–127.

[24] Ignasi Paredes-Oliva, Pere Barlet-Ros, and Josep Solé-Pareta. "Portscan Detection with Sampled NetFlow". en. In: *Traffic Monitoring and Analysis*. Ed. by Maria Papadopouli, Philippe Owezarski, and Aiko Pras. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 26–33.

[25] Rick Hofstede et al. "SSH Compromise Detection using NetFlow/IPFIX". In: *ACM SIGCOMM Computer Communication Review* 44.5 (Oct. 2014), pp. 20–26. URL: `https://doi.org/10.1145/2677046.2677050` (visited on 08/28/2020).

[26] Maryam M. Najafabadi et al. "Detection of SSH Brute Force Attacks Using Aggregated Netflow Data". In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. Dec. 2015, pp. 283–288.

[27] Jaehak Yu et al. "An in-depth analysis on traffic flooding attacks detection and system using data mining techniques". en. In: *Journal of Systems Architecture*. Advanced Smart Vehicular Communication System and Applications 59.10, Part B (Nov. 2013), pp. 1005–1012. URL: `http://www.sciencedirect.com/science/article/pii/S1383762113001562` (visited on 08/28/2020).

[28] Alan Saied, Richard E. Overill, and Tomasz Radzik. "Detection of known and unknown DDoS attacks using Artificial Neural Networks". en. In: *Neurocomputing* 172 (Jan. 2016), pp. 385–393. URL: `http://www.sciencedirect.com/science/article/pii/S092523121501053X` (visited on 08/28/2020).

[29] Johannes Krupp, Michael Backes, and Christian Rossow. "Identifying the Scan and Attack Infrastructures Behind Amplification DDoS Attacks". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 1426–1437. URL: `https://doi.org/10.1145/2976749.2978293` (visited on 08/28/2020).

[30] Xiao Han, Nizar Kheir, and Davide Balzarotti. "PhishEye: Live Monitoring of Sandboxed Phishing Kits". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 1402–1413. URL: `https://doi.org/10.1145/2976749.2978330` (visited on 08/28/2020).

[31] Ting-Fang Yen and Michael K. Reiter. "Traffic Aggregation for Malware Detection". en. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Diego Zamboni. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 207–227.

[32] Karel Bartos, Michal Sofka, and Vojtech Franc. "Optimized invariant representation of network traffic for detecting unseen malware variants". In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC'16. USA: USENIX Association, Aug. 2016, pp. 807–822. (Visited on 08/28/2020).

[33] Christian Rossow et al. “Sandnet: network traffic analysis of malicious software”. In: *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*. BADGERS ’11. New York, NY, USA: Association for Computing Machinery, Apr. 2011, pp. 78–88. URL: https://doi.org/10.1145/1978672.1978682 (visited on 08/28/2020).

[34] André Ricardo Abed Grégio et al. “Toward a Taxonomy of Malware Behaviors”. en. In: *The Computer Journal* 58.10 (Oct. 2015). Publisher: Oxford Academic, pp. 2758–2777. URL: https://academic.oup.com/comjnl/article/58/10/2758/454747 (visited on 08/28/2020).

[35] Z. Berkay Celik et al. “Malware traffic detection using tamper resistant features”. In: *MILCOM 2015 - 2015 IEEE Military Communications Conference*. Oct. 2015, pp. 330–335.

[36] Guofei Gu et al. “BotHunter: detecting malware infection through IDS-driven dialog correlation”. In: *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. SS’07. USA: USENIX Association, Aug. 2007, pp. 1–16. (Visited on 08/28/2020).

[37] G. Gu, J. Zhang, and W. Lee. “BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic”. In: *NDSS*. 2008.

[38] Guofei Gu et al. “BotMiner: clustering analysis of network traffic for protocol- and structure-independent botnet detection”. In: *Proceedings of the 17th conference on Security symposium*. SS’08. USA: USENIX Association, July 2008, pp. 139–154. (Visited on 08/28/2020).

[39] Jérôme François et al. “BotTrack: Tracking Botnets Using NetFlow and PageRank”. en. In: *NETWORKING 2011*. Ed. by Jordi Domingo-Pascual et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 1–14.

[40] Nizar Kheir and Chirine Wolley. “BotSuer: Suing Stealthy P2P Bots in Network Traffic through Netflow Analysis”. en. In: *Cryptology and Network Security*. Ed. by Michel Abdalla, Cristina Nita-Rotaru, and Ricardo Dahab. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2013, pp. 162–178.

[41] Leyla Bilge et al. “Disclosure: detecting botnet command and control servers through large-scale NetFlow analysis”. In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC ’12. New York, NY, USA: Association for Computing Machinery, Dec. 2012, pp. 129–138. URL: https://doi.org/10.1145/2420950.2420969 (visited on 08/28/2020).

[42] Manos Antonakakis et al. “From throw-away traffic to bots: detecting the rise of DGA-based malware”. In: *Proceedings of the 21st USENIX conference on Security symposium*. Security’12. USA: USENIX Association, Aug. 2012, p. 24. (Visited on 08/28/2020).

[43] Sandeep Yadav et al. “Detecting Algorithmically Generated Domain-Flux Attacks With DNS Traffic Analysis”. In: *IEEE/ACM Transactions on Networking* 20.5 (Oct. 2012). Conference Name: IEEE/ACM Transactions on Networking, pp. 1663–1677.

[44] Martin Grill et al. “Detecting DGA malware using NetFlow”. In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. ISSN: 1573-0077. May 2015, pp. 1304–1309.

[45] Aleksandar Lazarevic et al. “A Comparative Study of Anomaly Detection Schemes in Network Intrusion Detection”. In: *SDM*. 2003.

[46] Martin Rehák et al. “Trust-Based Classifier Combination for Network Anomaly Detection”. en. In: *Cooperative Information Agents XII*. Ed. by Matthias Klusch, Michal Pěchouček, and Axel Polleres. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 116–130.

[47] Valentín Carela-Español et al. “Analysis of the impact of sampling on NetFlow traffic classification”. In: *Computer Networks: The International Journal of Computer and Telecommunications Networking* 55.5 (Apr. 2011), pp. 1083–1099. URL: `https://doi.org/10.1016/j.comnet.2010.11.002` (visited on 08/28/2020).

[48] Thuy T.T. Nguyen and Grenville Armitage. “A survey of techniques for internet traffic classification using machine learning”. In: *IEEE Communications Surveys Tutorials* 10.4 (2008). Conference Name: IEEE Communications Surveys Tutorials, pp. 56–76.

[49] Enrico Mariconti et al. “What’s Your Major Threat? On the Differences between the Network Behavior of Targeted and Commodity Malware”. In: *2016 11th International Conference on Availability, Reliability and Security (ARES)*. Aug. 2016, pp. 599–608.

[50] Farzaneh Pakzad et al. “Efficient topology discovery in software defined networks”. In: *2014 8th International Conference on Signal Processing and Communication Systems (ICSPCS)*. Dec. 2014, pp. 1–8.

[51] Abdelhadi Azzouni et al. “sOFTDP: Secure and efficient OpenFlow topology discovery protocol”. In: *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. ISSN: 2374-9709. Apr. 2018, pp. 1–7.

[52] Nino Vincenzo Verde et al. “No NAT’d User Left Behind: Fingerprinting Users behind NAT from NetFlow Records Alone”. In: *2014 IEEE 34th International Conference on Distributed Computing Systems*. ISSN: 1063-6927. June 2014, pp. 218–227.

[53] Cristian Estan et al. “Building a better NetFlow”. In: *ACM SIGCOMM Computer Communication Review* 34.4 (Aug. 2004), pp. 245–256. URL: `https://doi.org/10.1145/1030194.1015495` (visited on 08/28/2020).

[54] Karel Bartos and Martin Rehak. “IFS: Intelligent flow sampling for network security-an adaptive approach”. In: *Networks* 25.5 (Sept. 2015), pp. 263–282.

[55] Daniela Brauckhoff et al. “Impact of packet sampling on anomaly detection metrics”. In: *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. IMC ’06. New York, NY, USA: Association for Computing Machinery, Oct. 2006, pp. 159–164. URL: `https://doi.org/10.1145/1177080.1177101` (visited on 08/28/2020).

[56] Rick Hofstede et al. “Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX”. In: *IEEE Communications Surveys Tutorials* 16.4 (2014). Conference Name: IEEE Communications Surveys Tutorials, pp. 2037–2064.

[57] Yeonhee Lee and Youngseok Lee. “Toward scalable internet traffic measurement and analysis with Hadoop”. In: *ACM SIGCOMM Computer Communication Review* 43.1 (Jan. 2012), pp. 5–13. URL: https://doi.org/10.1145/2427036.2427038 (visited on 08/28/2020).

[58] Samuel Marchal et al. “A Big Data Architecture for Large Scale Security Monitoring”. In: *2014 IEEE International Congress on Big Data*. ISSN: 2379-7703. June 2014, pp. 56–63.

[59] Jörg Wallerich et al. “A methodology for studying persistency aspects of internet flows”. In: *ACM SIGCOMM Computer Communication Review* 35.2 (Apr. 2005), pp. 23–36. URL: https://doi.org/10.1145/1064413.1064417 (visited on 08/28/2020).

[60] Zaoxing Liu et al. “One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon”. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM ’16. New York, NY, USA: Association for Computing Machinery, Aug. 2016, pp. 101–114. URL: https://doi.org/10.1145/2934872.2934906 (visited on 08/28/2020).

[61] Xiaoxin Yin et al. “VisFlowConnect: netflow visualizations of link relationships for security situational awareness”. In: *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*. VizSEC/DMSEC ’04. New York, NY, USA: Association for Computing Machinery, Oct. 2004, pp. 26–34. URL: https://doi.org/10.1145/1029208.1029214 (visited on 08/28/2020).

[62] B. Irwin and N. Pilkington. “High Level Internet Scale Traffic Visualization Using Hilbert Curve Mapping”. en. In: *VizSEC 2007: Proceedings of the Workshop on Visualization for Computer Security*. Ed. by John R. Goodall, Gregory Conti, and Kwan-Liu Ma. Mathematics and Visualization. Berlin, Heidelberg: Springer, 2008, pp. 147–158. URL: https://doi.org/10.1007/978-3-540-78243-8_10 (visited on 08/28/2020).

[63] J. R. Goodall. “Introduction to Visualization for Computer Security”. en. In: *VizSEC 2007: Proceedings of the Workshop on Visualization for Computer Security*. Ed. by John R. Goodall, Gregory Conti, and Kwan-Liu Ma. Mathematics and Visualization. Berlin, Heidelberg: Springer, 2008, pp. 1–17. URL: https://doi.org/10.1007/978-3-540-78243-8_1 (visited on 08/28/2020).

[64] T. Taylor, S. Brooks, and J. McHugh. “NetBytes Viewer: An Entity-Based NetFlow Visualization Utility for Identifying Intrusive Behavior”. en. In: *VizSEC 2007: Proceedings of the Workshop on Visualization for Computer Security*. Ed. by John R. Goodall, Gregory Conti, and Kwan-Liu Ma. Mathematics and Visualization. Berlin, Heidelberg: Springer, 2008, pp. 101–114. URL: https://doi.org/10.1007/978-3-540-78243-8_7 (visited on 08/28/2020).

[65] F. Mansman, L. Meier, and D. A. Keim. “Visualization of Host Behavior for Network Security”. en. In: *VizSEC 2007: Proceedings of the Workshop on Visualization for Computer Security.* Ed. by John R. Goodall, Gregory Conti, and Kwan-Liu Ma. Mathematics and Visualization. Berlin, Heidelberg: Springer, 2008, pp. 187–202. URL: https://doi.org/10.1007/978-3-540-78243-8_13 (visited on 08/28/2020).

[66] Fabian Fischer et al. “Large-Scale Network Monitoring for Visual Analysis of Attacks”. en. In: *Visualization for Computer Security.* Ed. by John R. Goodall, Gregory Conti, and Kwan-Liu Ma. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 111–118.

[67] Florian Stoffel, Fabian Fischer, and Daniel A. Keim. “Finding anomalies in time-series using visual correlation for interactive root cause analysis”. In: *Proceedings of the Tenth Workshop on Visualization for Cyber Security.* VizSec ’13. New York, NY, USA: Association for Computing Machinery, Oct. 2013, pp. 65–72. URL: https://doi.org/10.1145/2517957.2517966 (visited on 08/28/2020).

[68] Lothar Braun et al. “Flow-inspector: a framework for visualizing network flow data using current web technologies”. en. In: *Computing* 96.1 (Jan. 2014), pp. 15–26. URL: https://doi.org/10.1007/s00607-013-0286-4 (visited on 08/28/2020).

[69] Cameron C. Gray, Panagiotis D. Ritsos, and Jonathan C. Roberts. “Contextual network navigation to provide situational awareness for network administrators”. In: *2015 IEEE Symposium on Visualization for Cyber Security (VizSec).* Oct. 2015, pp. 1–8.

[70] Marco Angelini, Nicolas Prigent, and Giuseppe Santucci. “PERCIVAL: proactive and reactive attack and response assessment for cyber incidents using visual analytics”. In: *2015 IEEE Symposium on Visualization for Cyber Security (VizSec).* Oct. 2015, pp. 1–8.

[71] Vinícius Tavares Guimarães et al. “A Survey on Information Visualization for Network and Service Management”. In: *IEEE Communications Surveys Tutorials* 18.1 (2016). Conference Name: IEEE Communications Surveys Tutorials, pp. 285–323.

[72] Dustin Arendt et al. “CyberPetri at CDX 2016: Real-time network situation awareness”. In: *2016 IEEE Symposium on Visualization for Cyber Security (VizSec).* Oct. 2016, pp. 1–4.

[73] Marc Coudriau, Abdelkader Lahmadi, and Jérôme François. “Topological analysis and visualisation of network monitoring data: Darknet case study”. In: *2016 IEEE International Workshop on Information Forensics and Security (WIFS).* ISSN: 2157-4774. Dec. 2016, pp. 1–6.

[74] Tobias Post et al. “Visually guided flow tracking in software-defined networking”. In: *2016 IEEE Symposium on Visualization for Cyber Security (VizSec).* Oct. 2016, pp. 1–4.

[75] Likun He et al. “NetflowVis: A Temporal Visualization System for Netflow Logs Analysis”. en. In: *Cooperative Design, Visualization, and Engineering.* Ed. by Yuhua Luo. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 202–209.

[76] Brian Trammell et al. “Peeling Away Timing Error in NetFlow Data”. en. In: *Passive and Active Measurement.* Ed. by Neil Spring and George F. Riley. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 194–203.

[77] Alfonso Iacovazzi and Yuval Elovici. “Network Flow Watermarking: A Survey”. In: *IEEE Communications Surveys Tutorials* 19.1 (2017). Conference Name: IEEE Communications Surveys Tutorials, pp. 512–530.

[78] Cheng Lei et al. “Moving Target Defense Techniques: A Survey”. en. In: *Security and Communication Networks* (July 2018). ISSN: 1939-0114 Pages: e3759626 Publisher: Hindawi Volume: 2018. URL: `https://www.hindawi.com/journals/scn/2018/3759626/` (visited on 08/28/2020).

[79] H. Okhravi et al. *Survey of Cyber Moving Targets.* Tech. rep. MIT Lincoln Laboratory, 2013. URL: `http://web.mit.edu/ha22286/www/papers/LLTechRep.pdf`.

[80] Saptarshi Debroy et al. “Frequency-minimal moving target defense using software-defined networking”. In: *2016 International Conference on Computing, Networking and Communications (ICNC).* Feb. 2016, pp. 1–6.

[81] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. “Adversary-aware IP address randomization for proactive agility against sophisticated attackers”. In: *2015 IEEE Conference on Computer Communications (INFOCOM).* ISSN: 0743-166X. Apr. 2015, pp. 738–746.

[82] Hamed Okhravi et al. “Finding Focus in the Blur of Moving-Target Techniques”. In: *IEEE Security Privacy* 12.2 (Mar. 2014). Conference Name: IEEE Security Privacy, pp. 16–26.

[83] Daniel Fraunholz et al. “Demystifying Deception Technology:A Survey”. In: *arXiv:1804.06196 [cs]* (Apr. 2018). arXiv: 1804.06196. URL: `http://arxiv.org/abs/1804.06196` (visited on 08/28/2020).

[84] Kai Wang, Xi Chen, and Yuefei Zhu. “Random domain name and address mutation (RDAM) for thwarting reconnaissance attacks”. en. In: *PLOS ONE* 12.5 (May 2017). Publisher: Public Library of Science, e0177111. URL: `https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0177111` (visited on 08/28/2020).

[85] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. “Openflow random host mutation: transparent moving target defense using software defined networking”. In: *Proceedings of the first workshop on Hot topics in software defined networks.* HotSDN ’12. New York, NY, USA: Association for Computing Machinery, Aug. 2012, pp. 127–132. URL: `https://doi.org/10.1145/2342441.2342467` (visited on 08/28/2020).

[86] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. “An Effective Address Mutation Approach for Disrupting Reconnaissance Attacks”. In: *IEEE Transactions on Information Forensics and Security* 10.12 (Dec. 2015). Conference Name: IEEE Transactions on Information Forensics and Security, pp. 2562–2577.

[87] D. Kewley et al. “Dynamic approaches to thwart adversary intelligence gathering”. In: *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX’01.* Vol. 1. June 2001, 176–185 vol.1.

[88] S. Antonatos et al. "Defending against hitlist worms using network address space randomization". In: *Proceedings of the 2005 ACM workshop on Rapid malcode*. WORM '05. New York, NY, USA: Association for Computing Machinery, Nov. 2005, pp. 30–40. URL: `https://doi.org/10.1145/1103626.1103633` (visited on 08/28/2020).

[89] Matthew Dunlop et al. "MT6D: A Moving Target IPv6 Defense". In: *2011 - MILCOM 2011 Military Communications Conference*. ISSN: 2155-7586. Nov. 2011, pp. 1321–1326.

[90] Yue-bin Luo et al. "A keyed-hashing based self-synchronization mechanism for port address hopping communication". en. In: *Frontiers of Information Technology & Electronic Engineering* 18.5 (May 2017), pp. 719–728. URL: `https://doi.org/10.1631/FITEE.1601548` (visited on 08/28/2020).

[91] Yih Huang and Anup K. Ghosh. "Automating Intrusion Response via Virtualization for Realizing Uninterruptible Web Services". In: *2009 Eighth IEEE International Symposium on Network Computing and Applications*. July 2009, pp. 114–117.

[92] Dilli Prasad Sharma et al. "FRVM: Flexible Random Virtual IP Multiplexing in Software-Defined Networks". In: *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. ISSN: 2324-9013. Aug. 2018, pp. 579–587.

[93] Ehab Al-Shaer, Qi Duan, and Jafar Haadi Jafarian. "Random Host Mutation for Moving Target Defense". en. In: *Security and Privacy in Communication Networks*. Ed. by Angelos D. Keromytis and Roberto Di Pietro. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Berlin, Heidelberg: Springer, 2013, pp. 310–327.

[94] Brian Van Leeuwen, William M.S. Stout, and Vincent Urias. "Operational cost of deploying Moving Target Defenses defensive work factors". In: *MILCOM 2015 - 2015 IEEE Military Communications Conference*. Oct. 2015, pp. 966–971.

[95] Jafar Haadi H. Jafarian, Ehab Al-Shaer, and Qi Duan. "Spatio-temporal Address Mutation for Proactive Cyber Agility against Sophisticated Attackers". In: *Proceedings of the First ACM Workshop on Moving Target Defense*. MTD '14. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 69–78. URL: `https://doi.org/10.1145/2663474.2663483` (visited on 08/28/2020).

[96] Mohiuddin Ahmed, Abdun Naser Mahmood, and Jiankun Hu. "A survey of network anomaly detection techniques". en. In: *Journal of Network and Computer Applications* 60 (Jan. 2016), pp. 19–31. URL: `http://www.sciencedirect.com/science/article/pii/S1084804515002891` (visited on 08/28/2020).

[97] Diala Naboulsi et al. "Large-Scale Mobile Traffic Analysis: A Survey". In: *IEEE Communications Surveys Tutorials* 18.1 (2016). Conference Name: IEEE Communications Surveys Tutorials, pp. 124–161.

[98] Mauro Conti et al. "The Dark Side(-Channel) of Mobile Devices: A Survey on Network Traffic Analysis". In: *IEEE Communications Surveys Tutorials* 20.4 (2018). Conference Name: IEEE Communications Surveys Tutorials, pp. 2658–2713.

[99] Laurent Bernaille, Renata Teixeira, and Kave Salamatian. "Early application identification". In: *Proceedings of the 2006 ACM CoNEXT conference*. CoNEXT '06. New York, NY, USA: Association for Computing Machinery, Dec. 2006, pp. 1–12. URL: https://doi.org/10.1145/1368436.1368445 (visited on 08/28/2020).

[100] Pavel Minarik, Jan Vykopal, and Vojtech Krmicek. "Improving Host Profiling with Bidirectional Flows". In: *2009 International Conference on Computational Science and Engineering*. Vol. 3. Aug. 2009, pp. 231–237.

[101] Maddalena Favaretto et al. "You Surf so Strange Today: Anomaly Detection in Web Services via HMM and CTMC". en. In: *Green, Pervasive, and Cloud Computing*. Ed. by Man Ho Allen Au et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 426–440.

[102] Manos Antonakakis et al. "Understanding the mirai botnet". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. USA: USENIX Association, Aug. 2017, pp. 1093–1110. (Visited on 08/28/2020).

[103] Bushra A. AlAhmadi and Ivan Martinovic. "MalClassifier: Malware family classification using network flow sequence behaviour". In: *2018 APWG Symposium on Electronic Crime Research (eCrime)*. ISSN: 2159-1245. May 2018, pp. 1–13.

[104] Mark Allman, Ethan Blanton, and Vern Paxson. "An architecture for developing behavioral history". In: *Proceedings of the Steps to Reducing Unwanted Traffic on the Internet on Steps to Reducing Unwanted Traffic on the Internet Workshop*. SRUTI'05. USA: USENIX Association, July 2005, p. 7. (Visited on 08/28/2020).

[105] Kuai Xu, Zhi-Li Zhang, and Supratik Bhattacharyya. "Profiling internet backbone traffic: behavior models and applications". In: *ACM SIGCOMM Computer Communication Review* 35.4 (Aug. 2005), pp. 169–180. URL: https://doi.org/10.1145/1090191.1080112 (visited on 08/28/2020).

[106] Songjie Wei, Jelena Mirkovic, and Ezra Kissel. "Profiling and Clustering Internet Hosts". In: *Proc. of DMIN*. 2006.

[107] Thomas Karagiannis et al. "Profiling the End Host". en. In: *Passive and Active Network Measurement*. Ed. by Steve Uhlig, Konstantina Papagiannaki, and Olivier Bonaventure. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 186–196.

[108] Alice Zheng and Amanda Casari. *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. O'Reilly Media, 2018.

[109] Markus M. Breunig et al. "LOF: identifying density-based local outliers". In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. SIGMOD '00. New York, NY, USA: Association for Computing Machinery, May 2000, pp. 93–104. URL: https://doi.org/10.1145/342009.335388 (visited on 08/28/2020).

[110] Michael R. Smith and Tony Martinez. "Improving classification accuracy by identifying and removing instances that should be misclassified". In: *The 2011 International Joint Conference on Neural Networks*. ISSN: 2161-4407. July 2011, pp. 2690–2697.

[111] Mikel Galar et al. "An overview of ensemble methods for binary classifiers in multi-class problems: Experimental study on one-vs-one and one-vs-all schemes". en. In: *Pattern Recognition* 44.8 (Aug. 2011), pp. 1761–1776. URL: http://www.sciencedirect.com/science/article/pii/S0031320311000458 (visited on 08/28/2020).

[112] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *The Journal of Machine Learning Research* 12 (Nov. 2011), pp. 2825–2830.

[113] *Minkowski Distance*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.minkowski.html.

[114] J. Xu et al. "Prefix-preserving IP address anonymization: measurement-based security evaluation and a new cryptography-based scheme". In: *10th IEEE International Conference on Network Protocols, 2002. Proceedings.* ISSN: 1092-1648. Nov. 2002, pp. 280–289.

[115] Mark Berman et al. "GENI: A federated testbed for innovative network experiments". en. In: *Computer Networks*. Special issue on Future Internet Testbeds – Part I 61 (Mar. 2014), pp. 5–23. URL: http://www.sciencedirect.com/science/article/pii/S1389128613004507 (visited on 09/02/2020).

[116] Andrew W. Moore and Konstantina Papagiannaki. "Toward the Accurate Identification of Network Applications". en. In: *Passive and Active Network Measurement*. Ed. by Constantinos Dovrolis. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 41–54.

[117] Tim Shimeall. *Traffic Analysis for Network Security: Two Approaches for Going Beyond Network Flow Data*. https://insights.sei.cmu.edu/sei_blog/2016/09/traffic-analysis-for-network-security-two-approaches-for-going-beyond-network-flow-data.html.

[118] Grégoire Jacob, Hervé Debar, and Eric Filiol. "Behavioral detection of malware: from a survey towards an established taxonomy". en. In: *Journal in Computer Virology* 4.3 (Aug. 2008), pp. 251–266. URL: https://doi.org/10.1007/s11416-008-0086-0 (visited on 08/28/2020).

[119] Ethan M. Rudd et al. "A Survey of Stealth Malware Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions". In: *IEEE Communications Surveys Tutorials* 19.2 (2017). Conference Name: IEEE Communications Surveys Tutorials, pp. 1145–1172.

[120] Roberto Perdisci, Wenke Lee, and Nick Feamster. "Behavioral clustering of HTTP-based malware and signature generation using malicious network traces". In: *Proceedings of the 7th USENIX conference on Networked systems design and implementation*. NSDI'10. USA: USENIX Association, Apr. 2010, p. 26. (Visited on 08/28/2020).

[121] M. Zubair Rafique and Juan Caballero. “FIRMA: Malware Clustering and Network Signature Generation with Mixed Network Behaviors”. en. In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Salvatore J. Stolfo, Angelos Stavrou, and Charles V. Wright. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 144–163.

[122] Aziz Mohaisen et al. “Chatter: Classifying malware families using system event ordering”. In: *2014 IEEE Conference on Communications and Network Security*. Oct. 2014, pp. 283–291.

[123] S. García et al. “An empirical comparison of botnet detection methods”. en. In: *Computers & Security* 45 (Sept. 2014), pp. 100–123. URL: `http://www.sciencedirect.com/science/article/pii/S0167404814000923` (visited on 08/28/2020).

[124] FireEye. *Stages of a Malware Infection*. `https://community.fireeye.com/s/article/000002205`.

[125] M. Cotton et al. *RFC6335 - Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry*. Tech. rep. Internet Engineering Task Force (IETF), 2011. URL: `https://tools.ietf.org/html/rfc6335`.

[126] Giorgio Severi, Tim Leek, and Brendan Dolan-Gavitt. “Malrec: Compact Full-Trace Malware Recording for Retrospective Deep Analysis”. en. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 3–23.

[127] Marcos Sebastián et al. “AVclass: A Tool for Massive Malware Labeling”. en. In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Fabian Monrose et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 230–253.

[128] Adam J. Aviv and Andreas Haeberlen. “Challenges in experimenting with botnet detection systems”. In: *Proceedings of the 4th conference on Cyber security experimentation and test*. CSET’11. USA: USENIX Association, Aug. 2011, p. 6. (Visited on 08/28/2020).

[129] James Lewis. *Economic Impact of Cybercrime - No Slowing Down*. Tech. rep. McAfee and CSIS, 2018. URL: `https://www.mcafee.com/enterprise/en-us/assets/reports/restricted/rp-economic-impact-cybercrime.pdf`.

[130] Irfan Ahmed and Kyung-suk Lhee. “Classification of packet contents for malware detection”. en. In: *Journal in Computer Virology* 7.4 (Oct. 2011), p. 279. URL: `https://doi.org/10.1007/s11416-011-0156-6` (visited on 08/28/2020).

[131] Bushra A. AlAhmadi et al. “BOTection: Bot Detection by Building Markov Chain Models of Bots Network Behavior”. en. In: *Proceedings of the 2020 ACM Asia Conference on Computer and Communications Security*. Asia CCS ’20. Association for Computing Machinery, 2020. URL: `/paper/BOTection%3A-Bot-Detection-by-Building-Markov-Chain-AlAhmadi-Spolaor/b5a194948f85fcbe4aeff08c20c2483cc413b29a` (visited on 08/28/2020).

[132] Cyril Onwubiko. “Cyber security operations centre: Security monitoring for protecting business and supporting cyber defense strategy”. In: *2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*. June 2015, pp. 1–10.

[133] Toshiki Shibahara et al. “Efficient Dynamic Malware Analysis Based on Network Behavior Using Deep Learning”. In: *2016 IEEE Global Communications Conference (GLOBECOM)*. Dec. 2016, pp. 1–7.

[134] Ulrich Bayer et al. “A view on current malware behaviors”. In: *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*. LEET’09. USA: USENIX Association, Apr. 2009, p. 8. (Visited on 08/28/2020).

[135] Clemens Kolbitsch et al. “Effective and efficient malware detection at the end host”. In: *Proceedings of the 18th conference on USENIX security symposium*. SSYM’09. USA: USENIX Association, Aug. 2009, pp. 351–366. (Visited on 08/28/2020).

[136] U. Bayer et al. “Scalable, Behavior-Based Malware Clustering”. In: *NDSS*. 2009.

[137] Chaz Lever et al. “A Lustrum of Malware Network Communication: Evolution and Insights”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. May 2017, pp. 788–804.

[138] Colin C. Ife et al. “Waves of Malice: A Longitudinal Measurement of the Malicious File Delivery Ecosystem on the Web”. In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. Asia CCS ’19. New York, NY, USA: Association for Computing Machinery, July 2019, pp. 168–180. URL: `https://doi.org/10.1145/3321705.3329807` (visited on 08/28/2020).

[139] Sumayah Alrwais et al. “Under the Shadow of Sunshine: Understanding and Detecting Bulletproof Hosting on Legitimate Service Provider Networks”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. May 2017, pp. 805–823.

[140] T. Holz et al. “Measuring and Detecting Fast-Flux Service Networks”. In: *NDSS*. 2008.

[141] Daniel Plohmann et al. “A comprehensive measurement study of domain generating malware”. In: *Proceedings of the 25th USENIX Conference on Security Symposium*. SEC’16. USA: USENIX Association, Aug. 2016, pp. 263–278. (Visited on 08/28/2020).

[142] Florian Tegeler et al. “BotFinder: finding bots in network traffic without deep packet inspection”. In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. CoNEXT ’12. New York, NY, USA: Association for Computing Machinery, Dec. 2012, pp. 349–360. URL: `https://doi.org/10.1145/2413176.2413217` (visited on 08/28/2020).

[143] Ching-Hsiang Hsu, Chun-Ying Huang, and Kuan-Ta Chen. “Fast-Flux Bot Detection in Real Time”. en. In: *Recent Advances in Intrusion Detection*. Ed. by Somesh Jha, Robin Sommer, and Christian Kreibich. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 464–483.

[144] L. Invernizzi et al. “Nazca: Detecting Malware Distribution in Large-Scale Networks”. In: *NDSS*. 2014.

[145] Xiyue Deng and Jelena Mirkovic. “Malware analysis through high-level behavior”. In: *Proceedings of the 11th USENIX Conference on Cyber Security Experimentation and Test*. CSET’18. USA: USENIX Association, Aug. 2018, p. 5. (Visited on 08/28/2020).

[146] H. Aghakhani et al. “When Malware is Packin’ Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features”. In: *NDSS*. 2020.

[147] Robin Sommer and Vern Paxson. “Outside the Closed World: On Using Machine Learning for Network Intrusion Detection”. In: *2010 IEEE Symposium on Security and Privacy*. ISSN: 2375-1207. May 2010, pp. 305–316.

[148] Gonzalo Marín, Pedro Casas, and Germán Capdehourat. “Deep in the Dark - Deep Learning-Based Malware Traffic Detection Without Expert Knowledge”. In: *2019 IEEE Security and Privacy Workshops (SPW)*. May 2019, pp. 36–42.

[149] Ziyun Zhu and Tudor Dumitraş. “FeatureSmith: Automatically Engineering Features for Malware Detection by Mining the Security Literature”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 767–778. URL: `https://doi.org/10.1145/2976749.2978304` (visited on 08/28/2020).

[150] Ali Gezer et al. “A flow-based approach for Trickbot banking trojan detection”. en. In: *Computers & Security* 84 (July 2019), pp. 179–192. URL: `http://www.sciencedirect.com/science/article/pii/S0167404818309568` (visited on 08/28/2020).

[151] Emily H. Do and Vijay N. Gadepally. “Classifying Anomalies for Network Security”. In: *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. ISSN: 2379-190X. May 2020, pp. 2907–2911.

[152] Robert Lyda and James Hamrock. “Using Entropy Analysis to Find Encrypted and Packed Malware”. In: *IEEE Security Privacy* 5.2 (Mar. 2007). Conference Name: IEEE Security Privacy, pp. 40–45.

[153] M. Zubair Shafiq, Syed Ali Khayam, and Muddassar Farooq. “Embedded Malware Detection Using Markov n-Grams”. en. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Diego Zamboni. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 88–107.

[154] Alessandro Mantovani et al. “Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem”. In: *NDSS*. 2020.

[155] Apostolis Zarras et al. “Automated generation of models for fast and precise detection of HTTP-based malware”. In: *2014 Twelfth Annual International Conference on Privacy, Security and Trust*. July 2014, pp. 249–256.

[156] Pavlos Lamprakis et al. “Unsupervised Detection of APT C&C Channels using Web Request Graphs”. en. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Michalis Polychronakis and Michael Meier. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 366–387.

[157] Cisco Security. *What Is the Difference: Viruses, Worms, Trojans, and Bots?* `https://tools.cisco.com/security/center/resources/virus_differences`.

[158] Andrea Continella et al. “ShieldFS: a self-healing, ransomware-aware filesystem”. In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACSAC ’16. New York, NY, USA: Association for Computing Machinery, Dec. 2016, pp. 336–347. URL: `https://doi.org/10.1145/2991079.2991110` (visited on 08/28/2020).

[159] Pascal Vincent et al. “Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion”. In: *The Journal of Machine Learning Research* 11 (Dec. 2010), pp. 3371–3408.

[160] Lovedeep Gondara. “Medical Image Denoising Using Convolutional Denoising Autoencoders”. In: *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. ISSN: 2375-9259. Dec. 2016, pp. 241–246.

[161] Junyuan Xie, Linli Xu, and Enhong Chen. “Image denoising and inpainting with deep neural networks”. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’12. Red Hook, NY, USA: Curran Associates Inc., Dec. 2012, pp. 341–349. (Visited on 08/28/2020).

[162] *ClamAV*. `https://www.clamav.net/downloads`.

[163] Hyrum S. Anderson and Phil Roth. “EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models”. In: *arXiv:1804.04637 [cs]* (Apr. 2018). arXiv: 1804.04637. URL: `http://arxiv.org/abs/1804.04637` (visited on 08/28/2020).

[164] *GT Malware Netflow Daily Feed 2018*. `https://www.impactcybertrust.org/dataset_view?idDataset=1143`.

[165] *MalShare*. `https://malshare.com/`.

[166] *VirusShare*. `https://virusshare.com/hashes.4n6`.

[167] *VX Underground*. `https://vx-underground.org/samples.html`.

[168] *Malware Capture Facility Project*. `https://www.stratosphereips.org/datasets-normal`.

[169] *Yet Another Flowmeter*. `https://tools.netsa.cert.org/yaf/`.

[170] Christian Rossow et al. “Prudent Practices for Designing Malware Experiments: Status Quo and Outlook”. In: *2012 IEEE Symposium on Security and Privacy*. ISSN: 2375-1207. May 2012, pp. 65–79.

[171] *The ZeuS, ZBOT, and Kneber Connection*. `https://www.trendmicro.com/vinfo/us/threat-encyclopedia/web-attack/16/the-zeus-zbot-and-kneber-connection`.

[172] *ZBOT-UPATRE Far From Game Over, Uses Random Headers*. `https://blog.trendmicro.com/trendlabs-security-intelligence/zbot-upatre-far-from-game-over-uses-random-headers/`.

[173] *VirusTotal - Ulise Zusy*. `https://www.virustotal.com/gui/file/7891c60f2e6fec81733f3e7a5baca9d3cd894662ed6f977c2577895bc552f10a/detection`.

[174] *What is Trojan:Win32/Ulise!MSR infection?* https://howtofix.guide/trojanwin32-ulisemsr/.

[175] Matilda Rhode, Pete Burnap, and Kevin Jones. "Early-stage malware prediction using recurrent neural networks". en. In: *Computers & Security* 77 (Aug. 2018), pp. 578–594. URL: http://www.sciencedirect.com/science/article/pii/S0167404818305546 (visited on 08/28/2020).

[176] Eric Filiol. *Computer Viruses: from theory to applications*. en. Collection IRIS. Paris: Springer-Verlag, 2005. URL: https://www.springer.com/gp/book/9782287280993 (visited on 08/28/2020).

[177] Najmeh Miramirkhani et al. "Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts". In: *2017 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. May 2017, pp. 1009–1024.

[178] Akira Yokoyama et al. "SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion". en. In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Fabian Monrose et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 165–187.

[179] Gregoire Jacob et al. "JACKSTRAWS: picking command and control connections from bot traffic". In: *Proceedings of the 20th USENIX conference on Security*. SEC'11. USA: USENIX Association, Aug. 2011, p. 29. (Visited on 08/28/2020).

[180] Fabio De Gaspari et al. "The Naked Sun: Malicious Cooperation Between Benign-Looking Processes". en. In: *Applied Cryptography and Network Security*. Lecture Notes in Computer Science. Springer International Publishing, 2020.

[181] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: *OpenAI*. 2019.

[182] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *arXiv:1810.04805 [cs]* (May 2019). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805 (visited on 08/28/2020).

[183] Mark Thomas et al. "SiLK: A Tool Suite for Unsampled Network Flow Analysis at Scale". In: *2014 IEEE International Congress on Big Data*. ISSN: 2379-7703. June 2014, pp. 184–191.

[184] Peter J. Desnoyers and Prashant Shenoy. "Hyperion: high volume stream archival for retrospective querying". In: *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*. ATC'07. USA: USENIX Association, June 2007, pp. 1–14. (Visited on 08/28/2020).

[185] Gregor Maier et al. "Enriching network security analysis with time travel". In: *ACM SIGCOMM Computer Communication Review* 38.4 (Aug. 2008), pp. 183–194. URL: https://doi.org/10.1145/1402946.1402980 (visited on 08/28/2020).

[186] Francesco Fusco et al. "pcapIndex: an index for network packet traces with legacy compatibility". In: *ACM SIGCOMM Computer Communication Review* 42.1 (Jan. 2012), pp. 47–53. URL: https://doi.org/10.1145/2096149.2096156 (visited on 08/28/2020).

[187] Francesco Fusco, Marc Ph. Stoecklin, and Michail Vlachos. "NET-FLi: on-the-fly compression, archiving and indexing of streaming network traffic". In: *Proceedings of the VLDB Endowment* 3.1-2 (Sept. 2010), pp. 1382–1393. URL: `https://doi.org/10.14778/1920841.1921011` (visited on 08/28/2020).

[188] *Argus*. `https://qosient.com/argus/`.

[189] *flowd*. `https://code.google.com/archive/p/flowd/`.

[190] Petr Velan and Radek Krejčí. "Flow Information Storage Assessment Using IPFIXcol". en. In: *Dependable Networks and Services*. Ed. by Ramin Sadre et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 155–158.

[191] *nfdump*. `https://github.com/phaag/nfdump`.

[192] *pmacct*. `http://www.pmacct.net/`.

[193] Ronny T. Lampert et al. "Vermont - A Versatile Monitoring Toolkit for IPFIX and PSAMP". In: *MonAM*. 2006.

[194] *MariaDB ColumnStore*. `https://mariadb.com/kb/en/library/mariadb-columnstore/`.

[195] K. Wu et al. "FastBit: interactively searching massive data". en. In: *Journal of Physics: Conference Series* 180 (July 2009). Publisher: IOP Publishing, p. 012053. URL: `https://doi.org/10.1088%2F1742-6596%2F180%2F1%2F012053` (visited on 08/28/2020).

[196] Luca Deri, Valeria Lorenzetti, and Steve Mortimer. "Collection and Exploration of Large Data Monitoring Sets Using Bitmap Databases". en. In: *Traffic Monitoring and Analysis*. Ed. by Fabio Ricciato, Marco Mellia, and Ernst Biersack. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 73–86.

[197] Rick Hofstede et al. "The Network Data Handling War: MySQL vs. NfDump". en. In: *Networked Services and Applications - Engineering, Control and Management*. Ed. by Finn Arve Aagesen and Svein Johan Knapskog. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 167–176.

[198] Francesco Fusco and Luca Deri. "High speed network traffic analysis with commodity multi-core systems". In: *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. IMC '10. New York, NY, USA: Association for Computing Machinery, Nov. 2010, pp. 218–224. URL: `https://doi.org/10.1145/1879141.1879169` (visited on 08/28/2020).

[199] Jihyung Lee et al. "FloSIS: a highly scalable network flow capture system for fast retrieval and storage efficiency". In: *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '15. USA: USENIX Association, July 2015, pp. 445–457. (Visited on 08/28/2020).

[200] Zhen Chen et al. "A survey of bitmap index compression algorithms for Big Data". In: *Tsinghua Science and Technology* 20.1 (Feb. 2015). Conference Name: Tsinghua Science and Technology, pp. 100–115.

[201] Sirish Chandrasekaran et al. "TelegraphCQ: continuous dataflow processing". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. SIGMOD '03. New York, NY, USA: Association for Computing Machinery, June 2003, p. 668. URL: https://doi.org/10.1145/872757.872857 (visited on 08/28/2020).

[202] Chuck Cranor et al. "Gigascope: a stream database for network applications". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. SIGMOD '03. New York, NY, USA: Association for Computing Machinery, June 2003, pp. 647–651. URL: https://doi.org/10.1145/872757.872838 (visited on 08/28/2020).

[203] Xin Li et al. "Advanced Indexing Techniques for Wide-Area Network Monitoring". In: *21st International Conference on Data Engineering Workshops (ICDEW'05)*. Apr. 2005, pp. 1184–1184.

[204] Frederick Reiss et al. "Enabling Real-Time Querying of Live and Historical Stream Data". In: *19th International Conference on Scientific and Statistical Database Management (SSDBM 2007)*. ISSN: 1551-6393. July 2007, pp. 28–28.

[205] Zhen Chen et al. "TIFAflow: enhancing traffic archiving system with flow granularity for forensic analysis in network security". In: *Tsinghua Science and Technology* 18.4 (Aug. 2013). Conference Name: Tsinghua Science and Technology, pp. 406–417.

[206] Gaogang Xie et al. "Index–Trie: Efficient archival and retrieval of network traffic". en. In: *Computer Networks* 124 (Sept. 2017), pp. 140–156. URL: http://www.sciencedirect.com/science/article/pii/S1389128617302542 (visited on 08/28/2020).

[207] Francesco Fusco, Michail Vlachos, and Xenofontas Dimitropoulos. "RasterZip: compressing network monitoring data with support for partial decompression". In: *Proceedings of the 2012 Internet Measurement Conference*. IMC '12. New York, NY, USA: Association for Computing Machinery, Nov. 2012, pp. 51–64. URL: https://doi.org/10.1145/2398776.2398783 (visited on 08/28/2020).

[208] Gilbert Held and Thomas Marshall. *Data Compression: Techniques and Applications: Hardware and Software Considerations (2nd Ed.)* Wiley, 1986.

[209] *FlatBuffers*. https://google.github.io/flatbuffers/.