DETECTION AND EVALUATION OF DATA EXFILTRATION

Riccardo Bortolameotti

DETECTION AND EVALUATION OF DATA EXFILTRATION

Riccardo Bortolameotti

DETECTION AND EVALUATION OF DATA EXFILTRATION

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
Prof.dr. T.T.M. Palstra,
on account of the decision of the graduation committee,
to be publicly defended
on Friday 11th October 2019 at 12.45

by

Riccardo Bortolameotti

born on 12th of July 1990
in Trento, Italy

This dissertation has been approved by:

supervisors

Prof.dr. P.H. Hartel

Prof.dr. W. Jonker

co-supervisor

Dr. A. Peter

This research has been partially supported by the THeCS project, as part of the Dutch national program COMMIT/, and by the INAETICS project, as part of the European Regional Development Fund.

**UNIVERSITY
OF TWENTE.**

**DIGITAL SOCIETY
INSTITUTE**

Services and Cybersecurity Group

P.O. Box 217, 7500 AE

Enschede, the Netherlands

DSI Ph.D. Thesis Serie No. 19-013

Digital Society Institute

P.O. Box 217, 7500 AE

Enschede, the Netherlands

ISBN: 978-90-365-4824-3

ISSN: 2589-7721

DOI: 10.3990/1.9789036548243

<https://doi.org/10.3990/1.9789036548243>

Typeset with L^AT_EX.

Printed by: Ipskamp Printing

Cover design by: WDStudio and Riccardo Bortolameotti

Copyright © 2019 Riccardo Bortolameotti, Enschede, the Netherlands

All rights reserved. No part of this book may be reproduced or transmitted, in any form or by any means, electronic or mechanical, including photocopying, microfilming, and recording, or by any information storage or retrieval systems, without the prior written permission of the author.

Graduation Committee:

Chairman/Secretary	Prof. dr.	J.N. Kok	Universiteit Twente
Supervisors	Prof. dr.	P.H. Hartel	Universiteit Twente
	Prof. dr.	W. Jonker	Universiteit Twente
Co-Supervisor	Dr.	A. Peter	Universiteit Twente
Committee Members	Prof. dr. ir.	A. Pras	Universiteit Twente
	Prof. dr. ir.	R.N.J. Veldhuis	Universiteit Twente
	Prof. dr. ir.	H. Bos	Vrije Universiteit Amsterdam
	Prof. dr.	R. Perdisci	University of Georgia
	Dr.	M. Cova	Lastline Inc.

to Vera, Remo, Chiara and Anna.

Acknowledgments

What a memorable ride. Five years of emotional ups and downs. Joyful moments intertwined with distressful ones. During this rollercoaster of feelings, I have learned so much from so many people that I will never regret the choice of pursuing a PhD. It was a unique and sensational journey.

Dear Andreas, you are an amazing supervisor. Not only you taught me how to do research and how to express it, but you also helped me through all the difficulties I faced during this journey. You taught me to be positive and to look at the bright side of things, even when things were quite far from being positive. You were always available to listen to my ideas (and to shoot at them), to guide me in my uncertainties, and most importantly to support me when I had personal problems. I will be always thankful for this. You are a great listener.

Dear Pieter and Willem, thank you for pushing my limits. I don't think I have ever been challenged as much before. At times it felt like a stress test, but this taught me many important lessons. I have learned to *correctly* interpret feedback (spoiler alert: it is not a personal attack), to criticize my ideas, to improve and "sharpen" my communication (still working on it), and to plan my work. Now, to be completely honest, now I cannot work without making a plan first. Pieter, a special thanks for being patient and supportive every time I failed.

Dear Maarten, thank you for helping me with my research. Especially at the beginning, when we had many meetings to discuss about research problems to solve, and crazy ideas about how to solve them. Thank you also for always being available to help me with the writing.

Thank you to the Committee to take the time to read and to provide feedback about my work.

Thank you to Bertine, Geert Jan, and Suse, the core pillars of our research group. Bertine, if I got a house in 2014 was because of you. Geert Jan, thanks for supporting me with the labs and technical issues. Your contribution was very important for my research.

Although the PhD is mainly a personal quest, I was very lucky to share my experience with other brave adventurers. Tim, it was a lot of fun to work with you for four years. Thanks for supporting and tolerating me for so many years. By the way, somehow I miss *a bit* of that nasty humor. Thijs, I am very glad I had the chance to work with you for two years. It was a lot of fun working with you and I enjoyed our "whiteboard session" lasting entire afternoons. Andrea, thank you for being always there until the end, until the deadline, ready to help me

improve my work. Looking backward I am quite impressed by how we are still collaborating for almost three years across different timezones and with weekly meetings. Marco, thanks for teaching me the importance of networking, it helped me a lot. Since you left, the office has never been the same. It was a lot of fun with you and Ali in the office. I cannot forget the boardgames sessions and drinks at the De Vluchte with Dan, Roeland, Chris (thanks for your explanations about probabilities). I wish we would have found out about this passion a bit earlier. Thanks also to my other colleagues in the research group with whom I shared nice moments together with Ali (thanks for always being supportive), Herson, Elmer, Eleftheria, Valeriu, Phillip, Thomas, Erik, Prince, Lorena, Alexandr, Jan Willem, Susanne, Joao, Hans.

This adventure started after my internship at SecurityMatters. Thank you, Damiano and Emmanuele, because you introduced me to this topic and you introduced me to research. Thank you, Marco and Corrado, allowing me to join you in London. It certainly was one of the highlights of my PhD. I learned a lot from you despite the few months spent there. How can I ever forget my first code review?! Thanks also to all the London office, which made me feel at home and trained me quite well at table football. In particular, I want to thank Filippo, Andy, Stefano, Alessandro, and Luukas for the nice evenings we had at the Griffin. I also want to thank RedSocks for helping me with my research. Thank you Rick, Adrianus and Reza. Adrianus, I really enjoyed our discussions on IRC.

Well, if I made it to the end it is not only thanks to my colleagues but also my friends. Federico, thank you for hosting me at your place every week for our nerd nights, I needed them. Thanks to my D&D party companions and masters: Roelof, Nicholas, Luigi, Federico, and Tommaso. It was great fun to play together. For once I felt like an athletic person with my +10 of acrobatics. Thanks, Caterina and Etorino for tolerating us in your house. Tassos, Carles, Nabila, Bram and Francesca thank you for all the moments, weekends and weeks we spent together. Thanks, Cristina, Giorgio, and Daniele for the nice moments spent together, especially at the beginning of this adventure.

Roel, Roeli, and Jolien thank you for always being so kind with me and to accept me as a member of your family. Your presence and support helped me a lot during these years, and I feel at home when I am with you. You became a second family to me.

Roby, Pampa, Bacco e Mozk grazie per le vostre trasferte e per essermi stati sempre vicini, nonostante la distanza e nonostante non usassi WhatsApp. Anche se oramai ci vediamo poche volte all'anno, mi da sempre grande energia rivedervi e passar tempo con voi. Alby, Mela, e Ruzz grazie per tutte le serate passate assieme negli ultimi anni. Nonostante la distanza, spero di farne ancora tante.

Mamma e papà, grazie mille di tutto. Questo risultato è anche vostro. Vi ringrazio per esser sempre stati pazienti con me, e per avermi sempre dato la libertà di scegliere quello che volevo fare. Vi ringrazio per avermi sostenuto nelle mie scelte e non aver mai dubitato delle mie capacità. Nel 2013 sia io che Anna siamo usciti di casa e vi siete ritrovati con un grande vuoto a casa. Per voi è stato un duro colpo, però spero siate orgogliosi di voi stessi per quello che avete fatto. Ci avete insegnato a volare, e ora stiamo volando da soli verso i nostri obiettivi.

Annina, sono super orgoglioso di te. In questi anni ti ho vista crescere, raggiungere i tuoi obiettivi, e crearti una vita in una città e in un contesto completamente nuovi. Vorrei farti notare che tutto questo te lo sei creata da sola. In passato so di non esser stato il fratellone perfetto, sto cercando di recuperare piano piano. Grazie per avermi sopportato in questi anni.

Antonio, Dino e Rina, sin da bambino mi avete insegnato l'importanza dello studio e dell'impegno. Ammetto di non avervi ascoltato per molti anni. Mi son svegliato un pò tardi, però posso assicurarvi che ho messo in pratica quello che mi avete insegnato. Ho studiato e mi son impegnato, tanto. Questo è il risultato, e che dire ... avevate ragione. Mi duole il cuore per non poter condividere questo momento con voi. Spero di aver ripagato, seppur in parte, tutto quello che mi avete insegnato. Nonna Rosy, per fortuna te non molli mai. Spero di venirti a trovare presto e raccontarti di quest'avventura e celebrarla assieme.

Vera, I met you the first week I arrived in the Netherlands. You have been there since day one. You accompanied me throughout this journey. You helped me going through the bad days, and you celebrated with me the good days. These six years together have been memorable, and many more have still to come. You brought stability and love in my life. I would have never done this without you. I love you. You mean the world to me.

Riccardo Bortolameotti
Den Haag, 26/08/2019

Abstract

Nowadays data breaches are one of the most prominent cyber incidents affecting enterprises across all industries. These incidents are not only an issue for the finances and reputation of companies, but they are also a legal problem. According to data breach notification laws, companies are obliged to disclose incident details, including the number of affected individuals. Thus, companies need technical solutions to detect data breaches and to evaluate their impact.

The development of defensive mechanisms against data breaches is difficult because attackers deploy offensive techniques that increase in sophistication. This technical development is the consequence of the attacker's need of being stealthy, in order to perpetrate her offensive actions for a longer period of time.

Currently, companies lack in efficient defensive mechanisms to deal with such sophisticated attacks: (1) traditional detection systems do not effectively detect data exfiltration attacks, because they either cannot detect new and sophisticated attacks or they produce too many false alerts; (2) traditional logging systems are not designed to protect information from an attacker that compromised parts of the system and thus they cannot be relied upon for evaluating the impact of a data breach. This thesis proposes technical systems that can help companies to better detect and evaluate data breaches despite the presence of sophisticated attacks.

Concretely, we investigate the problem of detecting data exfiltration over HTTP and we propose different technical solutions to tackle it. Data exfiltration detection is a difficult problem because there are no clear predefined patterns to be identified. The attacker chooses how much data to hide, how to encode it, *etc.* Attackers can further improve the stealthiness of their communication by mimicking the traffic of their victim. In this thesis we address both scenarios: *non-mimicking and mimicking attacks*. In the setting of non-mimicking attacks, traditional signature-based detection solutions are not effective because they cannot detect unforeseen attacks. Similarly, existing anomaly-based detection systems rely on coarse-grained models that are imprecise and often miss malicious communication. Thus, we introduce a new anomaly-based detection approach for data exfiltration called *passive application fingerprinting*, which relies on fine-grained detection models to better identify anomalous connections. We show that our proposed system outperforms the current state-of-the-art solutions in terms of detection performance and evasion resistance. Moreover, we evaluate the current state-of-the-art detection systems against mimicking attackers over HTTP, and we show that none of them can accurately detect malicious communication while triggering few false alerts.

The reason is that mimicked communication helps malicious traffic to not deviate from normal traffic, thereby breaking a fundamental assumption in detection systems. Consequently, we present *honey traffic*, a deception-based detection system to identify mimicked communication, without relying on the same assumptions as existing approaches. The main idea is to generate fake network messages that an attacker may mimic while observing the victim communication. If an attacker mimics fake messages, then a security monitor detects the attacker by identifying inconsistencies between the original and mimicked messages.

We also present a technical solution for the impact evaluation of a data breach. Existing logging mechanisms are not reliable for impact evaluation because they can be tampered with by an attacker. The reason behind this is that machines are the *sole* responsible to generate the content of the log. Once they are compromised, it is not possible to know whether the content is legitimate or not. We present a distributed logging system to determine what has leaked after a data breach by combining threshold cryptography and Byzantine consensus protocols. Compared with related work, our system is more reliable in adversarial environments and more precise in determining what data has leaked.

To conclude, our work reduces the technical gap for detecting data exfiltration over HTTP for non-mimicking attackers, by providing better solutions than the current state-of-the-art. We provide insights in the inherent limitations of passive network monitoring solutions against mimicking attacks, specifically, for a sub-category that we call victim-aware adaptive covert channels. Our work makes a step forward in addressing the open problem of detecting victim-aware adaptive covert channels by introducing the first detection mechanism for such threat. Finally, under certain assumptions, we solve the problem of determining what has leaked after a data breach in adversarial environments.

Samenvatting

Tegenwoordig zijn datalekken één van de meest prominente cyberincidenten waar bedrijven in alle sectoren te maken mee hebben. Deze incidenten vormen niet alleen een probleem voor de financiën en de reputatie van bedrijven, maar leiden ook tot juridische kwesties. Door de meldplicht datalekken worden bedrijven verplicht om details van incidenten te openbaren, waaronder het aantal getroffen gebruikers. Als gevolg hebben bedrijven technische oplossingen nodig om datalekken te detecteren en om hun impact te bepalen.

De ontwikkeling van defensieve technieken tegen datalekken is moeilijk omdat aanvallers steeds geavanceerdere offensieve technieken gebruiken. Deze technologische ontwikkeling is het gevolg van de aanvallers noodzaak om ongezien te blijven, zodat deze langer zijn offensieve acties kan uitvoeren.

Momenteel hebben bedrijven geen efficiënte defensieve technieken om met deze geavanceerde aanvallen om te gaan: (1) traditionele detectiesystemen zijn niet effectief in het detecteren van data-exfiltratie, òf omdat ze nieuwe en geavanceerde aanvallen niet kunnen detecteren, òf omdat ze te veel onjuist alarm slaan; (2) traditionele logsystemen zijn niet ontworpen om te beschermen tegen een aanval-ler die delen van het systeem heeft overgenomen, en dus kunnen ze niet worden vertrouwd wanneer er een datalek moet worden onderzocht. Dit proefschrift stelt technische systemen voor, die bedrijven helpen om, ondanks de aanwezigheid van geavanceerde aanvallen, beter datalekken te kunnen detecteren en te onderzoeken.

Concreet onderzoeken we het probleem om data-exfiltratie via HTTP te detecteren en stellen we verschillende technische oplossing voor. Het detecteren van data-exfiltratie is een moeilijk probleem omdat er geen duidelijk vooraf te definiëren patronen zijn. De aanvaller kiest zelf hoeveel data te verbergen, hoe het wordt gecodeerd, enz. Aanvallers kunnen hun communicatie beter verbergen door het verkeer van hun slachtoffers na te bootsen. In dit proefschrift behandelen we beide gevallen: *niet-imitatie- en imitatie-aanvallen*. In het geval van niet-imitatie-aanvallen zijn traditionele op signatuur gebaseerde detectie-oplossingen niet effectief, omdat ze onvoorziene aanvallen niet kunnen detecteren. Evenzo zijn bestaande op afwijking gebaseerde detectiesystemen afhankelijk van onnauwkeurige modellen en missen ze kwaadaardige communicatie vaak. Hierom introduceren wij een nieuwe op afwijking gebaseerde aanpak voor de detectie van data-exfiltratie, genaamd *passieve applicatie herkenning*. Deze aanpak gebruikt verfijnde detectiemodellen om afwijkende verbindingen beter te identificeren. We tonen aan dat ons voorgesteld systeem de huidige oplossingen overtreffen met betrekking tot de detectie

en het tegengaan van omzeilen van detectie. Bovendien evalueren we de huidige state-of-the-art detectiesystemen tegen imitatie-aanvallen via HTTP en tonen we aan dat géén van deze accuraat kwaadaardige verbindingen kan detecteren zonder veel onjuist alarm te slaan. Dit is een gevolg van nagebootste communicatie dat kwaadaardig verkeer niet laat afwijken van normaal verkeer en daarbij een fundamentele aanname in ons detectiemodel schendt. Derhalve presenteren wij *honey verkeer*, een op misleiding gebaseerd detectiesysteem om nagebootste communicatie te identificeren zonder gebruik te maken van dezelfde aannamen in gebruikelijke methoden. De hoofdgedachte is om nepverkeer te genereren wat een aanvaller zou kunnen nabootsen na het zien van de communicatie van het slachtoffer. Als een aanvaller nepberichten nabootst zal een monitor de aanval detecteren door inconsistenties tussen het origineel en de nagebootste berichten te identificeren.

We presenteren ook een technische oplossing voor het bepalen van de impact van een datalek. Bestaande logsystemen zijn niet betrouwbaar voor het bepalen van de impact omdat logs kunnen worden vervalst door een aanvaller. De achterliggende reden is dat machines de *enige* verantwoordelijke zijn voor het genereren van de logs. Wanneer deze overgenomen zijn, is het niet meer mogelijk om te bepalen of de logs legitiem zijn. Wij presenteren een gedistribueerd logstelsel om te bepalen wat er is uitgelekt na een datalek door gebruik te maken van *threshold* cryptografie en *Byzantine consensus protocols*. Vergeleken met verwante werken is ons systeem betrouwbaarder in een vijandige omgeving en preciezer in het bepalen van welke data is uitgelekt.

Samenvattend, ons werk reduceert de technische kloof voor het detecteren van data-exfiltratie via HTTP voor niet-imitatie-aanvallers door betere oplossingen aan te dragen dan de huidige. We geven inzicht in de intrinsieke beperkingen van passieve netwerkmonitoring om imitatie-aanvallen te detecteren. In het bijzonder voor een ondercategorie die we *victim-aware adaptive covert channels* noemen. Ons werk maakt een stap voorwaarts in het aanpakken van het openstaande probleem om *victim-aware adaptive covert channels* te detecteren door het introduceren van de eerste detectietechniek voor deze dreiging. Tot slot lossen we het probleem, onder bepaalde voorwaarden, op om in binnengedrongen omgevingen te bepalen wat er is uitgelekt na een datalek.

Contents

Abstract	i
Samenvatting	iii
1 Introduction	1
1.1 Motivation	2
1.2 Problem Statements	3
1.3 Thesis Overview and Contributions	9
2 Intrusion Detection	13
2.1 Anomaly-based Intrusion Detection	14
2.2 Performance Evaluation	20
2.3 Data Exfiltration Detection	21
2.4 Summary	27
3 Detecting Data Exfiltration via Application Fingerprinting	29
3.1 Motivation	30
3.2 Intuition	30
3.3 Threat Model	31
3.4 DECANTeR	32
3.5 Evaluation	42
3.6 Discussion and Limitations	51
3.7 Conclusions	54
4 Enhancing Passive Application Fingerprinting	55
4.1 Motivation	56
4.2 Intuition	56
4.3 HEADPRINT	57
4.4 Evaluation	64
4.5 Limitations	74
4.6 Conclusions	76

5	Victim-Aware Adaptive Data Exfiltration	77
5.1	Motivation	78
5.2	Intuition	80
5.3	Threat Model	80
5.4	CHAMELEON	81
5.5	Evaluation	85
5.6	Limitations	94
5.7	Conclusions	95
6	Detecting Victim-Aware Adaptive Data Exfiltration via HONEY-TRAFFIC	97
6.1	Motivation	98
6.2	Intuition	98
6.3	Threat Model	99
6.4	HONEYTRAFFIC	99
6.5	Evaluation	104
6.6	Limitations	110
6.7	Conclusions	112
7	Impact Evaluation of Data Exfiltration by Determining What Has Leaked	115
7.1	Motivation	116
7.2	Intuition	117
7.3	Building Blocks	117
7.4	System and Threat Model	119
7.5	Security Requirements for a Reliable Log	120
7.6	A Distributed Secure Log	120
7.7	Discussion	134
7.8	Related Work	135
7.9	Conclusions	136
8	Concluding Remarks	139
8.1	Contributions	139
8.2	Open Problems	142

Chapter 1

Introduction

In the last decades, the way people communicate has changed dramatically. The Internet and computers are at the center of this revolution. While the Internet has enabled more efficient communication worldwide, computers have facilitated the process and storage of large quantities of information. Today, almost any business is partially, or totally, relying on these two fundamental building blocks. Most companies have a website, communicate via emails, or have an information system for administrative tasks [6]. Digital technologies provide a benefit because they process, transfer, and store data that is important for the business. Today, *data is a valuable asset*.

Unfortunately, data is also valuable for illegal businesses run by cyber criminals, whose goal is to attack companies' IT infrastructures, steal their valuable and sensitive data, and sell it on the underground market. Examples of stolen data include: customer personal information (e.g., social security numbers, credit card numbers, or medical records), company secrets (e.g., intellectual property or product blueprints) and sensitive employee information (e.g., salaries or emails). According to a recent study [7], the cybercrime economy has reached at least \$1.5 trillion in revenues per year, and at least \$500 billion of which is from trading secrets and intellectual properties theft, and another \$160 billion are from to data trading.

An incident where information is stolen or taken from a system without knowledge or authorization of the system's owner, is a *data breach* [8]. This type of incident does not only affect small or medium businesses, but also major corporations such as Yahoo, eBay, Sony, JP Morgan Chase, and Equifax just to name a few [9]. The 2018 data breach analysis from the Ponemon Institute estimates the average costs of a data breach around \$3.86M [10], which increased compared to previous years. Mega breaches with 1 million of compromised data records can cost in excess of \$40M.

A data breach does not only affect a company and its business, but also its customers' privacy. Governments consider this threat an important issue for society and started to enforce regulations known as data breach notification laws. These legislative acts oblige companies to disclose detailed information about data

breaches, including the individuals affected by it, to the appropriate authorities. Data breach notification laws are active in both the United States [11] and the European Union (article 33, Regulation (EU) 2016/679).

The increasing number of incidents, the costs to deal with them, and the legal obligations highlight the need for companies to protect their business from data breaches. In this work we propose different technical solutions that help companies to protect themselves.

1.1 Motivation

The protection of a company’s infrastructure from cyber attacks is a continuous process. From an abstract point of view, the process of protecting an infrastructure can be described as a life-cycle of three phases. The first phase is *Prevention*, which involves the understanding of risks the organization is exposed to, and the implementation of the appropriate security measures to protect the infrastructure. The second phase is *Detection*, which focuses on the discovery of cyber security incidents within the organization. Finally, the third phase is *Mitigation*, which involves the containment and eradication of the security threat, and the analysis of the incident, including the understanding of its impact. The lessons learned, after an incident is mitigated, can be given as input to the Prevention and Detection phases, in order to strengthen the security posture of the organization against similar future incidents.

The continuous process described by the aforementioned phases can be associated with any type of cyber incident. However, this work focuses specifically on *data breaches*. A data breach is a cyber incident where the attacker succeeded to compromise one or more systems within the targeted infrastructure, and stole sensitive data from it. The act of stealing data from a compromised system through the network is known as *data exfiltration*.

The prevention of a cyber attack has always been the main focus of the security community. Thanks to the efforts of many professionals, companies can rely on many procedures, technologies, and tools that help preventing many data breaches attempts. For example, there are well-known technologies such as access control [28] and cryptography, which can prevent unauthorized data access and preserve data confidentiality. The transmission of sensitive data towards unauthorized locations can be prevented using tools such as data loss prevention (DLP) systems [29]. There exist technologies that recognize malicious software and prevent its execution, such as intrusion prevention systems [30] and malware sand-boxes [31]. Furthermore, risk assessment methodologies help companies to understand the risks their business may encounter and help them prioritizing and identifying proper countermeasures. Lastly, vulnerability management focuses mostly on identifying and patching software vulnerabilities observed in the infrastructure. Most of these prevention solutions are macro-areas of security research and, although they may not seem specifically related to data breaches, they all contribute to data breach prevention.

Despite the plethora of prevention solutions, data breaches frequently occur [12].

Consequently, companies need technologies and tools to detect and mitigate the impact of data breaches. Unfortunately, detection and mitigation technologies have seen less attention compared to prevention solutions. This technology gap is also demonstrated by the long time it takes for companies to detect data breaches and to contain them. According to the latest Ponemon data breach study [10], the mean time to identify a breach is 197 days, and 69 days to contain it. Hence, there is a need to provide technologies and tools to discover and contain data breaches more quickly.

One important aspect of providing security solutions is to understand the capabilities of the attacker. Performing a successful data breach is not an easy task. Compromising an enterprise network and exfiltrating its secrets while remaining unnoticed, requires skills and resources. This type of advanced attacker is also known as *advanced persistent threat* (APT). APTs differ from common attackers because they have a clear target in mind, they are organized and have many resources, they perform long-term campaigns, and they use evasive and stealthy techniques [32]. In the rest of this thesis, *we refer to APT as advanced attacker*. When an advanced attacker targets an organization, we must assume she may succeed. An attacker has to find a single weak spot in the entire organization in order to get access to the network. Enterprises have a large attack surface, and it is unrealistic to assume that an attacker never succeeds in gaining a foothold in the network. Therefore, we must assume that prevention solutions are not enough against this type of attacker. Furthermore, only a subset of technologies, of those few commonly used during Detection and Mitigation phases, is designed to identify and mitigate breaches perpetrated by advanced attackers. For all these reasons, we focus the efforts of our work on the Detection and Mitigation phase.

In this thesis we propose technical solutions to handle data breaches, assuming an advanced attacker is the perpetrator. More specifically, we discuss technical solutions to detect data exfiltration (i.e., Detection phase), and to evaluate the impact of a data breach (i.e., determine what has leaked, which is an activity performed during the Mitigation phase). In short, this dissertation investigates the following *main research question*:

How to detect a data breach and how to analyze its impact?

In the following section we describe the open problems, and their corresponding subquestions, that we address in this work in order to answer the main research question.

1.2 Problem Statements

We discuss the open problems addressed in this work in two separate sections. The first section discusses the open problems in detecting data exfiltration. The second section discusses the open problems in evaluating the impact of a data breach.

1.2.1 Detection

It is crucial to identify when a data breach is occurring in order to mitigate the damages and reduce the impact of the incident [10]. Thus, it is important for companies to have automated tools that can identify when a data breach is happening. Timely detection can reduce the impact of a breach, saving money and possibly the reputation of the company. In this thesis we focus on detecting data breaches using automated tools.

The three main sources of information that can be used to detect a data breach are: (1) network traffic, which is used by network-based intrusion detection systems (NIDS), (2) operating system activities, used by host-based intrusion detection systems (HIDS) and (3) log files, which are used by dedicated log analysis solutions. Logs usually represent high level information about application, network connections, host activities, etc. Typically, the analysis of log files allows the correlation of information from different sources, whereas NIDS and HIDS have a smaller scope. However, log analysis usually does not provide real time detection, whereas HIDS and NIDS typically do. Despite their differences, each type of information source contributes to the detection of data breaches, and they can be considered complementary.

We focus on network traffic, especially in the setting of a NIDS. Network-based solutions are cheaper to deploy than HIDS, because they can offer protection to multiple machines, independently from their operating system or architecture, and they do not affect the performance of monitored machines. More information regarding HIDS can be found in the surveys published by Axelsson [33] and Idika and Mathur [34]. Regarding log analysis for intrusion detection, we refer the reader to the survey of Zuech et al. [35].

There are two types of NIDS. A signature-based NIDS focuses on identifying byte sequences known to represent malicious traffic (i.e., signatures). Although these detection systems are precise in identifying known threats, they miss new attacks because a signature can be generated only after an attack is analyzed. This dependency on known attacks makes a signature-based NIDS unsuitable to detect APTs, since their attacks are targeted and unlikely to be observed before. Thus, we focus on the other type of detection system, namely the anomaly-based NIDS. The main idea of an anomaly-based solution is to create a model that represents normal behavior. An anomaly, and ideally an attack, is detected when traffic shows a different behavior than that described by the model. Although this type of NIDS is capable of identifying new attacks, it is often causing false alerts (i.e., false positives). Obtaining high precision is very important for anomaly-based systems, and it is one of the main obstacles to their deployment to practice [36]. Another important aspect is to design the detection technique such that it is not easily evaded by an attacker. This becomes even more challenging in case of advanced attackers, who are known to use stealthy and evasive techniques to avoid detection.

Characterization of data exfiltration. Although the aforementioned challenges are generic to the design of anomaly-based detection systems in the presence

of advanced attackers, data exfiltration detection has also its own challenges.

Data exfiltration does not have clear patterns. The attacker can arbitrarily decide how data is exfiltrated by choosing: how data is encoded, how much and where data is stored in one message, how often messages are sent, and the destination. For example, an attacker can decide to encode an entire database, divide it into chunks and exfiltrate it within hours; or exfiltrate a cryptographic key pair within the same message.

Consequently, the lack of patterns makes data exfiltration detection difficult. Many detection heuristics often rely on *predefined threat knowledge* (PTK) (i.e., observable patterns associated with a specific threat), meaning that a detection heuristic usually tries to highlight patterns that are known to be suspicious. Such predefined knowledge can be inferred by data, using *supervised* learning approaches, where a detection model is built based from malicious and benign data. Alternatively, PTK can be provided by an expert knowledgeable about the threat, and it can be used to build detection rules. However, since data exfiltration is inherently diverse depending on the attacker choices, relying on PTK is a sub optimal approach against unknown exfiltration attempts, because only exfiltration attempts similar to previous attacks can be identified. This is the main reason why we focus our efforts on anomaly-based approaches, instead of focusing on signature-based solutions.

Besides having no patterns, data exfiltration can also be performed over many different network protocols. As a consequence, different detection heuristics are needed. Each protocol transmits data differently, both in terms of syntax and semantics, and the attacker has different ways to hide its data according to the protocol. In this work we decided to focus on data exfiltration detection over the HTTP protocol, because it is a protocol commonly used in malware communication [37]–[39]. A more detailed discussion about our choice is discussed in Section 2.3.3. The focus on HTTP traffic implies that our work cannot be directly applied to encrypted channels, such as TLS. However, the solutions proposed in this work can be used on encrypted web traffic (e.g., HTTPS) in case a TLS-proxy is deployed by the company. The proxy decrypts and re-encrypts the traffic, allowing security solutions to inspect the network messages in cleartext.

Little related work approached the detection of data exfiltration over HTTP using anomaly-based techniques, thereby aiming for a more generic approach without relying on PTK [40]–[42]. These works have shown potential in identifying anomalous outbound traffic. However, these approaches generate coarse-grained detection models which are not precise, meaning that they have a high false negative rate (i.e., they often miss malicious communication). Moreover, the proposed techniques are not robust against simple evasion techniques.

Our work focuses on proposing data exfiltration detection methods that, similar to [40], [42], do not rely on PTK, but at the same time are more precise and robust against evasion attempts. We achieve a more precise and robust data exfiltration

detection method by answering two research questions. The first question is:

RQ1: How to detect data exfiltration *without* predefined threat knowledge?

The second question focuses on evasive attackers. We assume an advanced attacker is capable of adapting her communication according to the communication of the victim’s system she has compromised. Such technique would improve the stealthiness of her data exfiltration. We define this way of communication as *victim-aware adaptive* (VAA) covert channels. Hence, our second research question is:

RQ2: How can we detect victim-aware adaptive (VAA) data exfiltration?

We now briefly discuss the intuition of our proposed solutions to address both research questions. More information regarding the contributions of each solution can be found in the next section.

To address **RQ1** we propose *passive application fingerprinting* for anomaly detection. The intuition is that each software, or application, has its own characteristics in generating network traffic. Thus, by inspecting the traffic from each application installed on a machine, we can create models in the form of fingerprints, that are capable of identifying traffic generated by the specific application. Once the fingerprints for a machine are known, we can identify when malicious software is installed and starts communicating, because its traffic likely does not match with any known fingerprint from applications installed on the compromised system. In contrast with previous work, our anomaly-based detection approach introduces fine-grained models (i.e., application fingerprints), which, as we show in Chapter 3, are more precise and more resistant to evasion than the coarse-grained models used by existing work (e.g., [42]). An overview of the granularity of existing data exfiltration detection approaches is shown in Figure 1.1. To answer the research question, we first propose DECANter, a detection system based on passive application fingerprinting, which shows that our technique is effective in detecting data exfiltration. Then, we propose HEADPRINT, which uses a different method to perform application fingerprinting that overcomes the limitations of DECANter’s fingerprinting method, thereby making passive application fingerprinting more practical.

To address **RQ2** we first introduce CHAMELEON, a toolchain to generate synthetic datasets including VAA data exfiltration attacks, and we use these datasets to evaluate the effectiveness of existing detection systems against this threat. We show the inadequacy of existing detection tools, and we propose HONEYTRAFFIC, a deception-based technique to detect some specific types of VAA data exfiltration. The intuition behind this technique is to create fake network packets from each machine (i.e., each potential victim), and wait until the attacker adapts its network messages to the fake traffic. Since benign applications do not try to mimic other application traffic, duplicated fake messages can only expose attackers that have copied them. We provide estimates showing that HONEYTRAFFIC is effective and requires negligible network overhead. We discuss how HONEYTRAFFIC complements existing detection solutions, thereby making it harder for attackers to evade

Detection Model Granularity	Model Context	Approach	Literature
Coarse
Model of the whole protocol	Model HTTP traffic with TCP features.	[21]
	Model per host	Model HTTP requests with Application-layer features, for each monitored host.	[20, 22]
	Model per application, per host	Model HTTP requests as a set of distinct applications using Application-layer features, for each monitored host.	Our work
Fine			

Figure 1.1: Overview of the model granularity of existing HTTP data exfiltration detection approaches.

detection. Finally, we discuss the limitation of HONEYTRAFFIC, highlighting what types of VAA covert channels cannot be detected.

Our work focuses on the detection of data exfiltration, however there are still other open problems that our work does not address. One open problem is data exfiltration detection over encrypted channels (e.g., TLS). Another problem is the identification of what data has been leaked through the network during the exfiltration.

1.2.2 Impact Evaluation

The Mitigation phase is responsible for containing the damage of an incident, collecting the digital artifacts and analyzing them. The collected artifacts can be used to learn details about the data breach, such as the causes and the impact.

Evaluating the impact of a data breach, and therefore determining what data has leaked during a data breach, is possible by using the information collected during the Mitigation phase. There are many different types of digital artifacts that can be used, for instance log files, memory dumps, traffic captures and disk images. All these artifacts require different tools and techniques to be collected, preserved and analyzed. Memory dumps can be analyzed to get insights and visibility about the current state of a machine (e.g., running processes, data used by processes, etc.). Memory analysis has a lot of potential especially in investigating advanced attackers that use fileless malware, which does not store artifacts on disk but only operates in the system memory [43]. Analyzing disk images (e.g., hard drives, usb devices) allows for the exploration of the filesystem stored on the devices, and inspecting files to identify malicious artifacts that may have been involved in the incident. The analysis of traffic captures from infected systems allows the analyst to understand how the attacker has communicated with her infrastructure, and possibly what data was transmitted. Lastly, log analysis focuses on inspecting log files that contain high level information from different sources, such as kernel logs, authentication logs, system logs, system boot logs, application logs (e.g., firewall or web proxy), etc. The correlation of logs, which is often the result of a significant

manual effort, helps understanding the dynamics of a cyber incident. All these techniques are complementary and they are all used during the analysis of an incident.

Our work focuses on log files. Log files are important during the analysis of an incident because they contain relevant information about the status of critical parts of the system. Moreover, they are easy to generate and to analyze, because they are usually simple text files. For example, if a log would store what files have been accessed, by which user and when, an automated tool could analyze the impact of a data breach, assuming the affected systems and users are known. Although their implementation and usage seems straightforward, their security is not, and therefore the information they store is not reliable. Below we discuss the open problems related to log files. More information related to memory analysis can be found in the survey presented by Vömel and Freiling [44], and the book authored by Ligh et al. [45]. We refer the reader to the work of Soltani and Seno [46] for disk analysis, and to Pill et al. [47] regarding network analysis.

Reliability of log information after compromise. Log files contain useful information that can be used to quickly understand what has happened during a breach, thereby reducing the costs of handling the incident. Unfortunately, existing logging systems do not produce reliable log files. When an attacker compromises a machine, she can manipulate the logs created or stored by the victim. She can delete log entries, modify them, or add fake entries. Such modifications are in the best interest of the attacker because they may help her to remain undetected and remove the traces of the attack. Thus, these manipulations can severely affect the determination of data leakage. Currently, secure logging mechanisms can protect log information generated *before* the system is compromised, and not *after* the compromise [48], [49].

Consequently, incident responders inspecting these logs have no other option than considering *all* data stored on that machine to be leaked, because they cannot prove otherwise. Alternatively, they must rely on other artifacts. The main reason why the attacker can easily manipulate log files after a compromise is because the compromised machines are solely responsible for the creation of content for log files. If the attacker has complete control over the only system responsible for log generation, then it is unavoidable that logs can be manipulated. Moreover, there is no way to recognize what information in the log has been modified after the compromise. Therefore, an important challenge to determine what data has leaked is to secure logs from advanced attackers, despite the compromise.

Data Loss Prevention systems (DLPs) cannot guarantee the prevention of data leakage in case advanced attackers are the perpetrators. Furthermore, once data is leaked, they cannot quantify what was stolen. Another way to prevent data leakage from happening, and therefore determine that *nothing* has leaked, is to encrypt the data and perform computations over it in the encrypted domain. Examples of these techniques are fully homomorphic encryption [50] and secure multiparty computation [51]. However, these techniques can be applied only in very limited settings, because they are computationally efficient only for a restricted set of simple data operations.

A promising approach to determine data leakage is to use data provenance systems. These solutions track all the historical information of a data item: who had access to it, when, what was operation was performed, etc. Bates et al. proposed the Linux Provenance Module (LPM) [52], a Linux kernel module that keeps track of the provenance of each data item, and can be used to investigate data leakage. Unfortunately, despite being a solution dedicated for a single operating system, LPM can be secured against advanced attackers only by using trusted hardware. Such solution is invasive and expensive because it requires important changes to the existing infrastructure.

In our work we focus on proposing a new way to secure information from advanced attackers, such that we can reliably determine what has leaked after a data breach. Although the threat model is similar to the one discussed in [52], our work does not rely on trusted hardware assumptions. Thus, our third research question is:

RQ3: How to reliably determine what data was exfiltrated despite the host *being compromised*?

To address **RQ3** we propose a distributed system where machines determine in a coordinated fashion what data should be accessed, when and by whom, and each node stores this information in a replicated secure log. Essentially, we shift the responsibility of generating a log, and accessing the data, from a single machine to a group of machines, making it a group effort. Consequently, even if the attacker is capable of compromising a subset of machines and tamper with their logs, there still exists a majority of machines having an intact log. The log contains the information of what data was accessed by compromised nodes, allowing us to determine an upper bound of what has leaked. More details about the contributions of this work can be found in the next section.

Our work focuses on providing a solution that is able to provide secure logging despite the system being compromised, however, it does not address the problem of providing legacy compliance, with respect to existing logging systems.

1.3 Thesis Overview and Contributions

Figure 1.2 provides an overview of this thesis and the contributions of our work. We start in Chapter 2 by presenting the background information necessary to understand the remainder of the thesis. Moreover, we discuss the state of the art related to data exfiltration detection in NIDS. DECANter is introduced in Chapter 3, a system that uses passive application fingerprinting for identifying data exfiltration attempts. We improve on DECANter with HEADPRINT in Chapter 4, where we present a fully automated application fingerprinting technique for anomaly detection. In Chapter 5 we introduce CHAMELEON, a toolchain that generates adaptive data exfiltration traffic, and we investigate the effectiveness of adaptive data exfiltration attacks against state-of-the-art detection solutions. We

Figure 1.2: Thesis outline.

introduce HONEYTRAFFIC in Chapter 6, a new deception-based detection technique to detect specific types of VAA data exfiltration. In Chapter 7 we present a secure distributed logging system that can be used to determine the impact of a data breach. Finally, in Chapter 8 we present the conclusions and future directions. We now discuss in more details the contributions of each chapter.

Chapter 3 - Detecting Data Exfiltration via Application Fingerprinting

We present a new methodology for anomaly detection called *passive application fingerprinting*. We propose DECANter, a system that leverages such methodology to detect data exfiltration attempts, without the need of predefined threat knowledge. We show DECANter is capable of detecting communications of malware known to be used in APTs. Furthermore, our proposed technique has shown better detection, lower false alert rate and more evasion resistance than DUMONT [42], which was, at the time, the state-of-the-art data exfiltration detection mechanism. DECANter has appeared in ACSAC 2017 [1].

Chapter 4 - Enhancing Passive Application Fingerprinting

Although DECANter has shown good detection performance, its fingerprinting method relies on a small set of fixed and hand-picked features, which has some performance limitations in correctly identifying some application traffic. DECANter overcomes these limitations by assuming the presence human operator who manually and continuously monitors and updates the fingerprints. Based on these observations, we present HEADPRINT, a new method that generates fingerprints by automatically identifying relevant traffic characteristics for each application. A real world validation shows that HEADPRINT outperforms DECANter both in terms of accuracy, to correctly identify application traffic, and resilience to software updates. Additionally, we show that HEADPRINT is still capable of identifying potential data exfiltration attempts, while generating significantly fewer false alarms than

DECANTeR. At the moment of the writing of this thesis, the paper was still in submission at a peer-reviewed conference [2].

Chapter 5 - Victim-Aware Adaptive Data Exfiltration A victim-aware adaptive (VAA) attacker is an attacker that mimics her victim’s communication in order to avoid detection from a security monitor. We investigate this type of threat by introducing CHAMELEON, a toolchain that allows researchers to generate VAA data exfiltration traffic. Using the datasets generated by CHAMELEON, we perform a comparative analysis of three state-of-the-art data exfiltration detection methodologies: HED [53], which is based on predefined threat knowledge, and DECANTeR and DUMONT [42] which are not based on predefined threat knowledge. The comparison shows that none of these techniques is effective in detecting adaptive data exfiltration. Our work highlights the intrinsic limitations of network-based heuristics against VAA attackers. This work will appear in SecureComm 2019 [3].

Chapter 6 - Detecting Adaptive Data Exfiltration via HONEYTRAFFIC We introduce HONEYTRAFFIC, the first detection solution capable of detecting specific types of VAA data exfiltration attempts. More specifically, the types represented by CHAMELEON. HONEYTRAFFIC is a signature-based network security solutions, which also relies on a client installed on each monitored host. We discuss how HONEYTRAFFIC can be easily integrated within existing network security solutions. Finally, we give estimates of its performance, which show a negligible network overhead and a negligible false alert rate. This work is part of the same paper discussing CHAMELEON, and, therefore it will appear in SecureComm 2019 [3].

Chapter 7 - Evaluating Data Exfiltration by Determining What Has Leaked We present the architecture of a distributed system, where machines coordinate each other to determine which machine should access what data and when. Moreover, machines are also responsible to store this information in a replicated log. Our architecture does not rely on trusted hardware, and we show it is capable of protecting logs from attacker manipulation as long as the attacker does not compromise at least one third of the total number of machines. For the concrete application of client-server authentication, which is relevant in the setting of data breaches, we show that our approach is feasible in practice, it can be integrated with existing services, and it can precisely identify what has leaked. This work has appeared in ACSAC 2016 [4].

This chapter introduces the research questions and the contributions of our work. In the next chapter we describe the background knowledge to understand our work on the detection of data exfiltration.

Chapter 2

Intrusion Detection

An Intrusion Detection System (IDS) is a tool that monitors events occurring in information systems or networks, and analyzes such events to identify potential security incidents. The first work describing the idea of IDS appeared in 1980 with the seminal work of Andersen [54], while the first real-time IDS model was discussed few years later by Denning and Neumann [55], [56]. The IDS concept kept evolving since then, and it is now a fundamental tool for companies to protect their infrastructures.

There are two main types of IDS. A *signature-based* IDS, also referred as misuse-based IDS [57], uses knowledge about malicious behavior and generates models of it. Traditionally, models are manually created by security experts for specific malicious samples. However, these models can also be generalized by analyzing similarities between clusters of malicious software [58]–[62]. The IDS takes these models as input, monitors events of a system or a network, and it verifies whether these events are similar to any known model. If similarities are found, an alert is triggered to the security operator. An *anomaly-based* IDS takes the opposite approach. Instead of relying on known malicious characteristics (e.g., specific byte sequences of malware communication), it uses only knowledge of benign behavior to generate models. In other words, the IDS learns a representation of the normal behavior of a system or network, by analyzing the characteristics of observed events. An alert is triggered to an operator when an event is not similar to learned models (i.e., anomalous). However, in some specific scenarios (e.g., SCADA networks, or building automation systems) model construction can also be supported by specifications describing the modus operandi of a system. For example, technical documentation could describe how exactly a specific device should normally behave [63]. An IDS that leverages this type of knowledge, is known as a *specification-based* IDS [64], and it can be considered a sub-category of anomaly-based IDS.

Both approaches have advantages and disadvantages. Signature-based approaches are easier to implement than anomaly-based approaches, and they are also more precise because they trigger fewer false alerts [65]. These advantages explain why signature-based are more widely used by companies compared to anomaly-based

Figure 2.1: Overview of an anomaly-based IDS's components. Red blocks represent the components we focus on in this work.

solutions. However, signature-based techniques are limited by existing knowledge of malicious behavior, thus such a technique is not suitable to identify unknown threats. On the other hand, anomaly-based techniques are harder to implement and learning normal behavior is not trivial in environments where data is heterogeneous and may change over time. Thus, an anomaly-based IDS usually generates more false alerts than traditional signature-based approaches. Nevertheless, anomaly-based models are not limited by existing knowledge about malicious behavior, thereby making this approach more suitable for detecting unknown malicious behaviors.

As we mentioned in the introduction, signature-based approaches are out of scope for this thesis, since they are intrinsically not capable of identifying unknown threats. In the next section we discuss the core components of an anomaly-based IDS, how IDS performance are evaluated, and, finally, the state of the art of anomaly-based research with regards to data exfiltration detection.

2.1 Anomaly-based Intrusion Detection

An anomaly-based intrusion detection system has four main components: *monitoring sensors*, *detection model generation*, *anomaly detection*, and an *alert processing*. An overview of the components of an anomaly-based IDS is shown in Figure 2.1. Sensors are responsible for monitoring an information system and for reporting events to other IDS components. The model generation component, sometimes referred to as learning engine [66], is responsible for the generation of models representing the normal behavior of the system. The anomaly detection module evaluates whether the events received from a sensors deviate from the normal behavior represented by the learned models. When an anomaly is identified, an alert is triggered and forwarded to the last module. The alert processing module elaborates the alerts and present the security events to the security operator. In case an IDS alerts a security operator, the IDS is considered to be passive. If the IDS enforces security policies to isolate and mitigate threats automatically, then the IDS is considered to be active. An IDS can also be both active and passive by both

Figure 2.2: Hierarchical overview of the topic of IDS. Colored blocks and dashed lines highlight the focus of this thesis within the topic of IDS.

warning the operator and taking actions on his behalf.

There exist two main types of anomaly-based IDS: host-based IDS (HIDS) and network-based (NIDS). The difference between these two type of IDS is the type of data they process. On the one hand, a HIDS analyzes the internal behavior of a machine. A HIDS collects information from the running operating system, in order to learn how users and legitimate software behave on the host. Finally, a HIDS uses detection metrics to compare the monitored behavior with the learned models. Examples of information used by a HIDS are system calls [66]–[68], storage accesses [69], and file system accesses [70]. On the other hand, a NIDS analyzes the communication of a machine. A NIDS collects information from the observed network traffic sent, and received, in order to learn how machines normally communicate. Lastly, a NIDS uses detection metrics to identify anomalous network behavior deviating from the learned models. The information used by a NIDS is represented by two macro categories: flow-based and payload-based. Flow-based approaches extract meta-information about network connections, such as IP addresses, ports, transport protocol and the number of transmitted bytes [71]. Payload-based approaches, also known as Deep Packet Inspection (i.e., DPI-based approaches), focus on the information contained within the payload of network messages [36]. As mentioned in the introduction, this thesis focuses on anomaly based NIDS using DPI-based approaches. Figure 2.2 illustrates the focus of the thesis.

2.1.1 Network Sensors

Network sensors play a crucial role for a NIDS, because they are the NIDS main source of information. Thus, sensors must be able to correctly parse the information observed in the network. The performance of sensors is usually represented by the amount of correctly processed network messages, and it is determined by different factors: how much detail should be extracted from a connection, the computational capabilities of the sensor (i.e., hardware specifications), and the bandwidth of the monitored network. These factors lead to two main type of NIDS sensors: flow-based and payload-based. The latter is also known as Deep Packet Inspection (DPI) based.

Flow-based technologies (e.g., Netflow and IPFIX [72]) focus on extracting high-level information from a connection (e.g., IP addresses, ports), focusing mainly on information up to the transport-layer. By not parsing application layer information these sensors can process more information than DPI based solutions. Consequently, with cheaper hardware they can monitor larger networks. However, the lack of detailed information limits the detection capabilities. On the other hand, DPI-based solutions focus on extracting as much information as possible from network traffic in order to have better visibility about the status of the network. Such a level of detail gives more opportunity to detect anomalous and malicious behavior, but it also affects the costs of deployment.

The tools available today are often open source and highly customizable, thereby leading to hybrid approaches that lie somewhere between flow-based and DPI-based. For example, there exist flow-based techniques that can be integrated with parsers for specific protocols, in order to extract more information about the network. For instance, a flow on port 80 can be enriched with the strings representing the `URI` and the `Host` headers for HTTP. These additions enable further detection capabilities. On the other hand, DPI-based solutions may limit the extraction of detailed information to a subset of protocols, in order to reduce the computational requirements while providing a detailed view for protocols where details are relevant for detection (e.g., HTTP, DNS, TLS). For other less critical protocols (e.g., ICMP, IRC), the DPI-based solutions may decide to extract only connection metadata.

Encrypted Channels and DPI-based Approaches The main goal of encrypted channels, such as TLS, is to protect the confidentiality and integrity of network communications. These communication channels are becoming more and more popular and they became an important challenge for a DPI-based NIDS. The detection capabilities of a DPI-based NIDS are severely limited by encryption, because network content cannot be inspected in order to identify malicious, or anomalous, patterns. Having no access to the content of network communication, does not only affect the performance of detection tools, but also reduces the visibility of what is happening in the network.

Today, the solution many enterprises are adopting is to deploy a TLS-proxy in their network. TLS-proxies are dedicated machines that intercept encrypted traffic, decrypt it, such that it can be inspected, and re-encrypt it again towards

the correct destination. In other words, TLS-proxies perform a (legal) man-in-the-middle attack. Currently, these proxies are the only solution that allows DPI-based detection systems to have access to encrypted traffic. Although this is the current solution, it is not ideal, because these machines are not affordable for many companies, and the security of these proxies is questionable at the moment [73].

An alternative solution has been proposed in academic literature, where DPI-based solutions could access encrypted data while never accessing the data in plaintext [74]. However, such alternative solution is not practical because it requires one of the endpoints (i.e., client or server) to be trustworthy, while in the setting of data exfiltration, both communicating endpoints are malicious. The only other practical (and cheaper) alternative is to rely on flow-based solutions. Flow-based approaches can still be used for detection because the connection metadata is available for analysis. However, choosing flow-based approaches comes at the costs of lower detection capabilities.

Security Risks for Network Sensors The role of sensors is to interpret and extract information from network traffic, so that the NIDS can analyze the sensor data for potential anomalies. Therefore, an attacker may try to hide her presence from the NIDS by attacking the sensor, and avoid that her communications are analyzed by the NIDS. This type of attack often focuses on the reconstruction of TCP streams, which is an operation performed by all sensors, both flow-based and DPI-based, in order to understand the connection between two IP addresses. Examples of this type of attacks are memory buffer attacks [75] and fragmentation attacks [13]. In this thesis we assume the sensor is resistant against such type of attacks [75], and, therefore, the sensor correctly extracts the information from the network. This assumption is needed to guarantee that the data generated by the attacker is analyzed by the NIDS. This assumption is often implicit in research NIDS.

2.1.2 Detection Model Generation

The most important phase in anomaly-based NIDS is the detection model generation (see Figure 2.1), which creates the models representing the characteristics of normal traffic. These models are then used by the anomaly detection module, which compares them with the observed traffic in order to identify anomalies.

An anomaly-based detection system is usually designed to detect a specific anomalous behavior, which is often associated with the behavior of well known threats (e.g., botnet, port scan, DDos). Hence, it is important that the design of each detection model is tailored to each specific use-case. A detection model is built in two separate steps: *feature selection* and *modeling technique choice*.

Feature Selection

During *feature selection*, the model designer identifies what network characteristics (i.e., features) can be useful to detect the specified anomalous behavior. This process usually entails *data analysis* and *expert knowledge*.

Data analysis is often used as an exploration process, where the designer analyzes network data, possibly containing both normal and anomalous behavior. This analysis process is important because it helps the designer to identify what features are relevant to recognize the anomalies. The designer can also use in this process techniques such as Principal Component Analysis (PCA) to automate the discovery of relevant features. However, data analysis is limited by the quality of the underlying dataset. If the dataset is not representative enough for both normal and anomalous behavior, the feature selection may not be as representative in a real network.

Consequently, expert knowledge becomes important when choosing the features. For example, expert knowledge may help a designer to discard certain features that seemed stable during data analysis, but may not be in a real network, thereby improving the future detection performance in real settings. Most importantly, expert knowledge helps formulating more accurate assumptions about the attacker capabilities. These formulations may also help identifying those features that are harder to evade for an attacker. A detection system that relies on features that are harder to evade is, overall, a better detection system.

Choosing a Modeling Technique

Once the set of network features has been chosen, the designer chooses what technique fits best to model the selected features. This choice is affected by the type of features (e.g., numerical, categorical, ordinal), the number of features, the computational costs of a modeling technique, the data available to be analyzed, and so on. There exist two main techniques to learn models: *machine learning based* and *knowledge based*.

Machine Learning based Machine learning based techniques analyze normal traffic data and they infer the traffic characteristics according to the chosen features. The analyzed data is usually labeled and it represents a single class (e.g., benign traffic). The output of these techniques is a model that describes the normal behavior. Models are built during a phase known as *training* phase. These machine learning techniques are known as *semi-supervised learning* techniques.

There exists another type of machine learning techniques that can be used to perform anomaly detection: *unsupervised learning* techniques. These techniques do not require any training phase or labeled data, and they can identify anomalous data points within a given dataset. However, unsupervised techniques are typically used for data mining purposes, where a user wants to identify anomalies within a dataset. Therefore, these techniques are mostly used in offline setting (e.g., to process batches of historical data). Unsupervised algorithms are rarely used in anomaly-based NIDS. In the rest of this thesis, we focus on semi-supervised approaches. Furthermore, we exclude *supervised* learning techniques, because models are trained relying on both malicious and benign data. Therefore, supervised learning techniques are not considered anomaly-based approaches.

The main advantage of machine learning techniques is that the algorithms can analyze large quantities of data, both in terms of data items and features, and

infer distinguishing patterns to identify anomalies. The main disadvantage is that networks often contain heterogeneous values and change over time. Therefore, it is difficult to generate a training dataset that is representative of the normal network traffic. Consequently, the models may not be representative enough for the normal traffic, and they can trigger large numbers of false alerts. Furthermore, it is not easy to interpret why certain events are triggered as anomalous by machine learning algorithms. Sommer and Paxson discuss, in a seminal paper [76], the practical limitations of using machine learning technique in the context of NIDS.

Knowledge based Knowledge based techniques are most often used in practice. Experts leverage their domain knowledge to define a set of heuristics (i.e., a model) which is used to identify anomalous traffic, according to the chosen features. These heuristics often rely on thresholds, which are set to specific values defined by the experts, to distinguish behaviors. For example, an anomaly-based port scan heuristic can be the following: if we observe a host sending more than 15 messages, on different destination ports, to another host in the internal network, then we consider this event as a port scan attempt. Such a heuristic implies that benign traffic (i.e., not a port scan attempt) sends fewer than 15 messages, on different destination ports, to a host on the internal network, and traffic towards non-internal IP addresses is not interesting for port scan detection.

The upside of these systems is that they are often computationally efficient and easy to interpret, allowing the system to provide a better reasoning about why a specific anomalous event has been triggered. The downsides of knowledge based heuristics are: 1) sub-optimal threshold values, often these values are set based on personal experience and they are not based on a thorough analysis; 2) sub-optimal feature selection, meaning that the experts generate heuristics based only on a small set of features, thus they may miss other important traffic characteristics.

Building the Detection Model

After choosing the technique to use, a machine learning based detection model can be built as follows: i) the network information transmitted by the sensor is transformed into feature vectors, representing the information chosen by the designer; ii) the feature vectors are collected, and form together the *training dataset*; iii) the model is created by analyzing the training dataset. In case of knowledge based approaches, the experts manually define the heuristics (i.e., the model) according to the selected features, their experience and the data they have manually analyzed. It is common to use both approaches in order to build detection models.

Once the models have been described, the designer needs to decide how to evaluate the model, meaning that he needs to choose a function that can compare the existing model with the newly observed data. Such a decision function should measure how similar the data is to the model, in terms of distance or other similarity metrics, and label the data as anomalous if it deviates too much from the model. Both the model and the decision function represent the detection model.

2.1.3 Anomaly Detection

Once the model is ready, the system can start detecting anomalous behavior. The sensors extract and process the network data, and they send it to the anomaly detectors. These detectors evaluate the extracted sensor data against the trained models. In case the new events deviates from the known model, then an anomalous event is triggered and forwarded to the alert processing module.

It is important to notice that network data is not only processed by the sensor, but it may also be processed by the detector itself. For example, in case of machine learning, events must be transformed in the same type of feature vectors that have been used to build the model, otherwise the event cannot be properly evaluated. Similarly, data must be processed if knowledge based heuristics analyze specific features and not only raw data.

2.1.4 Alert Processing

Before alerts are finally shown to the security operator, alerts are often processed in order to provide more valuable insights about the security events. Valeur et al. [77] provide a detailed description of this process. The main objectives of these procedures are: 1) reduce the number of events in order to avoid redundant information (i.e., *alert fusion*); 2) reduce the number of false alerts by verifying, according to predefined procedures, whether the alerts likely represent benign behavior or unsuccessful attacks (i.e., *alert verification*); 3) reconstruct an attack scenario involving one or more machines by correlating different alerts (i.e., *thread reconstruction and multi-step correlation*); 4) assign a severity score to the security events such that operators can focus on the most impactful attacks (i.e., *impact analysis and prioritization*); and 5) visualize the information to the end user, the security operator. All these intermediate steps are important to improve the usability of an IDS.

2.2 Performance Evaluation

When an IDS evaluates an event (e.g., a connection), the result of the evaluation falls in one of the following categories:

- *true positive* (TP): the IDS triggered an alert and the event was malicious;
- *true negative* (TN): the IDS did not trigger an alert and the event was not malicious;
- *false positive* (FP): the IDS triggered an alert and the event was not malicious;
- *false negative* (FN): the IDS did not trigger an alert and the event was malicious.

From the point of view of IDS performance, FP are particularly relevant because they represents false alarms, and they strongly affect the usability of an IDS.

A high False Positive Rate (FPR) can make the IDS not practical to use [36], [76]. Other metrics relevant for the evaluation of IDS performance are: *precision*, which describes the percentage of true alerts over false alerts; *recall*, which describes the percentage of detected malicious events over the total number of malicious events; and *accuracy*, which is an overall metric to describe the classification performance. These metrics are computed as follows:

- $FPR = FP / (FP + TN)$
- $Precision = TP / (FP + TP)$
- $Recall = TP / (TP + FN)$
- $Accuracy = (TP + TN) / (TP + TN + FN + FP)$

2.3 Data Exfiltration Detection

Malicious software requires network communication in order to perform many of its functionalities [37]. For instance, bots receive instructions from Command and Control (C&C) servers [78], ransomware downloads encryption keys from remote locations [79], and information stealers exfiltrate data to external servers [80]. Malware activities last as long as security monitoring products (e.g., NIDS) do not identify the suspicious behavior, and until operators are able to clean the compromised machine from the infection. Consequently, attackers implement techniques to evade network monitoring tools to pursue their malicious activities for a longer period of time.

In this thesis we focus on *data exfiltration*, which is defined as an unauthorized transfer of data from a network to a location controlled by the attacker [14]. In this setting, the attacker wants to hide the transfer of information from the network security monitor, in order to successfully finish the exfiltration.

In the next sections we discuss how attackers hide their communication channels, and what is the current state of the art for data exfiltration detection techniques.

2.3.1 Network Covert Channels

Attackers can hide their communication from network security tools by using network covert channels. A *network covert channel* is a communication channel over the network that violates security policies by using network protocols in ways which they were not designed for. The overall idea is to hide the presence of a malicious communication channel (e.g., a C&C protocol) within an existing protocol, such that a security monitor does not notice the presence of malicious communications. An example of a network covert channel is to hide encrypted messages (e.g., commands) in the subdomains of the DNS Query Domain Name [81]. In this example, the malware violates the security of the network by sending and receiving commands for malicious purposes. Moreover, such communication channel abuses the DNS protocol in order to hide its presence from a security monitor. Network

covert channels are commonly used to hide from a security monitor that is capable of inspecting the content of the messages.

There are two types of network covert channels [82]: *covert storage channels*, where the secret data to be transmitted is hidden within network messages; and *covert timing channels*, where the transmitted secret data is represented by the delay between messages. In this work we only focus on the former type, because it can be used to exfiltrate larger quantities of data and it is more popular than covert timing channels.

The main principle of covert storage channels is to embed the data within elements of a network message that are redundant or uncertain [83]. In fact, network security monitors are not able to determine whether these elements contain malicious data or not. The size and quantity of these elements define the bandwidth of the covert channel. Bandwidth affects the quality of a covert channel because more data can be transmitted, thereby allowing faster communications. Handel and Sanford [83] give examples of such elements across the entire OSI Network Model. After Handel and Sandford, many other researchers discussed possible elements in network protocols that could contain data for a covert channel. Most of these techniques have been summarized in several survey papers [82], [84], [85].

There are several examples of malware using different network covert channels techniques to avoid detection: Stegoloader [15] hides information using steganographic techniques in images, FAKEM RAT [16] mimic the traffic of known applications, while FrameworkPOS uses DNS tunneling to exfiltrate data [17]. Cabaj et al. [86] discuss examples of hiding techniques deployed by malware observed in the wild.

Regarding HTTP-based covert channels for data exfiltration, malware can use different methods to exfiltrate data. Figure 2.3 shows how a malware exfiltrates encoded data using GET parameters. Figure 2.4 shows malware mimicking a known application and exfiltrating encoded data over a custom HTTP header of a GET request. Figure 2.5 shows a malware POST request exfiltrating plaintext data from the compromised system. Finally, Figure 2.6 shows a malware POST request transmitting encrypted (or binary) data. These examples represent some of the techniques that attackers can use to exfiltrate data over HTTP. Note that all these samples are part of a dataset we collected in this work and they are publicly available ¹.

2.3.2 Anomaly-based Data Exfiltration Detection

Data exfiltration detection can be performed using different approaches. As mentioned by Wendzel et al. [82], there is a large number of covert channel techniques. Consequently, it is difficult to mention all the detection approaches proposed in literature. Hence, in this section we focus on those anomaly-based data exfiltration detection approaches that are more relevant within the context of this thesis.

¹[https://scs.ewi.utwente.nl/downloads/show,DataExfiltrationMalware\(DEM\)/](https://scs.ewi.utwente.nl/downloads/show,DataExfiltrationMalware(DEM)/)

```

GET /news.php?
m&Auth=80951A85&Session=11E668C0709577A5&DataID=1&FamilyID=5FB5D5B6AD144B47BFF1CB086129A255&BranchID=35&VolumeID=
200&User=4285927&Query=j1b2fvdznrrpyttof5j2rv3lhzcfoquqypop5lqfbucxxen2uzo5e3cvqvltxg3lon1pkkjy4trgkujn0g2sp0x5zy
ij3ys1qpudou5cfnu1fubuied2b2gidjpxdzy4rh14tbxrsx2yidh15e0ggt5m1i00a3umr3c3lika1fmbbg4f1qog0j20eoki1ofvs0jaetzc1
w2qj13k4ce1hu1aeb050g0kqur1hophyqw531jxoh5w5s1fre2bdj1b2fvdznrrpyttof5j2rv3lhzcfoquqypop5lqfbucxxen2uzo5e3cvqvl
txg3lon1pkkjy4trgkujn0g2sp0x5zyij3ys1qpudou5cfnu1fubuied2b2gidjpxdzy4rh14tbxrsx2yidh15e0ggt5m1i00a3umr3c3lika1f
mbbg4f1qog0j20eoki1ofvs0jaetzc1w2qj13k4ce1hu1aeb050g0kqur1hophyqw531jxoh5w5s1fre2bdj1b2fvdznrrpyttof5j2rv3lhzcfo
uqypop5lqfbucxxen2uzo5e3cvqvltxg3lon1pkkjy4trgkujn0g2sp0x5zyij3ys1qpudou5cfnu1fubuied2b2gidjpxdzy4rh14tbxrsx2yidh
15e0ggt5m1i00a3umr3c3lika1fmbbg4f1qog0j20eoki1ofvs0jaetzc1w2qj13k4ce1hu1aeb050g0kqur1hophyqw531jxoh5w5s1fre2bd
j1b2fvdznrrpyttof5j2rv3lhzcfoquqypop5lqfbucxxen2uzo5e3cvqvltxg3lon1pkkjy4trgkujn0g2sp0x5zyij3ys1qpudou5cfnu1fubu
ied2b2gidjpxdzy4rh14tbxrsx2yidh15e0ggt5m1i00a3umr3c3lika1fmbbg4f1qog0j20eoki1ofvs0jaetzc1w2qj13k4ce1hu1aeb050g0
kqur1hophyqw531jxoh5w5s1fre2bdj1b2fvdznrrpyttof5j2rv3lhzcfoquqypop5lqfbucxxen2uzo5e3cvqvltxg3lon1pkkjy4trgkujn0g
2sp0x5zyij3ys1qpudou5cfnu1fubuied2b2gidjpxdzy4rh14tbxrsx2yidh15e0ggt5m1i00a3umr3c3lika1fmbbg4f1qog0j20eoki1ofvs0
jaetzc1w2qj13k4ce1hu1aeb050g0kqur1hophyqw531jxoh5w5s1fre2bdj1b2fvdznrrpyttof5j2rv3lhzcfoquqypop5lqfbucxxen2uzo5e
3cvqvltxg3lon1pkkjy4trgkujn0g2sp0x5zyij3ys1qpudou5cfnu1fubuied2b2gidjpxdzy4rh14tbxrsx2yidh15e0ggt5m1i00a3umr3c3l
ika1fmbbg4f1qog0j20eoki1ofvs0jaetzc1w2qj13k4ce1hu1aeb050g0kqur1hophyqw531jxoh5w5s1fre2bdj1b2fvdznrrpyttof5j2rv3
lhzcfoquqypop5lqfbucxxen2uzo5e3cvqvltxg3lon1pkkjy4trgkujn0g2sp0x5zyij3ys1qpudou5cfnu1fubuied2b2gidjpxdzy4rh14tbx
sx2yidh15e0ggt5m1i00a3umr3c3lika1fmbbg4f1qog0j20eoki1ofvs0jaetzc1w2qj13k4ce1hu1aeb050g0kqur1hophyqw531jxoh5w5s
1fre2bdj1b2fvdznrrpyttof5j2rv3lhzcfoquqypop5lqfbucxxen2uzo5e3cvqvltxg3lon1pkkjy4trgkujn0g2sp0x5zyij3ys1qpudou5cf
nu1fubuied2b2gidjpxdzy4rh14tbxrsx2yidh15e0ggt5m1i00a3umr3c3lika1fmbbg4f1qog0j20eoki1ofvs0jaetzc1w2qj13k4ce1hu1a
eb050g0kqur1hophyqw531jxoh5w5s1fre2bd HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR
3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E)
Host: 199.231.188.109
Connection: Keep-Alive

```

Figure 2.3: CosmicDuke Malware HTTP Exfiltration Attempt

```

GET /pki/mscorp/crl/msitwww2.crl HTTP/1.1
Check:
6047ee556f20617ef5313237c244fe9b5ab203d04d46dd0935ad2535ef6401ac8f54498b9daf5232e004ef1dbda7551fed61ef38404a3e23
e17b8e59751a8ab912317db156ddbe33193b74e6a88d6ebaac1d238191c8da2411e349517538fbcadab3a12527723bb07111111db156acebcb
c5864cac28817a4550e8d65159ec2a88ebc3231bd9c98bd05f6c9d3c09658ad38e1af9fb9707ad20efea9720ad2d015458766dd4d673a5
d74a75539d6ee0993257ef207298d088d4a576863aeadd2c5d366bf3e3d77fce32143d01adb876ad775c6a4413246ceead4ee7e1047d52e2
f3729805e033809f967954551e58b71ac3414dc3a4b2699460d9ba360c56141badbd88dda099a46c073f09dee3525d1c06dc6dae6f915bd
a6f52f9379be1d4a2680c6fe82d8dd64236e3a183a33d2b63e9afb6d8d857c8ae3ee3a64f1758fde304ebf81b886f387962827cecfb46f08b
4663b79dedf6fe7a516b86b6cfb6db9e1a8c217c5ea886c19f81ff59c49f9fc24859e70c18ce6bf1184884e5e2c0085ecf009dedbfeff7d43
c4a8f2b682d5d6679f75fc7e1f3961c6f54583c067490c47fba36ee9e7de556712d1ea7285431ca44e23debc15c7616af16ce13a3421f0894
49cae5ca0a90e8cef3845b27180cd1756959f3c34d30715b4e2c90846efc3ffa673cddb3ef917210bb27382f8d5bed50296e96ff76d1d584ff
7437597949aa10d31a2d2569c966d3e5488227a6ea5b434d4fa01392b3385e56e5d10ff09365ee216021c87f92b809cf927cc5b3f4fe0f
d607b235d686e51701c986d3f63e0c91dda392eb9f88cd25e385b5ad4e3d7228b9e557cb1f2cd1c5d21246558490ce9363d2d8f6ae8cfe
9c816d775b436ee0c59350a92083296520243615304ba918a6f55e5f2c38058c73a35f8f3ac245c9970ac4a6ebfa005fc25df7c26e4558ef1
07a0c13ac5553410cbe0de2e4fe0522e37c460efe5196254c1b9a6ef76906c3e34bd7356f4fcbcd753c18bf076e8838d1fc048c7d6a7a2f
827b943b69cce17c2b07e8a477d6a8820b8cdd33044211e50f0281a0aeb2a592358f915a581e8ae3c6e33fd7263c96678215617b2d73f5
1da5c6b4d3ba1837a8d2106f75a430b056e66aed050e870927f776d6d30e96757979c76c64441037797b1badd9450989e57f357de946fe
fb899c27a64b8420b3bba0e69c458ec55b4aeb62d95542b01b189d451a3410cb65bec5a2a1761ec09c818668d1289febcbecaa4130463590
4fb6aac3a1e50da490e7dea081a0f802c59acab2a8ae5331d35b5d5c4f8289a73d930ff16b246d
User-Agent: Microsoft-CryptoAPI/6.3
Host: 5.135.156.80
Cache-Control: no-cache

```

Figure 2.4: Ursnif Malware HTTP Exfiltration Attempt

```

MSMQ,.....d38cb775ae24de29354b1effb7d3890fPOST /external/update HTTP/1.1
Accept: text/plain
Content-Type: application/octet-stream
Ex-TagID: d38cb775ae24de29354b1effb7d3890f
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
Host: web4solution.net
Content-Length: 4481
Cache-Control: no-cache

MSMQ.....MSMQ.....MSMQY.....MSMQ.....MSMQ.....smss.exe(MSMQ).....
\SystemRoot\System32\smss.exe(MSMQ).....MSMQ.....L.....MSMQ.....csrss.exe(MSMQ).....C:
\Windows\System32\csrss.exe(MSMQ).....MSMQ.....[.....MSMQ.....wininit.exe(MSMQ).....C:
\Windows\System32\wininit.exe(MSMQ).....MSMQ.....MSMQ.....csrss.exe(MSMQ).....C:
\Windows\System32\csrss.exe(MSMQ).....MSMQ.....MSMQ.....winlogon.exe(MSMQ).....C:
\Windows\System32\winlogon.exe(MSMQ).....MSMQ.....MSMQ.....services.exe(MSMQ).....C:
\Windows\System32\services.exe(MSMQ).....MSMQ.....MSMQ.....lsass.exe(MSMQ).....C:
\Windows\System32\lsass.exe(MSMQ).....MSMQ.....MSMQ.....lsm.exe(MSMQ).....C:

```

Figure 2.5: Shakti Malware HTTP Exfiltration Attempt


```

POST /ze-pi/gate.php HTTP/1.1
Accept: */*
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E)
Host: dhfgfgshds.top
Content-Length: 5204
Connection: Keep-Alive
Cache-Control: no-cache

*k.N.....f...X.T.M.'=.Er.....?..N.*<xa.....U.%.B...r.0uJ.....s:#...'%.dw...;.\.Y.[...
8..Q.....qVw...u..q.i9..f..7.....b.....57Ri...I.+c.%,.,].&k...C...<.&.....l![.?....'h.$.....~..
[.A%.....W.....
..+D.h74.wm...su...9...L.....uP.#,...k...v...B.W..D.8[.q.....]...0..E....#...C...0'f...b..&.<.4....{.....8....
$.*r.....;cZV../.Q..L#.T*...%.ve.I.}i..5M.7...!@jd....rTYr.....6G.h.....K.....X.....Yf.....mHg.6f...
5..I...A8...R.6...H.5U.....UN...>....R.G

.b]]....Z....0.....is0.....~.F....F(y.....-

```

Figure 2.6: Pony Malware HTTP Exfiltration Attempt

Since network covert channel are often developed for specific protocols, we organize the rest of this section by categorizing detection techniques per protocol.

TCP/IP

Hammouda et al. [87] propose different detection rules to identify covert channels embedding information in distinct parts of the TCP and IP protocols, for example they address “ISN” and “Urgent Pointer” fields for TCP, and the “Type of Service” and “IP identification” for the IP protocol. Zhai et al. [88] propose a Markov model that characterize the normal behavior of TCP flags for different application layer protocols. The authors show that these models can be used to identify cover channels abusing the TCP flag field. Zhao and Shi [89] propose a new statistical model to identify covert channels over the TCP Initial Sequence Number (ISN). Their technique requires little data representing benign TCP ISN sequences compared with previous work, and it also has a much lower computational complexity than other existing solutions. Murdoch and Lewis [90] give an overview of the opportunities to hide data within TCP/IP headers. The authors propose distinct rules to identify these type of covert channels. Moreover, they also show that naïve approaches to hide data can be easily distinguished because they significantly modify header values compared with the header values in normal traffic.

DNS

Paxson et al. [91] develop a procedure to detect DNS domains that are associated to suspicious outbound query streams. DNS domains are flagged as anomalous in case they are associated with a large amount of outbound traffic. This procedure is composed of different filtering stages and it relies on lossless compression techniques to precisely measure the amount of information an attacker has possibly transmitted. The authors show this method can trigger very few false positives, while being able to identify DNS tunnels in large ISP datasets.

Instead of quantifying the amount of information transmitted, Nadler et al. [92] propose an isolation forest classifier to identify DNS exfiltration attempts. The model uses features such as character entropy, unique query ratio, query volume, query average length, and other DNS characteristics. The authors show that their

approach can identify exfiltration attempts with both high and low throughput. Cambiaso et al. [93] propose a combination of dimensionality reduction and mutual information to model the normal behavior of DNS traffic. These techniques are applied to statistical features describing the inter-arrival time of DNS messages. The proposed technique is able to detect high throughput exfiltration attempts. Homem et al. [94] propose to learn the information entropy typically observed within the DNS messages, and use this characteristics to identify communication channels that embed other protocols, such as HTTP or FTP, within DNS. The proposed solution also identifies what type of protocols is being tunneled through DNS.

Some researchers has focused on covert channel detection over DNS using flow-based data. Ellens et al. [95] discuss a technique that use different statistical methods to identify DNS exfiltration. The authors use 8 numerical features related to DNS flow data (e.g., bytes per flow, packets per flow, flow duration, etc.), and provide a set of detectors that can be used to identify anomalies in DNS traffic. Similarly, Aiello et al. [96] use DNS flow statistics to identify DNS tunneling, but propose to model the traffic according to a two-tier system that rely on different machine learning algorithm to identify anomalies.

HTTP

Pack et al. [97] present an HTTP tunnel detection system by profiling user behavior according to traffic flow characteristics (e.g., average packet size, total packet size, etc.). The system tries to identify anomalies in sequences of HTTP messages, and within HTTP sessions. Anomalies are further analyzed by a verification module, which triggers an alert to the administrator in case it identifies specific keywords in the anomalous traffic.

Borders and Prakash [40] propose Web Tap, a system that models the outbound traffic characteristics of internal hosts to detect covert channels over HTTP. Web Tap profiles each host's traffic according to a set of filters (or sub models). Each model tries to identify anomalies within different HTTP aspects, such as header formatting, time delays, requests size and regularity, and so on. The system flags anomalies when HTTP connections do not fit at least one of these models.

A different approach was taken by Dusi et al. [41]. The authors propose to use only TCP information for covert channel detection, and they propose Tunnel Hunter, a system that uses one-class classification to identify anomalous traffic hidden in HTTP and SSH. Models rely on statistic information related to the packet size and the inter-arrival time between two consecutive packets, following the protocol fingerprinting ideas discussed in [98]. In case of HTTP, the model is trained with only benign HTTP data. The authors show that HTTP tunnels (e.g., SMTP over HTTP) can be identified by the models because tunnels statistics deviates from normal HTTP behavior.

A more quantitative approach was introduce by Borders and Prakash [99]. The authors introduce a new approach to quantify data leakage in HTTP traffic. The proposed approach inspects three main components of web traffic: HTTP messages, HTML content and Javascript code. The authors' solution evaluates these

components statically (e.g., header fields in HTTP messages) and dynamically (e.g., execute the scripts and analyze their generated traffic), in order to precisely quantify the amount of outgoing traffic. The authors show their approach is able to highlight the presence of exfiltration.

More recently, Schwenk and Rieck propose DUMONT [42], a system to detect covert communication over HTTP. DUMONT use a One-Class SVM (OCSVM) classifier to build a model for each monitored host. The system uses different numerical features to characterize normal HTTP traffic. The features are categorized as length, structural, entropy and temporal features of HTTP requests. The author show that DUMONT is able to identify HTTP tunnels and backdoors, and that it has better detection performance than Web Tap [40].

Other protocols and techniques

Marchetti et al. [100] propose a technique to rank internal hosts according to their possible involvement in data exfiltration or other malicious activities. The authors' solution analyzes only flow-based information, so it can be applied in any network independently from the protocol.

He et al. [101] discuss a method to identify encrypted data exfiltration in a cloud environment. The method models the meta-information (e.g., destination IP, port) of the encrypted communication of each host. Once the training phase is completed, all new encrypted communication are evaluated against the model of their corresponding host. In experimental settings, the authors show that their technique detects unexpected encrypted channels.

It is also possible to detect data exfiltration by detecting malware behavior. Botnets are known to perform data exfiltration activities [39]. Therefore, data exfiltration detection can be performed by detecting the presence of botnets. There exist several works in the literature that describe anomaly-based solutions to detect botnets. For example, botnets can be detected by correlating and finding similarity in spatial-temporal traffic characteristics [102], by clustering groups of hosts showing both similar network behavior (e.g., coordinate activities) and behavior that may indicate malicious activities [103] or by analyzing the persistent of traffic destinations [104].

2.3.3 Focus on HTTP traffic

Protocols with many redundant and uncertain values are ideal for storage covert channels [83]. Consequently, these diverse values are ideal to exfiltrate data, because a lot of secret information can be hidden, while it is difficult to distinguish them from normal protocol values. Furthermore, the larger is the volume of traffic generated by a protocol, the harder it becomes to reliably identify covert channel, because the detection mechanism would likely generate more false alerts. Protocols with these characteristics are ideal for an attacker also because they are not blocked by firewalls, and they guarantee communication with external entities.

The two main protocols that share these characteristics are HTTP and DNS, and it is not a coincidence that most malware uses these protocols for their com-

munication channels [37]. These protocols are both used to access web content, therefore, they inherently transport heterogeneous values, thereby making the detection of these covert channels not trivial. Furthermore, they are both never blocked by firewall. TCP/IP are also never blocked and have redundant header values to hide data into, but the bandwidth headers can guarantee to a covert channel is rather low compared to HTTP and DNS.

However, DNS only offers a single header value to hide data in: the Query Domain Name. This is a limitation that reduces the bandwidth of the covert channel. However, this limitation also helps detection mechanism to focus on a single aspect of the protocol, and it is the main reason why practical anomaly-based detection solutions for DNS exfiltration exist (e.g., [91], [92]).

On the other hand, HTTP offers multiple locations where data can be exfiltrated, such as URI, Body, and Cookie, just to name a few. This provides a high bandwidth for the covert channel, and it also makes detection difficult because multiple parts of the protocol could potentially be used to hide data. This helps explaining why a large majority of malware still uses HTTP for its communication [37]–[39]. Moreover, it also helps explaining why there exist no effective anomaly-based detection solutions to detect this type of threat. For the aforementioned reasons, we focus our efforts on detecting data exfiltration in the setting of HTTP.

2.4 Summary

In this chapter we have described the following:

1. A brief overview on the architecture of intrusion detection systems.
2. The focus of our work in the setting of network intrusion detection systems (NIDS), providing a detailed description of its main components.
3. We described how NIDS performance are usually evaluated.
4. We discussed the related work on the detection of data exfiltration, and we motivated our choice of focusing our efforts mainly on HTTP traffic.

This chapter provides the background knowledge to understand our works on the detection of data exfiltration, which we start discussing from the next chapter.

Chapter 3

Detecting Data Exfiltration via Application Fingerprinting¹

As discussed in Chapter 1, existing data exfiltration detection approaches over HTTP are limited. On the one hand, approaches based on predefined threat knowledge (e.g., signature-based) cannot identify unknown data exfiltration attempts. On the other hand, existing anomaly-based detection solutions are not precise in terms of detection performance, mainly due to the coarse granularity of their detection models.

In this chapter, we address **RQ1**. We present an anomaly-based detection approach based on *passive application fingerprinting*, and we introduce DECANter, a system that relies on passive application fingerprinting to detect anomalous outbound HTTP traffic. The core idea behind our approach is that malicious software, as any other application, has its unique traffic characteristics, which differ from other applications' characteristics installed on the same machine. Hence, if we can precisely recognize application traffic via application fingerprints, we can also identify unknown application traffic, which may indicate the presence of a newly installed malicious application. Our approach uses fine-grained detection models, because it models the HTTP traffic of each application installed on a monitored machine (i.e., application fingerprints). On the other hand, existing work [40], [42] generates a single model representing the whole HTTP traffic of a single host, thereby obtaining a more coarse-grained detection model, since the characteristics of all application traffic is modeled together. Figure 1.1 gives an overview of the granularity of the existing detection models. We implement DECANter and we evaluate it with realistic data from an international organization and datasets composed of malicious traffic. We show DECANter outperforms DUMONT [42], the state-of-the-art regarding anomaly-based covert channel detection over HTTP, in terms of detection performance and evasion resistance. Furthermore, we we publicly released the dataset of data exfiltration malware samples we used in our research. In addition, we released the implementation of DECANter and our

¹This chapter is an edited version of a peer reviewed paper that appeared in ACSAC '17 [1]

implementation of DUMONT that we used for comparing the two approaches.

3.1 Motivation

Existing anomaly-based detection systems for covert channels over HTTP do not show good detection performance [40]–[42], meaning that they either raise too many false alerts (i.e., high false positives) or they miss too many exfiltration attempts (i.e., high false negatives). This limitation is a consequence of the coarse granularity of their detection models.

Dusi et al. [41] propose to model the typical behavior of HTTP traffic using packet sizes and inter arrival time as characterizing features. Their model can distinguish when a protocol is being tunneled over HTTP (e.g., SSH messages are hidden within the HTTP protocol). However, in case the exfiltration occurs using the HTTP protocol itself, and not by using a different protocol within HTTP, (e.g., sending a POST request containing a list of password to a web server), the proposed solution does not detect the exfiltration attempt.

Borders and Prakash [40] propose to model different HTTP statistics (e.g., header formatting, request delays, etc.) for each monitored host. The proposed system, Web Tap, can identify a data exfiltration attempts that have values significantly different from the learned characteristics. Unfortunately, their approach is not practical, because it generates a high number of false alarms and it is easy to evade, due to its strong dependency on static header fields [42].

Schwenk and Rieck [42], similarly to [40], propose to create a single model to characterize the HTTP traffic generated for each monitored host. DUMONT [42] uses numerical features describing the content and the temporal characteristics of HTTP requests. Overall, DUMONT shows better detection performance than Web Tap [40], in terms of false positive rate and evasion resistance. The HTTP model of DUMONT can identify exfiltration attempts if the malicious messages have “different structure” than benign messages. For example, exfiltration attempts are detected if the generated messages contain very long headers field or header values, or if they contain content with high entropy (e.g., encrypted data), or if there is a sudden large number of outgoing messages. However, if the attacker generates messages with a “normal structure”, DUMONT likely misses those attempts. An attacker can achieve this by using common header fields and values, and by hiding the data to exfiltrate in parts of the protocol known to contain large values (e.g., URI).

Due to the aforementioned limitations, we decided to investigate a different, fine-grained, approach, which we describe in this chapter.

3.2 Intuition

The intuition of our work is the following: all traffic generated by a specific host is the consequence of network activities produced by a set of applications $A = \{a_1, \dots, a_n\}$ installed on the host. Each application a_i has specific network characteristics, and it is possible to create a fingerprint F_{a_i} for each application.

The network traffic of a host H can then be defined as the union of all application fingerprints: $H = \bigcup_{i=1}^n F_{a_i}$. Malware is also an application and is likely to have its own fingerprint. This also holds in case of malicious add-in software (e.g., a browser add-on produces a different fingerprint than the browser itself). Therefore, when the malware infects the host and communicates with the outside world, it should be possible to distinguish its traffic because it should differ from the set of benign applications A installed on that host. In other words, the set fingerprints act as a whitelist, and traffic that does not match the whitelist can represent a potential anomaly.

Although the intuition seems straightforward, there are several challenges to address. Traditional fingerprinting solutions create fingerprints from offline and often complete datasets, where an application is dynamically analyzed according to different inputs in order to trigger all possible behavior embedded in it [105]. In our setting this would require an analysis a priori of all existing HTTP clients (not only browsers), which is unrealistic. Therefore, one challenge is to generate fingerprints of applications from live traffic, which is likely to be incomplete (due to limited capture time) and heterogeneous (due to differing messages of the same application over time). Secondly, the system should provide an updating mechanism in case new fingerprints are created owing to new software installed on the host. In this chapter we address both challenges.

We are aware that many fingerprinting techniques have been proposed in literature, such as mobile applications or browser fingerprinting. However, those proposed techniques are not directly applicable to our setting, because they use features that are not able to characterize the traffic of workstation applications. For this reason, a new method is required.

3.3 Threat Model

We consider a scenario where an enterprise monitors the network traffic of its hosts by (passively) routing all traffic through a network monitor that cannot be bypassed. We assume that the network monitor cannot be compromised by an attacker. This is a common assumption because access to the monitor is assumed to be restricted. We also assume there is a security operator who analyzes the alerts produced by the network monitor. The attacker can, however, infect one or more enterprise monitored hosts with malware.

We assume the malware uses HTTP to communicate over the network. As already mentioned in Section 2.3.3, we focus on HTTP traffic mainly for two reasons: 1) a large majority of malware uses HTTP [37]–[39], either to communicate with their C&C server or to exfiltrate data; 2) many enterprise firewalls implement strict filtering rules that block non-web traffic, which forces malware to avoid customized protocols and to use HTTP or HTTPS.

Our approach can be applied to encrypted traffic only if the enterprise deploy a TLS proxy. Although proxies are not an optimal solution due to their security and privacy concerns [73], they are often deployed in enterprise environments to enhance network visibility and to protect the most sensitive parts of their networks,

Figure 3.1: High level overview of DECANter’s architecture.

by inspecting the traffic with advanced security solutions. These sensitive networks are those likely to be targeted by data exfiltration attempts.

3.4 DECANter

DECANter is a system with the goal of identifying anomalous outbound HTTP traffic. DECANter has two different phases: *training* and *testing*. The training phase works as a setup phase. For a fixed amount of time, the system passively monitors the network and learns the application fingerprints for each monitored host. Similarly, during the testing phase, the system monitors the network and extracts application fingerprints. However, the system additionally compares the newly extracted fingerprints to those learned during the training phase, in order to determine the presence of new applications on the monitored hosts. A high level overview of DECANter is shown in Figure 3.1.

The *training mode* is divided into two modules: labeling and fingerprint generation. The *labeling* module groups information from HTTP requests in different clusters and labels each cluster with its application type (i.e., browser or background). The labeling module enables the *fingerprint generation*, which creates the fingerprint according to the application type (i.e., label) of the cluster and its HTTP requests. Different features are used to describe browser and background messages as a consequence of their different characteristics. The outcome of the training mode is a set of fingerprints for each monitored host.

The training mode is assumed to be trusted, meaning that we assume no malicious application is installed on the machine. In practice, we can avoid to include malicious data in the training phase in two ways: 1) the host is in a malware-free state (e.g., just formatted or new); or 2) exclude from the training phase all the requests that are flagged as malicious by external threat intelligence. In our evaluation we use both strategies, depending on the dataset.

The *testing mode* has three modules: labeling, fingerprint generation and detection. The first two modules work the same as in training mode. The labeling and the fingerprint generation modules extract the set of fingerprints seen in the

Figure 3.2: Overview of different cases for the DECANTeR labeling method. A) represents the case of a background application cluster. B) represents the case of a browser application cluster, where two requests are considered suspicious of exfiltrating data and not being a browser. In the figure, each circle represents a request. Black circles are head nodes, outlined circles are requests spawned by head nodes, dashed circles are disconnected nodes, and dashed circles filled with a pattern are requests that may exfiltrate data.

network in the last X minutes. Next, the *detection* module verifies through different similarity checks whether these newly extracted fingerprints are anomalous or not.

Our solution focuses on outgoing HTTP requests only, and more specifically, GET and POST requests, as these are most commonly used. However, our method can easily be extended to other request types.

3.4.1 Labeling

The labeling module takes as input HTTP requests, clusters them according to their **User-Agent** header field, and assigns a label to each cluster according to its application type. Benign applications often use unique **User-Agent** values to be recognized by web servers, so there is almost a guarantee that all those requests have been generated by the same application. Therefore, using this value, allows us to isolate requests from distinct applications in an efficient manner. The labeling module collects traffic for a time window that we call *aggregation time* t . During testing mode, t is a fixed time window of X minutes, while in training mode, t matches the length of the training period. When t ends, each labeled cluster is passed to the next module.

Application Types: Background vs. Browser We have identified two types of HTTP applications: *background* and *browser*. The *background* type represents those applications whose traffic content and destination are not directly influenced by user inputs (e.g., an antivirus update query). These applications have predictable behavior, and show fixed patterns in their communication. They often use the same structure of HTTP headers. Their communication is often towards the same set of domains, and the size and content of their requests is similar over time.

The *browser* type represents web browsers, which generate HTTP traffic whose content is unpredictable and dynamic, because it directly depends on both user

actions and the specific visited web sites, especially considering the widespread use of dynamic web content.

The Labeling Process

The goal of the labeling process is to distinguish between background and browser application clusters. For each cluster, we inspect all the requests and we try to determine whether they show the dynamic behavior typical of browser traffic. We achieve this by relying on a specific data structure called the *Referrer graph*. If we are able to construct a Referrer graph from the requests of a cluster, then the cluster is labeled as a browser application, otherwise as background application. Mistakes in the labeling process likely cause false positives, because the clusters of a specific application would be fingerprinted with a different feature set and the fingerprint would not match the one of the original application. This issue is discussed further in our evaluation in Section 3.5.3.

In case the cluster is labeled as a browser, we apply an *exfiltration filter* over the requests that are not connected to the graph. This filter tries to identify requests that are potentially hiding in browser traffic while trying to exfiltrate data. This filter searches for requests (i.e., POST or GET requests with parameters) that are similar and which repeat over time, because they may represent an attempt of exfiltrating data with subsequent similar messages. The requests matching the filter are inserted in a new cluster labeled as a background application, otherwise they are considered part of the browser cluster previously identified. Figure 3.2 shows a detailed overview of the labeling process. In the next paragraphs we provide a more detailed description of the Referrer graph and the exfiltration filter.

The Referrer Graph

The labeling process distinguishes background and browser application clusters by leveraging the dynamic behavior of browser traffic. For example, when a user visits a website, the browser generates a request for a web page (usually HTML). Once the HTML page has been downloaded, the browser generates additional HTTP requests to retrieve extra information such as images, scripts, CSS and others. This information is needed to properly render the webpage. This behavior is unique to browsers, and it is not present in background applications traffic. Therefore, we can assign an application type to each cluster by identifying such behavior in the cluster’s requests.

Several researchers have previously proposed some solutions [106]–[108] to encode the dynamic behavior of a browser into a graph data structure known as *Referrer Graph*. Nonetheless, none of them are directly applicable to our setting. ReSurf [108] and ClickMiner [106] focus on reconstructing the user browsing activities from network traffic into a graph by connecting all requests that have been generated by user input, and consequently discarding all the other requests. The Triggering Relation Graph (TGR) [107] connects requests to user input in order to detect stealthy malware activities (i.e., identified by disconnected nodes). User input is collected by an agent hooking browser functions. The TGR approach

cannot be used because it requires access to the host, which is outside of our system model. The ClickMiner approach could be used, but processing all servers responses would be too resource demanding for live traffic analysis. Additionally, DECANTeR needs to identify all requests generated from the browser and not only from the user, as ReSurf and ClickMiner do.

For these reasons, we introduce a new approach to generate a *Referrer Graph* by introducing the concept of *head nodes*. For each cluster request we check based on the `CONTENT-TYPE` header if it accepts HTML, javascript, CSS or flash content. If so, we consider it as a *head node*. A head node is a request that may lead to the generation of other HTTP requests. Once we have identified all head nodes in a cluster, we link to them all requests that they have spawned; thereby a graph is created. Requests are linked to head nodes if their `Referer` or `ORIGIN` domain value matches with the `HOST` value of the head node. Head nodes may spawn other head nodes. If at least one graph is present, the connected nodes are moved to a new cluster, which is labeled as browser. In case all nodes are disconnected, we check if the nodes connect to the Referrer graph of the previous time slot $t - 1$. This step is needed because the head node, which may have spawned one or more requests, may have appeared during the previous aggregation time $t - 1$. Therefore, a link during aggregation time t cannot be found. If the node connect to the head nodes of the previous graph, the process continues as discussed above. Otherwise, we label all nodes in the cluster as background. Essentially, the Referrer graph distinguishes browser and background clusters.

The Exfiltration Filter

A malware may use the same User-Agent as the browser installed on the victim (accidentally or not). In case its traffic is close in time with (real) browser traffic, it is likely that malware traffic is included in the same cluster with the browser. This would allow an attacker to evade detection. However, it is unlikely that malware generates requests according to a Referrer Graph. Thus, the exfiltration filter tries to identify potential requests in browser clusters that may represent malicious exfiltration attempts, and re-classify them as background application. As such, they are more likely to be identified as new applications.

In case a cluster is labeled as a browser application, the labeling process further analyzes the disconnected nodes (i.e., nodes in the cluster that are not linked to the Referrer graph). We want to check if, in a cluster labeled as a browser, there are hidden malicious requests trying to exfiltrate data. The idea is to identify similar requests that transmits an excessive amount of information. For example, a set of successive POST requests containing different data items in the body of the message.

Firstly, the exfiltration filter marks all the disconnected requests that use the POST method, or use a GET method and include query parameters in the URI. Secondly, the filter searches for requests that share similar header fields and same URI path. We do not look for similarities in query values and parameters because, if they are used for exfiltration purposes, their values change across requests. Lastly, we check whether the values contained in the POSTs bodies, or GETs parameters,

exceed an outgoing information threshold of 500 bytes. If there are multiple requests that share the same URI, similar header fields, and exfiltrate enough data, then we label them as background request and insert them in a new cluster. The other disconnected nodes are inserted in the browser cluster.

3.4.2 Fingerprint Generation

This module takes as input the labeled clusters from the labeling module. The fingerprint generation module creates a fingerprint by extracting a specific set of features depending on the cluster label. We inspect each request in the cluster and we extract the following features:

1. *Host*: the set of domains that are stored in the HTTP field **Host**. More specifically, we consider only the top and second level domains. *Intuition*: we have observed that many applications, which often operate as background services, mostly communicate with the same set of domains.
2. *Constant Header Fields*: the set of header fields *always* present in the cluster requests. *Intuition*: many applications, especially non-browser applications, systematically use a fixed set of header fields for each request they generate, making it a unique characteristic. This feature is not new, but previously it was used to model malicious communication [61], while we use it to model benign software communication.
3. *Average Size*: the average size of an HTTP request, computed from the sizes of all HTTP requests (including both header and body size). *Intuition*: although the content may vary per request, some applications often generate requests of very similar sizes, especially when they are generated systematically.
4. *User Agent*: the string of the request field **User-Agent**. *Intuition*: this value is often unique for each benign application.
5. *Language*: a set of string values present in the **Accept - Language** HTTP field. *Intuition*: web browsers use this field to advertise which natural languages they prefer in the HTTP response. This field characterizes not only the instance of the browser, but also the user settings.
6. *Outgoing Information*: a close approximation of the total amount of information transmitted by all the requests belonging to the cluster. This feature is used only during testing phase. *Intuition*: we want to keep track of how much information the requests within a cluster have transmitted.

It has been already shown that malware can use **User-Agent** values that look like valid browser user-agents [37], [38]. Therefore, a fingerprinting method that relies on this feature alone is not effective. Hence, we want to create fingerprints that rely on several characteristics of an application network traffic. In this way, even if the malware correctly guesses the **User-Agent** value, we can still identify the malicious fingerprint as anomalous, because the other features may not match with the real application fingerprint.

Outgoing Information

The way the amount of information is calculated is relevant, especially for the case of data exfiltration. If an application periodically generates HTTP requests that always have the exact same content (e.g., antivirus software requesting updates), the amount of outgoing information should be only the information contained in the first request. On the other hand, if an attacker exfiltrates a data item through several HTTP requests, those requests should contain different information, and the amount of outgoing information should be as close as possible to original the exfiltrated item. A naïve approach that quantifies the outgoing information by summing the size of HTTP requests would fail to address these two cases, because it would not consider the content differences between requests.

Therefore, we introduce a new method to compute a more precise amount of outgoing information (*OI*). Given a set of requests $\text{REQ}_1 \dots \text{REQ}_n$ in cluster i ,

$$OI_i = \text{size}(\text{REQ}_1) + \sum_{i=2}^n \text{LevenshteinDist}(\text{REQ}_i, \text{REQ}_{i-1}).$$

The intuition behind this approach is that the content of the first request should be considered new information, while the next requests add new information *only if* they contain new (different) information than their prior request. Please note that to obtain a tight upper bound, we should compute the edit distance between the current request i and all previous requests, and then sum only the minimum value. However, this would be computationally expensive, therefore we decided to compute the edit distance only with the previous request $i - 1$.

Algorithm 1 Computation of Outgoing Information. The operator 'in' is iterator for the object, while '.' access an attribute of the object. In this algorithm a header is an attribute of a request.

Require: the current request REQ_i , the latest request REQ_{i-1} transmitted by the same application.

```

1: procedure COMPUTEOUTINFO( $\text{REQ}_i, \text{REQ}_{i-1}$ )
2:    $OI \leftarrow 0$  ▷ Initialize counter
3:   for all  $h_j$  in  $\text{REQ}_i$  do ▷  $h_j$  is the  $j$ th header in  $\text{REQ}_i$ 
4:     if  $h_j$  in  $\text{REQ}_{i-1}$  then
5:        $OI += \text{edit\_dist}(\text{REQ}_i.h_j, \text{REQ}_{i-1}.h_j)$ 
6:     else
7:        $OI += \text{length}(\text{REQ}_i.h_j)$ 
8:     end if
9:   end for
10:   $OI += \text{edit\_dist}(\text{REQ}_i.\text{body}, \text{REQ}_{i-1}.\text{body})$ 
11:   $\text{updateCache}(\text{REQ}_i.\text{user-agent}, \text{REQ}_i)$ 
12: return  $OI$ 
13: end procedure

```

Algorithm 1 represents the pseudocode to compute the outgoing information between two requests. The function `ComputeOutInfo` takes as input two values: the

new incoming request REQ_i , and the latest analyzed request REQ_{i-1} , which is stored in a cache (i.e., hash table) that uses the **User-Agent** strings as keys. The function verifies if there are headers, including the **URI**, present in REQ_i that are also present in REQ_{i-1} (line 3-4). If so, it computes the edit distance (i.e., Levenshtein distance) between the two header values. The result is the new information introduced by the specific header value of REQ_i and it is added to the counter of outgoing information (line 5). For those headers that are present in REQ_i but not in REQ_{i-1} , we consider their values as new information (line 7). Then the edit distance between the body of the requests is computed and is added to OI (line 10). Once all headers have been analyzed, we update the cache by substituting REQ_{i-1} with REQ_i (line 11), because now REQ_i is the last analyzed request.

Different Set of Features for Different Labels.

The traffic patterns of background and browser applications are almost poles apart. This diversity is the main reason why we decided to have a different set of features to represent the fingerprints of these two types. Tailoring the set of features according to the characteristics of an application and its type, strengthens the fingerprint against malicious emulation attempts. For example, background applications have stable patterns, so if a fingerprint characterize them, then it becomes more difficult for an attacker to emulate the application since it has to follow the same patterns.

We model *background* applications using the following features: *Host* (1), *Constant Header Fields* (2), *Average Size* (3), and *User-Agent* (4). We model *browser* applications using the *User-Agent* (4), and *Language* (5). These features capture fixed communication patterns that are characteristics of background and browser applications, respectively. Moreover, for both application types, we keep track of the *Outgoing Information* (6) for each cluster to determine, during the detection, whether an excess of data is being transmitted or not.

3.4.3 Detection

The detection module takes as input a set of application fingerprints $F_{\text{test}} = \{F_{a_1}, \dots, F_{a_n}\}$. Each fingerprint F_{a_i} is compared against the fingerprints generated during the training mode $F_{\text{train}} = \{F_{b_1}, \dots, F_{b_z}\}$. The comparison is done by computing specific similarity functions, which are application-type dependent and verify whether the features of the two fingerprints are similar or not. In case F_{a_i} is not similar to any of the fingerprints in F_{train} , DECANter considers F_{a_i} as a *new application*. The list of similarity functions is shown in Figure 3.3.

Once a new application is found, DECANter verifies if the new fingerprint is a software update (see Section 3.4.4). In case an update is found, the fingerprint representing the “old version” of the application (i.e., before the update), it is updated with the new information. Otherwise, if F_{a_i} is not an update, *an alert is raised if at least one of these two conditions is satisfied*: 1) the amount of outgoing information of F_{a_i} is above a threshold σ , or 2) the **User-Agent** in F_{a_i} resembles a browser **User-Agent** string. 2) is checked by simply verifying if strings such as

“Firefox”, “Chrome”, etc. are in the **User-Agent** string.

The detection checks 1) and 2) are used for the following reasons: with 1) we want to know if new applications on the machine are transmitting too much data over the Internet. This may indicate a malware installed on the hosts that starts exfiltrating data; with 2) we want to identify those applications that are trying to imitate a browser. This check is based on a common malware behavior, which tries to use user-agent strings of known browsers to hide themselves [37], [38]. Therefore, new browser-looking fingerprints should be considered as anomalies. In case the operator considers the alert to be a false positive, he can add the fingerprint to the list of known application fingerprints, as discussed in Section 3.4.4.

Background Similarity Function

A fingerprint is represented by a set of features. Let us consider F_a and F_b to be two background application fingerprints generated in F_{test} and F_{train} , respectively. F_a and F_b have the same label. The background similarity function verifies whether F_a and F_b are representing the traffic of the same application or not. The function is defined as

$$s_{\text{back}}(F_a, F_b) = \sum_{i=1}^4 s_i(F_{a_i}, F_{b_i}),$$

where s_i represents a function that checks the similarity of the i th feature. The first four similarity functions in Figure 3.3 are used to compare background applications. F_a and F_b are considered similar if and only if $s_{\text{back}}(F_a, F_b) \geq \alpha$, where α is the similarity threshold for s_{back} .

Function s_1 (3.1) assigns 1 point if *all* the *Host* domains visited by F_a were also seen by F_b , and 0 otherwise. We require all domains to be present because background applications often talk with the same set of domains. If the domain set differs, it might be an indication that the two fingerprints are not representing the same application.

Function s_2 (3.2) assigns 1 point if the *Constant Header Fields* found in F_a traffic matches those in F_b , 0.5 if they are a superset of the headers of F_b , and 0 otherwise. HTTP headers are often repeated in background applications requests. However, sometimes the same application generates requests with additional header fields. This case is addressed giving half of the similarity points. In case any of the header fields observed during training phase is not present, the fingerprints may not represent the same application.

Function s_3 (3.3) assigns 1 point if the absolute difference between F_a and F_b —the *Average Size*—is lower than ϵ , where $\epsilon = \frac{F_{b3}}{3}$. If it is lower than 2ϵ , 0.5 and 0 otherwise. An exact match is almost impossible to find. Therefore, we use two different intervals that depend on ϵ , which represents an *error rate*, so we address the case where the average size may have changed due to some dynamic properties of the application communication. If the average size is not within the intervals, the fingerprints are likely not generated by the same application.

Function s_4 (3.4) assigns 1 point if the *User-Agent* of the two fingerprints matches, and 0 otherwise. In case there is no match, applications are likely different.

These functions describe patterns that we have observed on real traffic and are tailored to some HTTP characteristics. Through the combination of these four features it is possible to correctly identify the fingerprints sharing the same applications despite small changes in behavior, for example a change in **User-Agent** or communication with a different domain. The “similarity threshold” α also plays an important role, because it guarantees a certain flexibility. From our empirical evaluation, $\alpha = 2.5$ is the best value that allows us to match different fingerprints from the same applications, and to distinguish them from those of other applications. A threshold of 2.5 makes sure that a fingerprint should match with at least three features. A partial match of the constant header features scores 0.5 points. We have observed from our data that scores lower than 2.5 would match fingerprints that do not represent the same application, because they may share the same headers and size, but they communicate with completely different services (e.g., using distinct hosts and User-Agent). Higher threshold values do not guarantee flexibility towards changes in the requests. This can be a problem when fingerprints are trained on little data, which contains only a specific subset of the requests generated by the application.

Browser Similarity Function

The browser similarity function is easy to compute, because there are only two features to evaluate:

$$s_{\text{brow}}(F_a, F_b) = s_4(F_{a_4}, F_{b_4}) + s_5(F_{a_5}, F_{b_5})$$

where s_4 is the same as in the background setting. Function s_5 assigns 1 point if the *Language* of the two fingerprints matches, and 0 otherwise.

Fingerprints F_a and F_b are considered similar if and only if $s_{\text{brow}}(F_a, F_b) = \beta$, where β is the similarity threshold for s_{brow} . For browsers both features should exactly match, thus $\beta = 2$. A lower β would result in a more permissive check, allowing one of the two features to not match, which would lead in an easier evasion for the attacker.

$$s_1(F_{a_1}, F_{b_1}) = \begin{cases} 1 & \text{if } F_{a_1} \subseteq F_{b_1}, \\ 0 & \text{otherwise;} \end{cases} \quad (3.1)$$

$$s_2(F_{a_2}, F_{b_2}) = \begin{cases} 1 & \text{if } F_{a_2} = F_{b_2}, \\ 0.5 & \text{if } F_{a_2} \supset F_{b_2}, \\ 0 & \text{otherwise;} \end{cases} \quad (3.2)$$

$$s_3(F_{a_3}, F_{b_3}) = \begin{cases} 1 & \text{if } |F_{a_3} - F_{b_3}| \leq \varepsilon, \\ 0.5 & \text{if } |F_{a_3} - F_{b_3}| \leq 2\varepsilon, \\ 0 & \text{otherwise;} \end{cases} \quad (3.3)$$

$$s_4(F_{a_4}, F_{b_4}) = \begin{cases} 1 & \text{if } F_{a_4} = F_{b_4}, \\ 0 & \text{otherwise.} \end{cases} \quad (3.4)$$

$$s_5(F_{a_5}, F_{b_5}) = \begin{cases} 1 & \text{if } F_{a_5} = F_{b_5} \\ 0 & \text{otherwise} \end{cases} \quad (3.5)$$

Figure 3.3: DECANter’s similarity functions. The function subscript is enumerated according to the feature the function analyzes, as discussed in Section 3.4.2, while ϵ refers to the error rate accepted for the Average Size feature, discussed in Section 3.4.3.

3.4.4 Fingerprint Update

When the detection module finds a new application, it is possible that a false positive is triggered. This can happen for different reasons: a new application has been installed, the application did not communicate during the training phase, or an existing application has been updated. These are events that may happen over time during live monitoring, therefore it is important that DECANter is updated.

Common browsers, such as Chrome, often update themselves without user interaction. When a browser is updated, its **User-Agent** string changes, leading to a new different fingerprint. DECANter addresses this issue by verifying for every new fingerprint if the **User-Agent** string is similar to any of the existing fingerprints. DECANter computes the edit distance between the two strings, it divides the outcome with the length of longest string, and it obtains a final value between 0 and 1. If the value is smaller than 0.1, DECANter considers the strings to be similar. We have determined this threshold empirically. Therefore, only if a small part of the string changes, they are considered similar (e.g. increased software version). If the strings are similar, DECANter runs the similarity functions (again) according to its type, and it automatically assigns the maximum score for similarity function s_4 (3.4). If the fingerprints are considered similar, the older fingerprint is updated with the new information.

The other main causes of false positives are: 1) the installation of new software, and 2) software that did not communicate during the training mode. In these cases, DECANter is updated by the security operator. Once he flags an alert as a false positive, DECANter can simply add the flagged fingerprint in its pool of trained fingerprints. This method of updating is computationally efficient, because DECANter only needs to add an element into a set.

3.5 Evaluation

In this section we describe the datasets we used to evaluate DECANter. We discuss how the main system parameters aggregation time t and threshold σ (i.e., threshold for the outgoing information) are chosen. We evaluate the detection performance of DECANter and compare it with DUMONT [42]. We have implemented a Python version of DECANter and DUMONT², because its original implementation was not available. In this evaluation, we consider alerts triggered by malware to be true positive (TP), while false positives (FP) are those alerts triggered by benign software.

3.5.1 Datasets

User Dataset (UD). We have collected data from 9 researcher machines at an international university. An overview of the dataset is shown in Table 3.1. User 5 has a significantly higher number of trained fingerprints due to an adware installed on its machine. We used real data to avoid possible biases by capturing data in a lab (e.g., a fixed set of installed applications). The period of time of collection varies per user, as it spans from three working days to a few weeks. The dataset contains 123,766 HTTP requests and represents more than 493 hours of network traffic.

Organization Dataset (OD). DECANter has been deployed to an international organization monitoring outgoing HTTP traffic on a network link with thousands of hosts. The traffic was inspected using Zeek (previously known as Bro [109]) and filtered on port 80, which created ad-hoc HTTP logs that have then been processed by DECANter. From the organization, we have obtained 307,053 fingerprints (representing 3,773,106 HTTP requests) generated by DECANter for 441 partially self-managed hosts that communicated for a period of a month: 291 employee workstations, and 150 infrastructure machines.

Data Exfiltration Malware (DEM). We analyzed hundreds of malware samples within a virtual machine (VM) for roughly 60 minutes per sample using Cuckoo³. In our VM, we have installed known software, stored account credentials for real services (e.g., Gmail, LinkedIn), and placed some decoy documents of different format containing sensitive information, which we obtained from Wikileaks.

²Both implementations are available at <https://github.com/rbortolameotti/decanter>

³<https://cuckoosandbox.org/>

User ID	HTTP Requests	Total Hours Traffic	Training Time (hours)	Testing Time (hours)	Trained Fingerprints
1	6739	27	10	17	8
2	4709	48	7	41	3
3	10250	18	7	11	9
4	4480	30	8	22	7
5	20075	35	9	26	93
6	3296	47	9	38	6
7	18525	17	7	10	20
8	41527	125	5	120	8
9	14165	145	10	135	11

Table 3.1: User Dataset Overview.

We have removed all network data samples that generated less than 100 bytes of HTTP traffic. This resulted in 59 traffic malware samples known for their exfiltration capabilities. The samples belong to 8 families of information-stealer malware: iSpy, Shakti, FareIT, CosmicDuke, Ursnif, Pony, Dridex, and SpyEye. The main reason why the number is low is because the (fresh) malware we evaluated was able to detect the VM, and therefore it did not perform any communication. Despite the relatively low number of samples, we believe this dataset can have an important value for the community, because there are two requirements needed to collect it that are not easy to fulfill for all researchers: 1) access to fresh malware samples from specific malware families, because (even few months) old malware may not be able to communicate anymore, and 2) connect the VM to the Internet, which may not be allowed due to risks the infrastructure may incur by running live malware. The collection of this dataset has been approved by the IT department, and the dataset is public ⁴, together with both implementations, to foster research on data exfiltration.

Ransomware (RAN). This dataset consists of ransomware traffic. We obtained 290 pcaps from the authors of FSShield [79]⁵, and the virtual machine used in their analysis. The VM is used to label benign and malicious traffic (see Section 3.5.2). We removed those pcaps that did not generated at least 100 bytes of HTTP traffic, obtaining a total of 287 samples. These samples belong to 5 different families of ransomware: CryptoWall, CryptoDefense, Critroni, TeslaCrypt, and Crowti.

3.5.2 Evaluation Setup

The UD, RAN, and DEM datasets are network traffic files (i.e., .pcap). Each file is analyzed with Zeek [109] to generate a log file, which contains the HTTP headers and additional metadata for each request. The OD dataset is a set of log files.

⁴[https://scs.ewi.utwente.nl/downloads/show>DataExfiltrationMalware\(DEM\)/](https://scs.ewi.utwente.nl/downloads/show>DataExfiltrationMalware(DEM)/)

⁵The RAN dataset can be requested from the authors of [79].

Labeling the datasets. We have labeled malicious and benign requests in our datasets as follows. In UD, we have manually labeled requests as malicious if they were showing **User-Agent** values not matching any application installed on the machine (e.g., iPhone in **User-Agent** strings on a Windows machine). In RAN and DEM, we have generated network traffic with the VMs, without running malware, and we labeled as benign all requests that had same **User-Agent** as the applications found on the VMs traffic. The other requests are considered malicious, including browser requests, because we know that during the analysis only the malware could have used the browser to generate traffic. We have manually labeled the OD dataset after analyzing it with the help of external threat intelligence services⁶ and with indicators of compromise we have obtained from a professional threat intelligence provider.

Dataset split for training and testing. In all experiments with the UD dataset we have used the traffic of the first working day for training and the rest has been used for testing. The training mode—as any setup phase—is trusted, therefore we manually checked if there were malicious fingerprints, and if so we removed them. We removed dozens of connections known to be malicious from a single host, which was infected by malicious software. In the experiments with RAN and DEM we used the VMs traffic generated without malware for training, and all malicious samples are used for testing. In case of the OD dataset, we have trained DECANter using the first week of traffic, and the rest was used for testing.

Fingerprint Updates. In our evaluations, we assume that DECANter updates its fingerprints. Thus, if DECANter would raise a false positive for five consecutive time slots, and these are identical to each other, only the first is counted as a false positive and the rest is considered true negatives. This represents the scenario where an operator flags the first alert as a false positive, and DECANter adds the new fingerprint to its trained set of fingerprints.

Parameter Selection. σ is the system parameter in the detection module that decides whether to trigger an alert or not, depending on the amount of new information generated by a fingerprint. We have evaluated the number of false positives (FP) σ would raise on the UD according to different values. Figure 3.4 suggests that the number of FPs is proportionally inverse to the threshold value. A high threshold produces few FP. Obviously, a too high threshold can also lead to a low detection rate of anomalous traffic. We suggest $\sigma = 1000$ bytes, because the decrease of FP is less significant for higher thresholds. Moreover, 1000 bytes is a low value that could detect an exfiltration of very small data items such as cryptographic keys (e.g., a .pem file with a 2048-bit RSA key has a size of 1700 bytes).

⁶We have used the information provided by ThreatCrowd (<https://www.threatcrowd.org/>) and VirusTotal (<https://virustotal.com/>).

Figure 3.4: Represents the number of false positives generated from the traffic of 9 users, from the UD dataset, for different values of σ , and aggregation time $t = 10$ minutes.

The other relevant parameter of DECANter is the aggregation time t , used in testing mode. The advantages of a long detection time are: the lower number of FP, because more requests are aggregated in the same fingerprint, and there are fewer fingerprints to analyze, if compared with lower thresholds; and an attacker, who wants to remain undetected, must severely decrease its communication. If we set $t = 30$ minutes and $\sigma = 1000$ bytes, it means that the attacker cannot transmit more than 1000 bytes within half an hour. The disadvantage is slow detection, because the detection is performed less often and an attacker can transmit more data before it is detected. A low detection time has the opposite advantages and disadvantages. We have tested DECANter with $\sigma = 1000$ and three different t values to understand the relation between t and the number of FP triggered. We used $t = 1, 10$ and 30 minutes. The overall number of FPs DECANter triggered was 116, 73, and 63 respectively. Showing a clear relation between time and FP reduction. Both 10 and 30 minutes cases generate a lower number of FP. However, we consider 10 minutes a better trade-off, because it still provides enough time to the operator to verify the alert, and it provides quicker detection. If having low FPs is a necessity, 30 or more minutes as t values is a better option. For the rest of the paper, we use $t = 10$ minutes and $\sigma = 1000$ bytes.

3.5.3 Detection Performance

We have evaluated the detection performance of DECANter against the UD, RAN, and DEM datasets. We have trained and tested DECANter for each dataset as discussed in Section 3.5.2. The analysis evaluates the number of fingerprints that are correctly classified (or not) by DECANter. Fingerprints are labeled as malicious if they contain at least one request previously labeled as malicious, otherwise they are considered benign. The results are shown in Table 3.2.

Analysis of the User Dataset. In the UD dataset, DECANter detected 117 malicious fingerprints for a specific user, despite the use of a known antivirus

Dataset	Malware Detection	Classification Performance					
		TP	FN	TN	FP	FPR	TPR
UD	-	117	36	4291	73	1.6%	76.4%
RAN	98.6%	3348	438	4257	2	0%	88.4%
DEM	96.8%	237	67	24	1	4%	77.9%

Table 3.2: Detection performance for different datasets with $\sigma = 1000$ and $t = 10$ minutes. Malware Detection indicates the percentage of detected samples by DECANter. For the classification performance, the values represent the number of fingerprints classified as true positives (TP), false negatives (FN), true negatives (TN) or false positives (FP). FPR indicates the false positive rate $FPR = \frac{FP}{FP+TN}$, while TPR indicates the true positive rate $TPR = \frac{TP}{TP+FN}$.

software. The user was infected by an adware, which explains the presence of many different fingerprints. Many requests contained distinct **User-Agent** values (even mobile strings), header fields and domains. Possibly, the malware wanted to emulate various requests, as they were generated from distinct hosts, to increase the number of visits (or clicks) to specific ads links. Although adware may not seem very harmful, we know that they also have data exfiltration capabilities [18]. The false negatives (FN) were all unknown fingerprints, meaning that they did not match any of the existing fingerprints. However, DECANter did not trigger an alert because they transmitted fewer bytes than the threshold σ (i.e., 1000 bytes) during the aggregation period (i.e., in our evaluation is 10 minutes). Additionally, those fingerprints did not have a **User-Agent** value of a known browser, which is the last check that triggers alerts in case of fingerprints that transmit little data. The false positives (FP) were mainly caused by applications that were not fingerprinted during training mode. For instance, traffic generated by a Windows VM on a Linux host. Other FPs were triggered due to labeling “mistakes”, where some browser requests were considered as background. This labeling mistakes mainly occurred in case of web scripts, which do not reference to previous requests and send out information through GET with parameters (or cookies). OSCP POST requests also were often mistakenly consider as background applications. Nonetheless, when DECANter is updated, it learns the background fingerprints of these scripts, and next time they are triggered DECANter will not raise a FP. Since we do not know the amount of malicious software running on the infected host, we do not know the number of malicious samples DECANter has detected.

Analysis of Data Exfiltration and Ransomware Malware Datasets. In the RAN and DEM datasets, DECANter classifies most of the fingerprints correctly. As shown in Table 3.2, DECANter on average detects 8 malicious fingerprints out of 10. Considering that a malicious sample is successfully detected if at least one TP is triggered to the operator, our system detected 98.6% and 96.8% of malicious sample from the RAN and DEM dataset, respectively. The FPR for the DEM dataset seems very high, but only 1 FP has been in fact triggered by DECANter.

Similarly to the UD dataset, FNs are mostly caused by malware that have fingerprints with low outgoing information. Moreover, a few malware samples (i.e., Ursnif family⁷), which were successfully detected, generated additional browser-like traffic that was not properly classified. The samples in question turned out to generate traffic that matched the browser traffic of our VM, because it generated a Referrer Graph and it has been analyzed accordingly by DECANTeR. Therefore, the communication was considered as a FN. However, this was only noise created by the malware, which communicated with its C&C through other requests that had different **User-Agent** and were not connected in any graph. This led to the generation of background fingerprints that have been detected by DECANTeR as anomalous, resulting in a successful detection of the sample. Another interesting behavior used by other Ursnif samples was a mimicking attempt. Figure 3.5 shows a comparison between the mimicking request 3.5b and the original request 3.5a. The malware camouflaged itself by using a legit **User-Agent** value (i.e., Microsoft Crypto API 6.3) and a valid URL (e.g., the shown URL is commonly used by Microsoft Crypto API requests). However, as shown in Figure 3.5b, there have been four differences that exposed the malware to detection. First, the malware used a different set of headers than the original software (e.g., the **Check** header does not exist). Second, the size of the exfiltrating request was much larger than the original request (e.g., 3000 bytes per request). Third, the **User-Agent**, despite being a valid value, did not match with the software version used on the infected machine. Lastly, the **Host** value was not a standard DNS hostname, typically used by the original software. Due to these differences, DECANTeR was able to identify this case as well.

Analysis of the Organization Dataset. DECANTeR detected 8 machines with actual anomalous behavior. Half of these machines showed traffic patterns known to be caused by malware according to different sources. The other half has shown known anomalous patterns related to advertisement websites, which suggests the presence of some type of adware, perhaps browser hijackers. Unfortunately, we could not check directly the host machine to get further proofs. In one case, DECANTeR has identified a malicious IP address before being blacklisted by VirusTotal. The overall FPR⁸ is 0.9%. The specific FPR values per machine category are: 1% for workstations and 0.3% for servers. These values are expected, because workstations produce more outbound HTTP traffic.

3.5.4 Comparison with DUMONT

The closest anomaly-based related work to DECANTeR are WebTap[40] and DUMONT [42]. In [42] the authors have already compared these two solutions, showing that WebTap suffers of a high false positive rate, and the detection rate drops significantly in case malware would use simple evasion techniques. Therefore, we compare DECANTeR with DUMONT.

⁷ <https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Win32/Ursnif>

⁸Also in this scenario we assume an operator is updating DECANTeR.


```

> Internet Protocol Version 4, Src: 192.168.56.12, Dst: 93.184.220.29
> Transmission Control Protocol, Src Port: 50182, Dst Port: 80, Seq: 1, Ack: 1, Len: 231
4 Hypertext Transfer Protocol
  > GET /MFEwTzBNMEswSTA3BgUrDgMCGGUABBTfqhLjKLEJQZPin0KCzkdAQpVYowQUsT7DaQP4v0cB1jgmGggC
    Connection: Keep-Alive\r\n
    Accept: */*\r\n
    User-Agent: Microsoft-CryptoAPI/6.1\r\n
    Host: ocsdp.digicert.com\r\n
    \r\n
    [Full request URI: http://ocsdp.digicert.com/MFEwTzBNMEswSTA3BgUrDgMCGGUABBTfqhLjKLEJQ]
    [HTTP request 1/2]
    [Response in frame: 12906]
    [Next request in frame: 17595]

```

(a) Original Microsoft Crypto API request.

```

> Internet Protocol Version 4, Src: 192.168.56.12, Dst: 5.135.156.80
> Transmission Control Protocol, Src Port: 49778, Dst Port: 80, Seq: 1, Ack: 1, Len: 2280
4 Hypertext Transfer Protocol
  > GET /pki/mscorp/crl/msitwww2.crl HTTP/1.1\r\n
    [truncated]Check: 9bbd04e3bd639262e50797c5a7429d4505149642a2de8e70d97b8c15a8c6827e67
    User-Agent: Microsoft-CryptoAPI/6.3\r\n
    Host: 5.135.156.80\r\n
    Cache-Control: no-cache\r\n
    \r\n
    [Full request URI: http://5.135.156.80/pki/mscorp/crl/msitwww2.crl]
    [HTTP request 1/1]
    [Response in frame: 121]

```

(b) URSNIF mimick attempt.

Figure 3.5: Mimicked attempt from URSNIF malware detected by DECANter. Figure 3.5a shows a typical request of Microsoft Crypto API from the infected machine. Figure 3.5b shows the mimicking request generated by the malware. Red boxes highlight the differences between the two requests. The mimicked request has: 1) a different list of headers; 2) large header value; 3) valid **User-Agent** but wrong in the context of the infected machine; and 4) IP address instead of DNS hostname for **Host** value.

System	Dataset	Malware Detection	Classification Performance					
			TP	FN	TN	FP	FPR	TPR
DECANter	UD	-	928	51	90566	3378	3.5%	94.7%
	RAN	98.6%	81910	1123	13520	10	0%	98.6%
	DEM	96.8%	4887	2643	352	3	0.8%	65%
DUMONT .1	UD	-	49	930	87003	6959	7.4%	5%
	RAN	81.8%	17426	65607	13529	4	0%	20%
	DEM	4%	20	7513	351	4	1%	0.2%
DUMONT .6	UD	-	164	815	64824	29138	31%	16.7%
	RAN	100%	81708	1325	1203	12330	91.1%	98.4%
	DEM	40.5%	2688	4845	132	223	62.8%	35.6%

Table 3.3: Comparison between DECANter and DUMONT, evaluated over our three datasets. We show the results for DUMONT according to two different threshold, one (0.1) that is conservative and tries to raise the least number of FPs, and the other (0.6) that shows a better ratio FPs and TPs.

DUMONT creates a one-class SVM (OCSVM) from HTTP requests according to 17 different numerical features that involve different metrics such as entropy, length of header fields, and temporal traffic characteristics. During a testing phase, DUMONT verifies if the features of each new requests are within a certain distance from the hypersphere represented by the OCSVM. We have reimplemented DUMONT because its implementation was not available. Similarly, we evaluated it using our datasets, because the dataset used in the original paper was not available. Our implementation takes as input only HTTP headers, therefore DUMONT cannot use the entropy features for HTTP POST body. Considering that the number of POSTs in the network is much lower than the number of GET requests, we do not consider this a serious limitation. For each experiment, we have followed the procedure discussed in [42], by calibrating the model with malicious data (from both RAN and DEM datasets), and tested it with different parameters (i.e., OCSVM soft margin) to find a suitable ratio of false positive and detection rate. In our results, we discuss two different values used to compute the optimal soft margin: 0.1 that triggers a low number of FPs, and 0.6 that gives a reasonable ratio between FPs and TPs.

For a fair comparison we have evaluated the correct classification of requests, in contrast to fingerprints, since DUMONT works only on requests. We considered for DECANter all requests in malicious fingerprints to be malicious. Similarly, all requests were considered benign for those fingerprints labeled as benign. After all, fingerprints are abstractions of a group of requests. The results are shown in Table 3.3. The first observation is DECANter outperforms DUMONT in all three detection aspects: FPR, TPR and Malware Detection (as shown in the tables). One of the biggest differences lies in the detection of malicious samples, where DECANter shows consistent detection independently from the underlying malicious data, while DUMONT detection strongly suffer this dependency. The second observation is that the detection performance numbers of DECANter are better than those shown in Table 3.2. The different results between the two evaluations are related to the distribution of requests across the fingerprints. For example, given two fingerprints, one classified as TP containing 5 requests and one FN with 1 request, then the TPR is 0.83. On the contrary, if we consider only the number of fingerprints (i.e., exclude the amount of requests), the TPR is 0.5.

The low detection performance of DUMONT is explainable by the fact that many malicious requests are not structured (e.g., length of header fields) differently from benign requests. Thus, the feature vectors representing the malicious requests do not deviate from benign requests feature vectors. DUMONT misses this type of requests. DECANter overcomes this issue by using semantics information and different features per application type, leading to a higher specificity. A surprising result is the high number of FPs in the UD dataset triggered by DUMONT. This may be caused by two different behaviors. Firstly, the training set did not contain all possible applications, so requests generated in testing phase by VMs, new browsers, and other applications may have influenced DUMONT. Secondly, many benign requests contain a large amount of data (e.g., COOKIE, large URI), which are bigger than average HTTP requests. DECANter suffers less these behaviors because it does not monitor structural characteristics (e.g., request or header

System	Dataset	TPR	Malware Detection
DECANTeR (requests)	RAN	99.9%	100%
	DEM	99.2%	100%
DUMONT .1	RAN	4.3%	25.6%
	DEM	0.4%	5.5%
DUMONT .6	RAN	58.4%	100%
	DEM	15.8%	62.1%

Table 3.4: Evasion test against DECANTeR and DUMONT

values length). Moreover, DECANTeR can adapt over time through the update mechanism. Thus, if a new benign application is detected as a false positive, its fingerprint can be added to the set of known fingerprints. A similar update process cannot be performed by DUMONT, because to retrain the entire OCSVM, the system has to recollect all the user data again and the training of the model again. This procedure can take a long time (e.g., several hours) depending on the amount of data and features.

3.5.5 The Evasion Test

As we know malware tries to imitate browser **User-Agent** strings [37], [38]. A piece of malware can choose an existing browser **User-Agent** or even copy the same as its victim’s browser, by inspecting the OS (e.g., Windows Registry) or by sniffing the network.

We evaluated how DECANTeR performs in case malware would use the same **User-Agent** as its victim, and even its language. A similar test was performed in [42]. We give the malware the exact features needed to bypass our browser similarity check. We have modified all malicious requests in the DEM and RAN logs by substituting the original **User-Agent** value with the one of the VM browser, which in this case was the victim. We did the same for the **Accept-Language** header, and if it was not present we have injected it in the log. We have tested both DECANTeR and DUMONT, and again for fair comparison we considered the requests.

The results shown in Table 3.4 ironically shows an increase in detection for DECANTeR, even though one may expect a drop of detection. The reason lies in the labeling method. Although part of the content of the message is exactly the same as the real browser, malicious requests are still labeled as background, because they do not create a referrer graph. Moreover, since they all share the same **User-Agent**, their amount of outgoing information adds up and it always exceeds σ . The results also show that DECANTeR is more robust than DUMONT against these simple evasion attempts.

3.6 Discussion and Limitations

In this section we discuss the application of DECANter in the specific use-case of data exfiltration, how an attacker can evade DECANter and the limitations of our fingerprinting approach.

3.6.1 Use-case: Data Exfiltration

DECANter has detected 96.8% of data exfiltration malware and also detected a tool specialized in data exfiltration (i.e., Data Exfiltration Toolkit)⁹. We believe DECANter is a good fit for detecting data exfiltration because the detection works independently from the content of the communication (i.e., payload), which is often obfuscated by attackers (e.g., using steganography [110]) as main mechanism to avoid detection over the network.

From the network perspective it is impossible to determine whether a specific communication contains sensitive data when it is obfuscated, unless the obfuscation scheme, or decryption key, is known. This is the main reason why current Data Loss Prevention (DLP) solutions [111]–[116] fail to detect data exfiltration.

DLP systems work as follows: 1) someone (e.g., security operator) collects documents that are known to be sensitive for the company (i.e., employees make assumptions about documents' sensitivity); 2) sensitive documents are provided as input to the DLPs; 3) depending on the type of DLP solution, the system learns the characteristics of a sensitive document (e.g., hash files); and 4) the DLP monitors the network and blocks connections that are transferring sensitive files. Unfortunately, when an attacker transmits sensitive files, she transforms them such that are not recognizable anymore. For example, the attacker can divide documents into small files and obfuscate each of them separately. Thus, when the documents are transferred over the network, the DLP is not able to re-identify the sensitive documents, because they have been transformed and they look very different from the original document.

Approaches that try to detect anomalous encrypted outbound communication [101], [117] also fail, because they rely on the high entropy values of encryption or compression. However, when malware uses encoding after encryption, the entropy drastically drops and the exfiltration is not detected.

Instead of focusing on the actual payload being transmitted, DECANter focuses on identifying anomalies in the metadata, which is described by the fingerprint features, and on quantifying the amount of data being transmitted. This combination has shown good performance by detecting 96.8% of information stealers samples, showing an improvement with the current state-of-the-art which detected only 40% of the samples.

3.6.2 Evasion

The evasion test has shown that DECANter is not easy to evade with simple evasion techniques such as spoofing **User-Agent** or other header values (see Sec-

⁹<https://github.com/sensepost/DET>

tion 3.5.5). However, it is still possible to evade DECANter.

The first type of evasion is to exploit σ and t , by communicating little data in each time slot without triggering the threshold, and using a non-browser **User-Agent** string. This type evasion is inherent in anomaly detection, because it relates to the thresholds and timeouts used for detection. It is still possible to reduce the risk of FNs by reducing σ and t , however this could lead to a larger number of FPs as well. Lastly, randomizing σ and t within a certain range of values can make the system less predictable to the attacker, and therefore may lead him to make mistakes that can cost him the detection. Nonetheless, the attacker can still evade using low values for σ and t .

The second type of evasion is more advanced. A malware can evade DECANter by mimicking the dynamic behavior of a browser and using the correct header values that are used by the victim's browser. Alternatively, the attacker may mimic an installed background application by mimicking its header values and message sizes. In both cases, the attacker's communication would show the same characteristics as the installed applications, therefore it would become impossible for DECANter to identify any difference. We call this type of threat an *adaptive attacker*. To the best of our knowledge, we are not aware of malware samples that use such type of mimicking techniques to evade detection. At most, we have observed that malware mimics the communication of an application *a priori*, meaning that it is programmed to communicate similarly to an existing application. However, it does not change its way of communicating *depending* on the victim's communication. Consequently, malware has to evolve from the techniques it uses today, if it wants to evade techniques such as passive application fingerprinting. An adaptive attack is quite advanced since the malware should be able to monitor the network of the victim, and to adapt its content according to live traffic details.

The third type of evasion is also advanced. An attacker can exploit an installed application, and use its network capabilities to communicate with its infrastructure. Using this technique the attacker is indistinguishable from the existing application, therefore the malicious communication is not detected. This type of attack is most common against browsers, and it is known as Man-In-The-Browser attack [118]. The attack could be performed against other applications as well. However, this type of attacks limits the attacker to compromise only what the application can access. In the case of browser, the attacker can read and write all the information a user inserts in the browser. However, the attacker cannot access files stored in the file system, therefore he cannot exfiltrate sensitive documents stored in dedicated folders.

Lastly, the attacker can try to *poison* the fingerprint update mechanism by convincing DECANter to update a benign fingerprint with a malicious one. For example, the malware can start generating HTTP requests with a **User-Agent** similar to the victim browser, and same language feature. This would trigger an update, and future connections of the malware will not be detected. However, DECANter can detect this by monitoring the old **User-Agent** string (before the update takes place). In case DECANter detects requests with the old **User-Agent**, then a poisoning attack has been detected. This works because, once it updates,

benign software does not switch back to the old value.

3.6.3 Fingerprinting Limitations

There are two main limitations of DECANTeR’s fingerprinting method.

The first limitation is encryption. DECANTeR’s fingerprinting method is designed to work on plaintext HTTP traffic. Consequently, unless a TLS-proxy is being used, HTTPS traffic cannot be analyzed using passive application fingerprinting.

The second main limitation relates to the way fingerprints are built. A central component in DECANTeR is the Referrer Graph, which tries to abstract the browser dynamics to distinguish between background and browser applications, by leveraging the requests generated by browsers to download the website resources. There are three main situations that are problematic to handle for the Referrer graph.

The first is when there are browser requests that do not contain the **Referer** field. Web scripts (e.g., asynchronous javascript) sometimes generate messages without such a header. OSCP requests never contains a **Referer**. Consequently, if they are observed while no other browser message has created a referrer graph, then the whole cluster would be considered a background application and a false positive would be generated. This is the most frequent situation of the three. DECANTeR tackles it using the update mechanism, which keeps the list of fingerprints updated, thereby suppressing future false positives.

The second problematic situation *may* occur when non-browser applications use the **Referer** field (e.g., cloud storage or chat clients). Do note that we did not find any example of such an application in our analyses, probably because they were encapsulated in TLS and we did not use a TLS proxy. In such cases, a Referrer Graph would be present and the application would be labeled as a browser. However, the fingerprint would be different from the browser’s fingerprint, due to the different header values. Nonetheless, this situation is not a major concern since such a fingerprint would be either added during the training phase, or after it is observed as FP. Therefore, it would not lead to many FP. In case a browser is used to upload data on a cloud storage service (e.g., Google Drive) or a messaging service (e.g., Facebook Chat), DECANTeR generates fingerprint that matches the browser fingerprint. This happens because the presence of **Referer** and **Origin** fields generate a Referrer Graph for the cluster, and user-agent and language match those of the browser. This can be checked using a tool that intercept HTTP/HTTPS requests, such as Burp Suite¹⁰.

The third situation that is hard to handle for DECANTeR is when websites are loaded only from the HTML. In this case a graph is not generated, because there are no extra resources being requested to load the website page. In case the browser has accessed only this “type” of websites within t , DECANTeR would generate a background fingerprint. Thus, DECANTeR would likely trigger an alert (i.e., FP) because, despite the low outgoing information, the fingerprint has a known browser **User-Agent**. However, it is more likely that, within t , the browser also

¹⁰<https://portswigger.net/burp/>

other websites with additional resources to download. In this case, there would be at least one graph for the cluster, and a browser label would be assigned to it. The disconnected nodes (e.g., the requests to websites without extra downloadable content) would be checked against the exfiltration filter. These requests probably do not show signs of exfiltration, therefore they are just assigned to the cluster previously labeled as browser.

3.7 Conclusions

We have proposed and implemented a new anomaly-based approach to detect anomalous outbound HTTP traffic. We have evaluated DECANTeR against traffic from a real organization, and from different sets of malicious traffic samples. We have compared our solution with the current state-of-the-art DUMONT, and we have shown that passive application fingerprinting performs better than existing solutions, both in terms of detection performance and evasion resistance. Moreover, we have shown that DECANTeR can successfully identify malicious communication that are known to be related to information stealers, which are malware samples with data exfiltration goals.

With this chapter we partially address our first research question **RQ1**. We have shown that passive application fingerprinting is a valid approach for data exfiltration detection without relying on predefined threat knowledge. Although the overall approach is valid, the fingerprinting technique of DECANTeR is technically limited, as discussed in Section 3.6.3. Therefore, we will continue to address **RQ1** in the next chapter, where we discuss a more technically advanced fingerprinting method that brings passive application fingerprinting a step closer to practice.

Chapter 4

Enhancing Passive Application Fingerprinting¹

In Chapter 3, we have shown that passive application fingerprinting is a promising approach for data exfiltration detection. However, as discussed in Section 3.6.3, the fingerprinting methodology used by DECANTeR suffers of some limitations. Therefore, to bring passive application fingerprinting one step closer to practice, a technical improvement is needed.

This chapter continue to address **RQ1**. We introduce a different fingerprinting technique, which can automatically infer the relevant characteristics of application traffic by only examining the protocol headers of its messages. For this reason, we name this technique: HEADPRINT. In contrast with DECANTeR’s fingerprinting method, which relies on few hand-picked features, HEADPRINT takes into consideration all message headers. More specifically, HEADPRINT generates fingerprints based on two different types of information: 1) the order of the header fields, and 2) the header values that are identified, using an entropy-based heuristic, as relevant characteristics for the observed application traffic. Our approach compares each network message to the learned fingerprints according to dedicated similarity functions. Finally, the score of the similarity functions is evaluated against a decision function, which determines whether a message belongs to a specific application or not. HEADPRINT optimizes the decision function by analyzing existing data. The automated discrimination of characterizing headers and the combination of two orthogonal message characteristics, namely the order of the headers and the values associated with them, represent the technical contribution of this chapter.

We implement HEADPRINT and we evaluate it in terms of fingerprinting accuracy, resilience to application variations (e.g., software updates), and detection performance in the setting of passive application fingerprinting for anomaly detection. We compare HEADPRINT with DECANTeR, showing a significant improvement in all aspects of the evaluation. On average, HEADPRINT shows an improvement

¹This chapter is an edited version of a paper that is currently under review at a peer-reviewed conference [2].

of 20% in accuracy and 30% in update resilience. Furthermore, our approach generates significantly fewer false alerts than DECANter, while still being able to correctly detect malicious HTTP communications. HEADPRINT relies on HTTP traffic characteristics, therefore it can be used to monitor TLS traffic in case TLS proxies are deployed within the monitored network.

4.1 Motivation

The small set of hand-picked features used by DECANter’s fingerprinting method intrinsically causes the following limitations: 1) low accuracy in identifying applications messages for which these features are not discriminative (e.g., web scripts in browser traffic); and 2) the fixed feature selection cannot deal with dynamic changes introduced by application updates. An example of the first limitation occurs when there exist HTTP messages without **Referrer** header, such as those generated by web scripts, as we discussed in Section 3.6.3. An example of the second limitation is the browser update, which changes at least the **User-Agent** value when it is updated. DECANter requires a dedicated heuristic to deal with browser updates (see Section 4.4.5), and it is not the fingerprinting method itself to directly tackle the issue. Moreover, background application updates are not taken into account.

DECANter overcomes these fingerprinting issues by assuming a human operator who manually and continuously monitors and updates the fingerprints. Unfortunately, this assumption is unrealistic in practice, because a security solution should not heavily rely on the maintenance from an operator in order to properly function.

For this specific reason, we decided to investigate a more accurate fingerprinting technique, which can also identify applications traffic after updates. The overall goal of this chapter is to provide a more precise solution that allows to use passive application fingerprinting for anomaly detection with less human involvement.

4.2 Intuition

The intuition behind HEADPRINT is that the network messages of an application share similar *header values*, and *header order* over time. In other words, we can distinguish applications by learning what values an application associates with each specific header (i.e., header values), and by learning in what order the headers are inserted in each application message. These characteristics complement each other: while the header values capture the semantics of the applications, the header order catches their implementation details.

We can *automatically* identify the distinguishing headers by identifying those header values that are being reused by the application. If a header value is often present in the messages of a specific application, then we can consider that header, and its values, to be representative for that specific application. On the other hand, in case header values change very often, it is unlikely that the information of that specific header can be useful to re-identify the application. For example,

for browsers the **Cookie** header is not representative because its values change depending on the visited webpage, however the **Accept-Language** value is often constant, and therefore it is more representative.

By systematically analyzing all the headers contained in applications' messages, we can create fingerprints that capture specific applications details. This is not the case for DECANter, which heavily relies on predefined headers.

4.3 HEADPRINT

In HEADPRINT, each application fingerprint is represented by two distinct models: a *header-value entropy* model and a *header sequence* model. The process of generating a fingerprint for an application a works as follows: (1) we collect the set of messages M generated by application a ; (2) we create from M the two different models VAL_a and SEQ_a , which represent the fingerprint of the application $F_a = (\text{VAL}_a, \text{SEQ}_a)$.

Once the fingerprint is generated, we can evaluate whether a new message t has been generated by application a by comparing t with the models of F_a . This process is performed in two steps. First, we evaluate how similar the header values and the header sequence t are with the respective models VAL_a and SEQ_a in F_a . We achieve this by using tailored similarity functions, which output scores between 0 and 1 (= identical). Second, a decision function d checks whether both scores are above (automatically determined) thresholds. If so, it attributes the message t to the application a .

The details regarding the fingerprint generation and the similarity functions for the two models are discussed in Sections 4.3.1 and 4.3.2. We discuss how to choose the decision function d in Section 4.3.3.

4.3.1 Header-Value Entropy Model

The header-value entropy model VAL_a for application a consists of a set of (h, V) tuples, where h is a header field and V is the multiset of values assigned to h , for all messages M . The underlying assumption for this model is that an application a generates messages such that the values corresponding to certain headers often reoccur. Hence, we expect that a new message t from application a also contains header values that have previously been observed, and are therefore contained in VAL_a .

Unfortunately, not all header values exhibit recurring behavior. When a header field value changes very often, it is likely that a new message from the same application will also contain a different value for the same header field. Non-recurring headers are not characteristic for an application, therefore the header-value entropy model identifies and discards them. In order to recognize if a header h and its corresponding values V are characteristic for the application, we compute the *entropy* $H(\cdot)$ for the values in V . We use entropy because it is a measure for *unpredictability*. If the entropy is low, the values in V can be considered “predictable,” meaning that they are likely to reoccur over time, and that the tuple (h, V) is considered characterizing for the application. In case of high entropy values, the

tuple is discarded. Consequently, we define VAL_a as the set of (h, V) tuples, where $H(V) \leq \alpha$, and α is a threshold that defines the highest entropy value V can have to be considered “predictable”. Figure 4.3 shows how HTTP headers are used differently across applications.

To check whether a new message t has similar values to application a , we evaluate $\text{value_sim}(t, \text{VAL}_a) \in [0, 1]$, which is a similarity function for categorical data. It evaluates how similar the values in message t are to the historical messages of application a (i.e., represented by VAL_a). The function identifies the subset of headers $\{h_i\}$ that represents the intersection between the headers in t and in VAL_a . For each header in $\{h_i\}$, if the value corresponding to h_i in message t is present in the historical values V_i where $(h_i, V_i) \in \text{VAL}_a$, then we consider the value to be a *match*. The similarity score is the number of matches divided by the number of headers in $\{h_i\}$. This technique for computing the similarity for categorical data is called Overlap [119].

4.3.2 Header Sequence Model

The header sequence model SEQ_a consists of the set of header sequences of all messages M observed for application a , where a header sequence is defined as the list of headers occurring in a message. The underlying assumption of this model is that application a uses a specific order of headers when it generates its messages. Therefore, we expect that a new message t from application a uses a similar (if not the same) header sequence that has previously been observed from a .

Although the majority of applications use a fixed set of header sequences for their communications, this is not true for all applications. For instance, the header sequences of browser messages sometimes depend on third-party JavaScript code, which injects additional headers, thereby making it hard (if not impossible) to find an exact match. Nonetheless, most of the time these new header sequences are still similar with those previously observed from the same application. For this reason, the concept of similarity for the order of the headers is effective.

To evaluate whether a new message t has a similar header sequence to application a , we compute $\text{seq_sim}(t, \text{SEQ}_a) \in [0, 1]$, a function that computes a similarity score between the header sequence of t and each sequence in SEQ_a . We use the sequence similarity metric proposed by Bakkellund [120], which is based on the *longest common subsequence* (LCS) problem. The similarity score is defined as the length of the LCS between the two sequences, divided by the length of the longest sequence. In other words, the larger the longest common subsequence two sequences, the more similar the sequences are. After we compute the scores, we return the maximum. We choose to rely on LCS because subsequences are not required to occupy consecutive positions within the original sequences, while this is not the case for the longest common substring problem. It is difficult to predict where new headers can be observed in a header sequence, because it may depend on specific web application scripts. Therefore, LCS can still identify two messages as similar despite new headers being inserted, because previously known headers occur with the same relative order, thereby generating a long subsequence.

Figure 4.1: Distribution of messages after they have been compared with the fingerprint of Firefox. Crosses represent messages from applications other than Firefox. Dots represent messages from Firefox. The straight and dashed lines represent two potential classifiers to distinguish when a message is generated by Firefox or not.

4.3.3 Decision Function

When we compare a message t with a fingerprint F_a , we first compute two distinct scores x and y , which are the output of $value_sim()$ and $seq_sim()$, respectively. These scores are then evaluated by a decision function $d(x, y) \in \{0, 1\}$, which determines whether t is a message of application a or not.

In other words, d is a binary classifier for a two dimensional plane, where $value_sim()$ and $seq_sim()$ similarity scores represent the axes (i.e., x and y , respectively). The classifier decides whether the point (x, y) is in an area of the plane where messages are considered to match the fingerprint, or not. Figure 4.1 shows the distribution of the scores of messages when compared with the fingerprint of an instance of Firefox. The green crosses represent messages from all the applications installed on the host that *are not* Firefox, while the blue circles represent messages from Firefox. When t is compared with the fingerprint of its application, we expect a point in the top-right part of the plane as both similarity functions should yield scores approaching to 1. Assuming that the models underlying the fingerprint of most applications differ from each other, a high similarity score is only produced when a message is compared to the fingerprint of the application it originated from. This observation suggests that we can create a *general* decision function d for all fingerprints.

As shown in Figure 4.1, there can be different ways to determine a decision function d . In this work, we propose and evaluate two different approaches:

- Static-rule: we use a threshold-based decision function (i.e., similar to $d1()$ in Figure 4.1) that we define according to our domain knowledge.

- ML-classifier: we train a classifier using machine learning (e.g., similar to $d2()$ in Figure 4.1).

Choosing the Decision Function.

We can define a static rule as a decision function because messages receive a high similarity score when evaluated against their application fingerprint and low otherwise, thereby creating a predictable distinction. In this work, we chose to evaluate threshold-based rules that consider a similarity score (x, y) to be a match if $x > X$ and $y > Y$, where X and Y are two thresholds.

Although a predetermined function has the advantage of neither requiring a learning phase nor being computationally expensive, it is likely a sub-optimal solution in terms of classification performance. At the expenses of an ad-hoc learning phase, machine learning can be used to learn a decision functions from the data, which likely provides a more precise solution. However, to properly learn a decision function, we need to obtain a representative dataset for a binary classification problem, where the positive class represents the scores of messages compared with their application fingerprint, and the negative class represents the score of messages compared with other applications fingerprints.

The generation of such dataset is independent from the fingerprinting approach itself, and it can be achieved in two different ways, depending on the available resources. One method is to generate a dataset by analyzing many applications offline. Although this setup can provide a precisely labeled dataset, it can require a lot of resources. Alternatively, the dataset can be built by monitoring the traffic for a set of hosts. This setup requires fewer resources, and it can contain data contextualized to the network where the system would ultimately be deployed, but the dataset is unlabeled. In this case the **User-Agent** can be used as groundtruth, since it commonly represents a unique string identifying a specific application. Consequently, when a message is compared with a fingerprint having a *similar* **User-Agent** value (e.g., variations of version should be taken into account due to updates), the similarity score (x, y) is labeled with 1 (positive class), and 0 otherwise (negative class). In this work, we use the latter approach.

In Section 4.3.5 we discuss the implementation details regarding the chosen thresholds and machine learning algorithms, and how we generated a dataset to train the decision functions for machine learning algorithms.

4.3.4 HEADPRINT for Anomaly Detection

We model the machine’s normal communications by identifying and fingerprinting the applications installed on it. The setting works as follows: (1) during a training phase, we passively learn the fingerprints of applications installed on a machine; (2) during testing phase (i.e., when the system is live) we verify if each new message matches with any known fingerprint, and, in case no match is found, the message is considered *suspicious*, since it is generated from an *unknown* application. Please note that the *training* is always passive, therefore the fingerprints are generated by analyzing only the network traffic observed from a host. HEADPRINT does not

Figure 4.2: Overview of HEADPRINT anomaly detection. First, in the training phase, we leverage our core fingerprinting approach to passively generate application fingerprints. Then, in the testing phase, we compare new messages with known fingerprints. If they do not match any fingerprint, we check whether they transmit a certain amount of data (more than T) and raise alerts accordingly.

require access to the operating system or the application software installed on each host.

In this work, we are interested in using application fingerprinting to detect data exfiltration attempts. Therefore, we do not want to trigger alerts for each suspicious message, but rather when the amount of data transmitted by suspicious messages, which share the same destination, exceeds a certain threshold T . The overview of the testing phase is shown in Figure 4.2. The threat model for HEADPRINT is the same as the one discussed in Section 3.3.

Fingerprint Generation

We assume the training phase to be *trusted*, meaning we expect that machines are not compromised during the training period. In practice, the training phase can be assisted by deployed security tools, which can help identify malicious hosts, thereby avoiding the collection of fingerprints for known malware communication. The assumption of a clean training phase is common among anomaly detection approaches. We made the same assumption in Chapter 3.

Fingerprint Convergence. A distinctive aspect of HEADPRINT is how it automatically determines the required amount of data needed to train a certain application’s fingerprint. Intuitively, the amount of training data affects the robustness of our fingerprints in terms of fingerprinting accuracy. However, this parameter strongly depends on the complexity of the application that HEADPRINT fingerprints. For instance, some applications always produce the same network traffic, and hence can be accurately fingerprinted after a few messages, while others (e.g., browsers) might have a very dynamic network behavior. Therefore, similarly to [121], HEADPRINT divides the training phase in intervals, repeatedly tests the

generated fingerprints after each interval, and stops the training when new training data does not add information to the fingerprint. We call this a *converged state*. In practice, we achieve this by testing the freshly generated fingerprints at intermediate steps, HEADPRINT can understand when it collected enough data to model the application’s network behavior. In practice, we say that an application reaches convergence when we do not see any prediction mistake for its fingerprints for K consecutive intervals.

More precisely, HEADPRINT performs the following steps: (1) collecting the data of an application for a certain time interval i ; (2) splitting the collected data in training and testing datasets using 70-30 split while preserving the chronological order of the requests; (3) training the fingerprint and tests it; (4) if there are no prediction mistakes a success counter sc is increased by 1, otherwise sc is set to 0; (5) moving to the next interval $i + 1$; (6) finally, when sc reaches a convergence threshold K (i.e., the minimum number of consecutive successful iterations), the system stops the training and returns the generated fingerprints for that application. The data collected in each interval is aggregated with the previously collected data from other intervals. Also, because some applications might never reach convergence, we set a maximum number of intervals.

Training Fingerprints. HEADPRINT monitors host traffic and it clusters web requests with “similar” **User-Agent** values, and it applies the aforementioned training process to each cluster. **User-Agent** values are commonly used by benign applications as unique identifiers to be correctly recognized by web servers. Since we assume the training phase to be trusted, no application would actively try to mimic other applications values. Thus, it is likely that requests generated by the same application, within a specific time interval (e.g., training period), have the same, or similar (e.g., version number increased) **User-Agent** value. The training process stops when a fingerprint $F_{a_i} = (\text{VAL}_{a_i}, \text{SEQ}_{a_i})$ is trained for each cluster. The set $F = \{F_{a_i}\}$ represents the set of fingerprints of a machine. HEADPRINT requires a specific training timeout that forces it to stop generating new clusters, otherwise, in the unlikely case new applications are regularly installed, HEADPRINT would keep producing new clusters, thereby never ending the training. We set this timeout to 5 days.

Testing New Messages. In anomaly detection we are interested in identifying messages originating from unknown applications. We achieve this by comparing each new message t against all fingerprints in F . If t does not match with any fingerprint, then we consider t to be generated by an unknown application. From now on we refer to these messages as *suspicious* messages. The comparison between a message t and a fingerprint $F_{a_i} \in F$ is achieved by computing $\text{test}(t, F_{a_i}) \in \{\text{True}, \text{False}\}$. The function returns True only if the decision function d , as defined in Section 4.3.3, returns 1 for the similarity scores computed from t and F_{a_i} .

Alerts. Due to the high volume of HTTP traffic, and its heterogeneity, the system triggers an alert only when it observes suspicious messages transmitting an amount of data greater than a threshold T within a certain period of time (e.g.,

five days), which may represent an attempt of a newly installed application (e.g., malware) to exfiltrate data. Thus, suspicious messages are aggregated per destination into clusters, one representing the domain name (i.e., second and top level domains) and the other one representing the destination IP. Domain names are identified through the `Host` header field. Each cluster has an attribute describing the amount of data transmitted toward the destination by the messages within the cluster. Every time a new message is added to the cluster, we update the data transmitted with the minimum edit distance between the new message and the previous messages. This way of computing outgoing information is discussed in Section 3.4.1. Once the data transmitted exceeds a threshold T , an alert is generated containing all the messages as evidence. To avoid duplicated alerts, we verify if the new alerts share messages with previous alerts. If so, we merge the two alerts together.

It is important to note that the same message is assigned to two distinct clusters (i.e., one for domain and one for IP address). The reasoning behind this seemingly redundant check is the following: if we aggregated messages only based on IP addresses, attackers may use known techniques, such as Fast-flux [122], to avoid detection by communicating with the same domain over multiple IPs. Similarly, if we used the domain as a single aggregation parameter, the attacker could spoof different domains in order to exfiltrate data towards the same destination IP. The redundant checks make it harder for attackers to exfiltrate data without being detected, because it forces them to use multiple domains and IPs.

4.3.5 Implementation

In this section, we describe the implementation details of HEADPRINT, such as how we chose the decision functions based on thresholds and machine learning. We implemented HEADPRINT entirely in Python.

Threshold-based Decision Functions. We know that (most of the time) a message obtains a high similarity score when it is tested with the fingerprint of the same application. However, a perfect match does not always occur, because messages may use different headers or values over time. Moreover, we observed in our data that scores from the header-value entropy model are usually lower than the header sequence model. After this observation, we decided to verify the performance of HEADPRINT using two different threshold-based decision functions: 1) the first uses 0.7 as value for both thresholds; 2) the second uses 0.65 for the header value scores and 0.75 for the header sequence score. These values have been chosen after an empirical evaluation on a small subset of data, and they allow us to compare the differences in classification performance by using equal thresholds for both scores, and two different values where the header-value score has a lower threshold.

Machine Learning-based Decision Functions. We followed the steps in Section 4.3.3 to create a training dataset to train different classifiers, which we use to evaluate our fingerprinting approach.

More specifically, we did the following: (1) we randomly selected a subset of 60 hosts out of 302 from our organization dataset (described in Section 4.4.1), in order to obtain a representative dataset with a diversity of applications that might be analyzed by our tool after being deployed; (2) for each of these selected hosts, we used the first few days of data for training, following the same approach discussed in Section 4.3.4 to train the fingerprints (see the training setup in Section 4.4), and the remaining days of data for testing; (3) we tested each message against the host fingerprints, and we labeled the score (x, y) with 1 if the **User- Agent** value in the fingerprint is similar to the one in the message, and 0 otherwise²; (4) we grouped together all scores with label 1 and label 0, respectively; (5) we under-sampled the overly represented negative class, which had almost 20 times more datapoints than the positive class, by randomly picking samples with replacement using `RandomUnderSampler` from the `imblearn.under_sampling` Python library; (6) we used the obtained (balanced) dataset of 2M datapoints to train three different classifiers: Adaboost, `NearestNeighborsClassifier` (with $k=5$) and SGD³, using the `sci-kit` library [124].

We chose these classifiers because they cover both the linear and non-linear case, and they can be trained in a short amount of time despite millions of datapoints. Finally, we want to highlight that other classifiers can be used with HEADPRINT as well, as long as they can work with two numerical features.

4.4 Evaluation

In this section we evaluate our approach and compare it with DECANter. We describe the dataset used in our evaluation. We evaluate the accuracy of HEADPRINT and DECANter in fingerprinting applications, their resilience to application updates, and their detection performance in the context of anomaly detection.

4.4.1 Datasets

We evaluate our approach using a dataset, represented as a set of Zeek [109] HTTP logs, obtained from the network of an international organization. The logs contain only HTTP metadata, more precisely the HTTP headers without the body of the messages. Only outgoing traffic generated by clients in the network is included; server responses are not present. The dataset contains the network traffic of 302 hosts, from three different subnets, for a period of 40 days, for a total of 3.87 million HTTP requests. The collected traffic is generated by hosts with static IPs, in order to be able to correctly identify hosts over time. Hosts represent mostly workstations (no laptops) in the premises of the organization, and servers. Traffic has been captured on port 80 and towards external network addresses. In other

²We consider two values to be similar if the Python library function `difflib.SequenceMatcher().ratio` returns a value greater or equal than 0.9. The similarity function is an elaborated version of a pattern- matching algorithm based on the gestalt approach [123].

³In our evaluation we have used all default parameters of the `scikit` library.

words, HTTP traffic within internal services has been excluded. HTTP messages without headers are excluded from this evaluation.

4.4.2 Experiment Setup

To be able to evaluate the fingerprint accuracy and the resilience to update, we need a labeled dataset such that we can verify whether messages and fingerprints correspond to the same application. Considering the large size of the dataset, it was not practical to manually label each individual message. Moreover, we also could not install an agent on each machine to help labeling the data. Thus, we decided to consider a message and a fingerprint to belong to the same application, if their **User-Agent** values are similar. As discussed in Section 4.3.4, **User-Agent** values are often used as unique identifiers by benign applications. Thus, we can use these values to label the traffic of different applications.

In our experiments, we use the following HEADPRINT parameters. Regarding the training phase, we use a limit of 20 intervals of 6 hours each for the training (5 days in total), and a convergence threshold K equal to 3. We picked these values after a small-scale empirical evaluation, which showed that these parameters do not significantly affect the performance of our tool. Regarding the header-value entropy model, we performed some empirical tests to choose an α value that is representative for predictable header values. We tested different values and observed which headers were considered predictable. Using domain knowledge we decided to choose $\alpha = 1$, as it returned the most representative results. Overall, our approach has evaluated more than 3.4M messages, and used about 430k messages for training. Please note that in our evaluation HTTP **BODY**, **Method**, **Version** and **URI** are *not* considered HTTP headers.

To achieve a fair comparison, we compared HEADPRINT and DECANter in terms of the number of messages correctly classified by the fingerprinting method. Since DECANter evaluates fingerprints and not individual messages, if a fingerprint is considered a TP, then we consider the amount of messages observed by DECANter to create that specific fingerprint to be TP as well. We use the same method to compare DECANter with DUMONT in Section 3.5.

4.4.3 Characterizing Application Headers

Header Values. Applications associate different values to HTTP headers. Figure 4.3 shows how 10 distinct applications associate values differently to the headers. The heatmap shows the entropy of header values observed by applications (rows) for common HTTP headers (columns). Blank cells represent an entropy value of -1 , which means the header was not observed for that application. We can see that different applications use different headers and their values are more predictable (i.e., light cells) or less predictable (i.e., dark cells). HEADPRINT is capable of identifying these differences in headers usage, and it embeds their values in each application fingerprint, thereby achieving a more accurate characterization of the application.

Header values may change over time for some application. In case of software

Figure 4.3: Heatmap representing the entropy of header values for 10 popular HTTP header and 10 different applications, including five different browsers. White cells, where entropy is equal -1, mean that the header has never been used by the application. When entropy is equal 0, it means the application uses a constant value for the header.

updates, headers such as **User-Agent** can change values [125], [19]. These updates do not disrupt our approach, in case enough training data is collected, because most of the header values remain the same, yielding still a high similarity score. We discuss how our approach deals with software updates in more detail in Section 4.4.5.

Header Sequences. We observed that the majority of application fingerprints in our dataset have a single header sequence, which on average is composed of a small number of headers (i.e., around four or five). Another large amount of applications shows fewer than ten header sequences. Since HTTP has a limited set of commonly used headers, this can lead to potential *collisions*, meaning that two or more applications on the same machine share one or more header sequences. Indeed, we found cases of collision in our dataset. For example, the header sequence used by Thunderbird can also be found in Firefox traffic. Thus, header sequences cannot be used as a standalone technique to distinguish applications, but we combined it with header values, which leverage other parts of the message.

Collisions are rare among browsers, despite the fact that their fingerprints are characterized by a large number of header sequences. This has two reasons: few browsers are installed on each host, and browsers from different vendors use standard headers (e.g., **Host**, **User-Agent**, **Connection**, **Accept**) in different orders. This makes header sequences a good metric to distinguish browsers' messages. The reason behind the large number of sequences is twofold: (1) browsers implement complex functionalities, uncommon for other applications, which require special HTTP headers, and (2) web pages can influence (e.g., via JavaScript) the headers of browser messages.

Fortunately, these header sequence changes do not disrupt our approach because, as Figure 4.4 shows, the accumulated amount of unforeseen header sequences

Figure 4.4: Example of how often browsers generate new header sequences.

for new browser requests shows a logarithmic behavior, meaning that the more the browser communicates, the rarer it becomes to find new header sequences. Moreover, we observed that once enough samples are included in the fingerprint generation, new header sequences are often evaluated with high similarity scores.

4.4.4 Fingerprinting Accuracy

In this section we evaluate the accuracy of the fingerprinting mechanisms used in HEADPRINT and DECANter. For a fair comparison, we assume DECANter does not rely on the operator to update the fingerprints.

We consider a message to be: *true positive* (TP) if the returned fingerprint with maximum score corresponds with the application generating the message; *false positive* (FP), if the returned fingerprint does not correspond with the application generating the message; *false negative* (FN) if no fingerprint is returned (i.e., the decision functions identified all similarity scores to be too low to match any fingerprint), but the fingerprint of the originating application is known; and *true negative* (TN) if no fingerprint is returned, and indeed there is no fingerprint for the originating application because it is unknown (i.e., not present in the training).

HEADPRINT evaluates a message against multiple fingerprints and returns only one fingerprint. The system is accurate if it returns the fingerprint of the correct originating application. Thus, HEADPRINT can be adjusted to return only the fingerprint that yielded the highest score. Since (1,1) is the maximum score possible, the highest similarity score (x, y) can be defined as the closest point (e.g., using the Euclidean distance) to (1,1). On the other hand, DECANter evaluates newly observed fingerprints against the trained fingerprints. The system is accurate if it matches the new fingerprint with the highest similarity score to the fingerprint of

Decision Function	TP	TN	FP	FN	Precision [%]	Recall [%]	Accuracy [%]
Threshold 0.7-0.7	2,891,064	217,382	21,064	284,835	99.20	91.03	91.04
Threshold 0.65-0.75	2,969,353	213,249	24,196	207,546	99.19	93.46	93.21
Adaboost	3,052,589	206,104	34,065	122,242	98.89	96.16	95.44
NearestNeighbor5	2,913,111	207,668	45,834	247,731	98.45	92.16	91.40
SGDHingeL2	2,879,448	218,954	20,045	295,897	99.30	90.68	90.74
DECANTeR	2,170,663	217,232	8,143	784,014	99.62	73.47	75.09

Table 4.1: Classification performance of HEADPRINT and DECANTeR for the generic use-case of application fingerprinting, where we measured if messages were correctly associated with their application fingerprint. The differences in the total amount of processed messages is caused by the autoconvergence of HEADPRINT. HEADPRINT stops the fingerprint training once the fingerprint seems to perform well, thus it starts “deploying” the fingerprint earlier than DECANTeR, which rely on a fixed period of time.

the originating application.

Table 4.1 shows the classification performance of both HEADPRINT, using different classifiers, and DECANter. Independently from the classifier used, HEADPRINT outperforms DECANter in terms of recall and accuracy. The recall measures how many of the requests, for which an application fingerprint exists, have been correctly identified. The results show that DECANter, without the help of an operator, cannot correctly associate messages to their application. This is a consequence of the strict and ad-hoc fingerprinting mechanism that cannot correctly handle application variations.

On the other hand, HEADPRINT *without any update* (human or automated) shows an accuracy above 90% for all classifiers, and for Adaboost, which is the best performer, the accuracy is 95.44%. All classifiers show very high precision, meaning that, when they associate a message to a fingerprint, they associate it to the correct one in the 98-99% of the cases. The recall is generally lower, because the classifiers sometimes do not associate any fingerprint to messages, although their fingerprint exists. Software updates are one of the causes of lower recall. Additionally, by using autoconvergence in its training phase, HEADPRINT uses less data than DECANter during training, thereby making the fingerprint mechanism also more efficient.

Overall, Adaboost performs better than other classifiers. This confirms our hypothesis that machine learning decision functions can be more precise than threshold-based ones. Adaboost identifies non-linear patterns that help to optimize the classification performance, and this is not possible to do with the threshold-based functions. However, threshold-based classification still shows good results compared with DECANter. For threshold-based classifiers we observe that a lower score for the header-value entropy model increases the recall. Therefore, HEADPRINT can find a match more easily. On the other hand, the precision is lower, because a lower thresholds allows messages of other applications, that may be similar, to be misclassified. The reason of a lower accuracy for the threshold (0.7,0.7) is the large number of browser requests. Browsers are updated over time, which lowers the score of the header-value model (as discussed in Section 4.3.5). A high threshold for header values may misclassify the requests as a non-match, while a higher threshold for header-sequences affect less the results.

These results suggest that the models used by HEADPRINT are capable of capturing distinguishing characteristics of application messages. Moreover, the results show that it is possible to accurately fingerprint applications without actively maintaining the system.

4.4.5 Update Resilience.

In this experiment we evaluate how HEADPRINT and DECANter perform in case of application updates, where the traffic of applications changes over time [125], [19]. We considered only those applications that did not contain an update during their fingerprint generation, but only during testing. We verified for all messages having **User-Agent** value different (e.g., increased version number) than the one used in training, whether they were correctly assigned to their application finger-

Decision Functions	Correct Matches [%]
Threshold (0.7,0.7)	75.34
Threshold (0.65,0.75)	84.20
Adaboost	92.00
NearestNeighbors5	77.64
SGDHingeL2	74.58
DECANTeR	54.32

Table 4.2: Percentage of updated application messages that successfully match with the fingerprint of their originating application.

print or not. The total number of messages in our dataset generated by updated application is 962,915.

Table 4.2 shows the percentage of updated application messages that are correctly matched with the fingerprint of their originating application. Due to the predefined features used in its fingerprinting method, DECANTeR is not capable of identifying applications after they are updated, unless there is human intervention. This result highlights even further the limitations of DECANTeR to changes. On the other hand, HEADPRINT shows better performance. Adaboost is the best classifier, and it correctly fingerprints more than 92% of the messages. Stricter functions that focus only on high scores, such as SGD, are more penalized in this scenario. Threshold mechanisms perform better when the threshold for *value_sim* is lower, because they are more “flexible” toward header value changes, which can be caused by updates.

The reason why HEADPRINT is resilient to updates is shown in Figure 4.5. The consequences of software update are observed in lower *value_sim* scores (x axis), where the red circles never reach 1.0. This happens because the update permanently changes *some* of the header values that are always in the application messages. The *value_score* does not completely drop because several headers continue to show the same values. More importantly, updates do not seem to affect the *sequence_sim* score (y axis), which still yields the same high similarity scores as in the messages before the update. This allows HEADPRINT to be update resistant. It correctly identifies update application messages, because the similarity score remain in top-right area of the plane, and the decision functions can make the correct decision.

To the best of our knowledge, HEADPRINT is the first fingerprinting approach that addresses software updates (including browser and background applications).

Updating HEADPRINT’s Fingerprints. The fingerprints of HEADPRINT can be updated over time, thereby improving their accuracy and resilience against traffic variations. Whenever a message is considered similar to a fingerprint, without showing a perfect similarity score (i.e., 1 for both scores), we can include the header values and their order in the new models, and regenerate the fingerprint. This operation can be performed periodically after a certain numbers of similar messages has been found, or after a specific period of time. Since we do not know

Figure 4.5: Similarity scores in case of browser update.

how much an application behavior changes over time (e.g., after 10 or 20 updates), it can be a good idea to periodically update the fingerprints.

However, we decided not to investigate such updating mechanism, because HEADPRINT is used in adversarial settings. Thus, an attacker can try to abuse weaknesses in the updating process in order to ensure her evasion. We consider the design of a secure updating mechanism, for a detection model in adversarial setting, as a separate research question. Nonetheless, it is still possible to use the aforementioned process to manually update the fingerprints.

4.4.6 Detection Capabilities

In this evaluation we compare the detection capabilities of the two approaches: HEADPRINT and DECANter, and we assume DECANter uses the help of the operator to update the system. We analyze our dataset using our implementation of HEADPRINT, and the publicly available implementation of DECANter. We have then manually analyzed, with the help of third-party threat intelligence, the alerts generated by the two systems. Both DECANter and HEADPRINT (see Section 4.3.4) rely on a threshold-based system to trigger alerts. DECANter applies it to each anomalous fingerprints, while HEADPRINT applies the threshold to the amount of data transmitted by anomalous messages towards a specific application within a week of time. We evaluate both systems according to different threshold values to understand their effect on the false positive rate.

Alert Thresholds. Table 4.3 shows the number of alerts we obtained for the different classifiers we tested, and for different thresholds. The results show that for HEADPRINT, the higher the threshold the smaller is the number of alerts. Adaboost shows better performance than other classifiers. For Adaboost, the lowest threshold triggers 2,355 alerts, which represents 1.04% of all messages. With higher

Decision Function	Alert Thresholds (kB)							
	1	2.5	5	7.5	10	20	50	100
Threshold (0.7,0.7)	4343	1428	746	476	346	154	44	18
Threshold (0.65,0.75)	3296	1117	580	354	262	113	33	12
Adaboost	2355	721	345	212	157	67	17	9
NearestNeigh.5	4218	1288	588	375	270	109	30	11
SGDHingeL2	4543	1474	759	486	350	153	46	17
DECANTeR	2861	2739	2666	2609	2580	2544	2512	2486

Table 4.3: Number of alerts generated by different classifiers according to different alert thresholds.

exfiltration thresholds, such as 5,000 bytes or 7,500 bytes, the number of alerts decreases to 345 ($\sim 0.30\%$) and 212 ($\sim 0.20\%$), respectively. This trend holds for all classifiers. Figure 4.6 shows that the false positive rate is below 0.5% for rather small thresholds. For example, if we consider Adaboost with a threshold of 5KB for five days of exfiltration, we would obtain 345 alerts over 35 days, which means 10 alerts per day on average. With regards to data exfiltration, these thresholds are very low, if we consider that a single document transfer involves few hundreds Kilobytes. Smaller and more appropriate thresholds for data exfiltration, such as 50Kb or 100Kb, produce a smaller number of alerts such as 17 ($\sim 0.05\%$) and 9 ($\sim 0.01\%$), respectively.

The threshold does not affect as much the number of alerts generated by DECANTeR. This occurs because in case an anomalous fingerprint contains a **User-Agent** value that contains browser strings (e.g., ‘Firefox’), DECANTeR triggers an alert independently from the threshold. Table 4.3 shows that even with considerable human effort in updating the system, DECANTeR generates more alerts compared with HEADPRINT. Similarly, the false positive rate in terms of requests is significantly higher for DECANTeR than in HEADPRINT.

Detection of Infected Hosts. In this analysis we manually inspected the alerts generated by HEADPRINT and DECANTeR. For HEADPRINT we analyzed the scenario where it uses Adaboost as classifier and a threshold of 5KB. We opted for this threshold because it is low enough to potentially detect also prolonged C&C communications, where the malware communicates with its infrastructure without necessarily exfiltrating data. Furthermore, 5KB would also allow us to identify data exfiltration attempts.

To check whether our alerts actually came from infected machines, we leveraged two sources of threat intelligence: VirusTotal and a third-party company involved in the threat intelligence business. The former did not show any malicious behavior, whereas the latter provided us with 18 indicators of compromise for our dataset. Please note that an indicator of compromise does not mean that the URL itself is malicious. For example, checkip.dyndns.org/ is often contacted by malware to verify the IP of their victim, but the service is not malicious.

Figure 4.6: False positive rate (FPR) according to the number of alerts triggered by different thresholds. FPR represents the percentage of messages analyzed that are contained in the false alerts.

The 18 indicators provided by the third-party company involved requests generated by 13 different hosts. For 5 of these hosts we were able to confirm that the indicators were false positives. For the other 8 hosts, we inspected the alerts generated by both HEADPRINT and DECANter. For HEADPRINT we found evidence of suspicious behavior of 2 of these hosts, while with DECANter we found 3 suspicious hosts. These hosts generated traffic associated with browser hijackers or generic adware. The alerts included HTTP requests transmitting several thousands bytes encoded in the URI. For the 5-6 remaining hosts, we did not find any evidence of suspicious behavior. After analyzing the indicators, we inspected all the remaining alerts. For HEADPRINT we identified 3 other hosts showing suspicious connections, which did not show suspicious behavior according to the indicators. One of these hosts showed highly suspicious behavior (i.e., unknown software uploading 50Kb-100Kb of data toward cloud services), which merits extra investigation. On the other hand, DECANter found 5 additional hosts showing anomalous connections, such as sequences of POSTs to a suspicious domain and IP, and large quantities of anomalous fingerprints from the same host but with many different (and inconsistent) **User-Agent** values.

Overall, HEADPRINT is capable of identifying anomalous HTTP traffic, similar to DECANter. However, HEADPRINT achieves this result without requiring any human intervention and by raising much fewer alerts. *This difference is caused by the more accurate and update-resilient fingerprinting technique used in HEADPRINT.*

4.4.7 Detection of Mimicry Attempts

To avoid detection, malware needs to generate messages with high similarity scores for both the header-value and header-sequence models of an existing fingerprint. If the malware does not mimic an existing application, it is likely to be detected as shown in Chapter 3.

We consider the scenario where a malware generates network messages following a predefined format that correctly uses header sequences and values of a known application. We assume the malware *does not adapt the sequences and values to the traffic of its victim* (i.e., the malware remains non-adaptive). The scenario for adaptive malware involves a more advanced mimicry technique and we discuss it in Section 4.5.

Therefore, malicious messages contain header values that are common for each specific header (e.g., **User-Agent** string that the malware author has observed as common values from an existing application). Since our implementation of HEADPRINT worked with log files, we generate messages containing header sequences and values that were observed in the default installation of known browser. We compare the mimicry messages against the fingerprints of each host, present in our dataset, and we consider the mimicry to be successful if the message matches with at least one fingerprint. Lastly, we compute the percentage of hosts for which HEADPRINT would have been able to identify the mimicry attempt (i.e., the mimicry message does not match any fingerprint). We assume malware tries to mimic browser messages, since they are common applications, and they generate heterogeneous traffic.

Table 4.4 shows how HEADPRINT is capable of identifying mimicry attempts. First of all, even if the mimicked message represents exactly the format of a known application, if the application is not installed, a match cannot be found. Second, the values used in the mimicked message headers may differ from those used by the same browser of the compromised system. For example, the malware may use an **ACCEPT-LANGUAGE** value that is uncommon in victim’s country, or the user runs an old version of an application, or user applications update over time changing its values.

These value changes decrease the similarity scores making it harder to find a correct matches, exposing the mimicry attack to detection. Consequently, “stricter” models (e.g., Threshold (0.7,0.7)) are less vulnerable to these mimicry attempts, as show in 4.4. Thus, there is a tradeoff to consider when a classifier is chosen for fingerprinting, because accurate classifiers are more exposed to evasion. The results in Table 4.4 shows that HEADPRINT is not easily bypassed by a mimicry attempt. In Section 4.5, we discuss possible evasion techniques an attacker may used to avoid detection.

4.5 Limitations

HEADPRINT overcomes some of the DECANTeR fingerprinting limitations discussed in Section 3.6.3, because HEADPRINT does not rely on specific header such as the **Referer** header. However, other limitations have not been addressed. First

Classifiers	Mimicry Failure [%]		
	Chrome	Firefox	Safari
Threshold (0.7,0.7)	97.68	87.09	80.46
Threshold (0.65,0.75)	74.83	80.13	74.83
Adaboost	76.82	64.24	57.95
NearestNeighbors5	76.82	67.22	59.93
SGDHingeL2	98.68	85.76	80.46

Table 4.4: Percentage of hosts where a mimicry attempt would fail, according to different applications.

of all, HEADPRINT does not work on encrypted data, because the headers and their values cannot be observed. Therefore, HEADPRINT can work with the TLS protocol only in case a TLS proxy is deployed in the network. This is a setting that can occur in an enterprise. It *may* also be possible to apply HEADPRINT to the unencrypted part of TLS (i.e., TLS Handshake Protocol) in order to model applications. The study of HEADPRINT applicability to the TLS handshake protocol, or the design of new methods to fingerprint applications over encrypted data, is considered future work. Second, also HEADPRINT does not detect advanced mimicking attempts, where a malware reads the traffic of the compromised system and generates messages that mimic an installed application. We investigate the effects of this threat in Chapter 5. Lastly, an attacker can try to evade HEADPRINT by sending an amount of data to each destination that does not exceed the threshold to trigger an alert.

4.6 Conclusions

In this chapter we proposed HEADPRINT, a technique for passive application fingerprinting, which examines protocol headers to differentiate between various applications. Our results show that HEADPRINT achieves 95.44% accuracy in correctly identifying application messages, while being resilient to minor variations like software updates, and coping with different versions of the same application. Outperforming the fingerprinting method proposed in Chapter 3, the accuracy of HEADPRINT also positively affects its performance in the setting of anomaly detection, showing its capabilities in detecting malicious communications, while raising significantly fewer false alerts. As a consequence of better performance, HEADPRINT also does not require as much operator attention as DECANter does. In fact, all the results discussed in the evaluation assume no human involvement. HEADPRINT requires an operator attention whenever a new application is installed and a new fingerprint needs to be added. Lastly, despite the fact that HEADPRINT has been evaluated on HTTP, its underlying fingerprinting technique is protocol agnostic and may be applied to other protocols as well, such as SMTP, TLS Handshake Protocol, etc. We consider the applicability of HEADPRINT to other protocols as future work.

With this chapter and Chapter 3, we address **RQ1**. In this chapter we have shown that it is possible to precisely fingerprint applications by analyzing the headers of their messages. More specifically, the order of the headers and the header values seems to provide a significant amount of information to distinguish different applications. Although our fingerprinting approach brings application fingerprinting one step closer to practice, there are more challenges that must be addressed in order to consider application fingerprinting to be practical in the context of anomaly detection. These additional challenges are discussed in Chapter 8, and are not addressed in this thesis. However, we make further contributions for data exfiltration detection with Chapter 5 and Chapter 6, where we discuss the threat of adaptive data exfiltration and how it can be detected, thereby addressing **RQ2**.

Chapter 5

Victim-Aware Adaptive Data Exfiltration¹

In Chapters 3 and 4, we discussed how it is possible to detect data exfiltration attempts. As any anomaly-based detection solution, our approach relies on the assumption that an attack shows traffic characteristics that deviate from normal behavior. This assumption holds as long as an attacker does not want to increase its efforts to achieve stealthy communication. An attacker can increase the stealthiness of her communication by creating a *victim-aware adaptive (VAA) covert channel*. This means that the attacker does not only hide data within a message, but she also tries to craft a message such that it looks like a legitimate message generated by the victim. In other words, the attacker tries to attack the fundamental assumption underlying detection heuristics (i.e., malicious traffic has different characteristics from benign traffic). We call an attempt to exfiltrate data over such a channel: *victim-aware adaptive data exfiltration*.

The main goal of **RQ2** is to investigate how VAA exfiltration attempts can be detected. A new detection approach is only needed if the existing solutions are not effective. Therefore, the goal of this chapter is to evaluate whether existing detection approaches are effective against VAA data exfiltration. First, we introduce the design of a VAA covert channel, which mimics legitimate traffic at the application layer. The covert channel works in two phases: 1) the victim traffic is observed and its network characteristics are learned, and 2) a message with the victim's characteristics is created, and the secret data is hidden in some elements of the message. Essentially, it is a mimicry attack performed on live data and it is contextualized to each specific victim. Second, we introduce CHAMELEON, a framework that implements the VAA covert channel. CHAMELEON is a configurable toolchain that generates labeled datasets containing VAA covert channel attacks from any network trace of benign traffic. We use CHAMELEON to generate synthetic datasets that represent different strategies an advanced attacker may use to

¹This chapter is an edited version of a peer reviewed paper that will appear in SecureComm '19 [3]

perform VAA data exfiltration. We use these datasets to evaluate the effectiveness of three different covert channel detection solutions over HTTP: DECANter [1], DUMONT [42], and HED [53]. We chose these three approaches because they represent both *semi-supervised* (i.e., [42], [1]) and *supervised* (i.e., [53]) detection approaches. Our evaluation shows that these detection solutions are not effective in detecting VAA data exfiltration attacks. A solution to detect the attack presented in this chapter is discussed in Chapter 6.

5.1 Motivation

Attackers can make their communication stealthier by using mimicry techniques. Mimicry techniques were introduced in the host-based IDS domain by Wagner and Soto [126], but their underlying principle can also be applied to network traffic. Attackers can apply different techniques to morph network messages into benign-looking traffic, and therefore try to evade detection. We call this type of channels *adaptive covert channels*. Adaptation can be performed in two ways: *apriori adaptation* and *victim-aware adaptation*.

The most common type of adaptation is *a priori adaptation*, where the attacker implements her communication to look like benign software or protocol *before* compromising any machine. The anti-censorship community provides several examples [127], [128]. Wright et al. [129] proposed to modify the packet sizes distribution of one class of applications in order to resemble another class. Moghaddam et al. [130] and Weinberg et al. [131] proposed two tools to camouflage TOR traffic as Skype and HTTP protocols, respectively. Examples can also be found in known malware samples, which mimic predefined, well-known applications, such as the Windows and Yahoo Messenger protocol [16], common browsers, or other known application (e.g., see URSNIF in Figure 3.5).

However, there exist detection systems that detect covert channels by modeling the normal network behavior of a monitored host (i.e., potential future victims), as we have shown in Chapter 3 and 4. These systems likely detect *apriori adaptation* because the choice of adapting to a specific application (or protocol) does not necessarily match the behavior of a specific victim. Thus, the adaptive covert channel would show different network characteristics from the victim traffic. An example is shown in Figure 3.5b, where the malware mimicked the Microsoft Crypto API (i.e., *apriori adaptation*), but DECANter was still able to detect the anomalous message. An *a priori adaptation* technique can avoid detection if the detection model is known. In this setting, Fogla et al. [132], [133] proposed polymorphic blending attacks (PBA), a technique to adapt shellcode payloads to fit the statistical representation of normal traffic embedded in the detection model. However, the detection model is not always available. In this work we assume the model is not available to the attacker.

As discussed above, detection heuristics assume that malicious traffic deviates from normal traffic characteristics. On the one hand, misuse-based approaches (e.g., supervised learning) analyze the characteristics of benign and malicious traffic to create a model that can reliably distinguish between the two groups. This

approach works as long as future malicious communications have similar traffic characteristics as malicious samples analyzed to create the detection model. On the other hand, anomaly-based approaches (e.g., semi-supervised learning) only rely on benign data, and the generated model describes the common characteristics of benign traffic. This approach works as long as malicious traffic characteristics deviate from those representing benign traffic. However, *what happens if the covert channel takes benign traffic characteristics of its victim into account to generate its messages?*

Victim-aware adaptation occurs when malicious channels mimic the observed victim traffic to bypass detection. Casenove [134] proposed a technique called polymorphic blending technique (PBT) that is based on PBA [132]. PBT learns the statistical representation of normal TCP payloads from the victim traffic. The data to be transmitted is then encoded using byte frequency distribution, such that the payload statistical byte distribution is similar to the victim traffic. However, PBT does not preserve the correct syntax of the application layer protocol, including its metadata. By substituting bytes in the payload, the syntax of the messages may be disrupted. Similarly, also PBA does not try to resemble a message syntactically similar to the victim traffic, but focuses on the byte frequency distribution. Consequently, detection systems that rely on application layer information for detection (e.g., [40], [42], [1], [53]) could detect PBT due to corrupted messages (e.g., unparsable syntax) or unrecognizable byte sequences. Yarochkin et al. [135] propose to use covert channels over different protocols depending on the protocols used by the victim. Although their solution adapts to the victim, they do not consider to mimic the victim messages, thus their content would show deviating characteristics from the victim traffic and the covert channel would be detected.

In this chapter we propose a different mimicking approach, which also provides statistical similarity, but retains the syntactic structure of a message. Our mimicking approach preserves the content and metadata of the victim traffic, which cannot be achieved using PBT.

Although we are not aware of malware families using VAA covert channels, there exist malware families capable of sniffing and manipulating their victim traffic [20], [21]. This suggests that existing malware already has the technical capabilities to implement VAA covert channels, and that malware has interest in performing malicious activities using the victim traffic. For this reason, we consider VAA covert channels to be practical for advance attackers and we decided to investigate the effects of *VAA covert channels* on existing detection approaches. Specifically, our evaluation focuses on data exfiltration attacks over VAA covert channels. Due to the lack of existing malware samples performing VAA data exfiltration, we introduce a configurable framework that allows us to create synthetic labeled datasets containing both benign and victim-aware adaptive data exfiltration traffic: CHAMELEON. This framework creates the conditions to evaluate the detection performance of existing detection heuristics.

5.2 Intuition

The goal of VAA data exfiltration is to hide secret information within messages that resemble benign messages generated by the victim. In other words, VAA data exfiltration tries to *fit* the “benign class” of a detection model.

We assume that the attacker monitors the infected machine by collecting traffic information such as headers and their values. Then, the attacker can create a message that: 1) resembles the information observed from the infected machine; and 2) hides the secret data that the attacker wants to exfiltrate. A detection system would transform such a message into network features, and it would evaluate them with the model representing benign traffic. Since most of the message is the same as the content that appears in normal traffic, it is likely that the malicious message has network features similar to benign traffic features. Thus, the detection system would label the message as benign. This is an attack against the underlying assumption of anomaly-based detection solutions. However, VAA data exfiltration has also effects against misuse-based solutions. First, a supervised model trained with the traffic of a malicious sample that infected machine *X*, may not identify as malicious the traffic of *the same sample* that infected machine *Y*. The reason for this is that machine *Y* may have different traffic patterns than machine *X*, therefore the traffic patterns of the malware change as well. Thus, the model may not work as expected, despite being trained on exactly the same sample. Second, in case the characteristics of benign and malicious data are similar, it becomes difficult to identify patterns that effectively distinguish the two classes. As a consequence, the final model may become ineffective in identifying the two classes, thereby generating too many false positives.

5.3 Threat Model

In this chapter we assume a more powerful attacker than the one discussed in Section 3.3. We assume an attacker wants to establish a hidden communication channel to bypass a security monitor (i.e., NIDS). The attacker controls both client and server, and the monitor is capable of inspecting the communication channel (e.g., the monitor can read communications as plaintext). The attacker is capable of observing the network traffic of the compromised host, and may generate messages that are similar to those generated by the compromised host (i.e., adaptive). Therefore, we assume the attacker has enough privileges on the infected machine to read from its network interface. Note, that this does not necessarily imply that the attacker has system privileges (i.e., root) on the compromised machine. In fact, this can occur, for instance, when sniffing tools (e.g., `tcpdump`) are given the permission and capability to allow raw packet captures (e.g., `setcap cap_net_raw,cap_net_admin`). Thus, an attacker can access network packets by compromising a user having permissions to run such network tools.

Although it is not trivial for an attacker to compromise such a user, we believe it is realistic for an advanced attacker (e.g., APT). The first advanced attacks have emerged. For example, security researchers have discovered different malware samples with network sniffing capabilities. The Emotet botnet [20] uses sniffing

capabilities to steal information from the host, or VPNFilter [21], a more recent botnet, can sniff and also manipulate network traffic in order to perform malicious actions. These examples show that it is realistic to assume that a malware can (and wants to) read network traffic in non-targeted attacks such as botnets. Consequently, it is also realistic to believe that APT may use the same techniques targeted attacks.

Lastly, we assume that the attacker only adapts outgoing traffic. We analyze a unidirectional VAA covert channel, because it is enough to bypass existing detection systems.

5.4 CHAMELEON

CHAMELEON is a configurable toolchain that generates VAA covert channel traffic. We use CHAMELEON to create a groundtruth dataset to evaluate the effectiveness of existing detection solutions against VAA data exfiltration attacks. The toolchain will be publicly released. In the next sections we briefly discuss the process of generating VAA messages, and we describe the architectural and implementation details of CHAMELEON.

5.4.1 Crafting VAA Messages

Adaptive network messages are generated following a two phase process: the *collection* phase and the *blending* phase. This terminology was introduced by Casanove [134]. Figure 5.1 shows a high level abstraction of how adaptive messages are crafted. These phases are performed by the client controlled by the attacker.

The collection phase is responsible for gathering information about the victim traffic. For instance, what headers are used in a specific protocol, what values are associated to each header, and so on. This step can be performed by sniffing the traffic on the network interface (see step 1 in Figure 5.1). The collected information allows the attacker to correctly mimic the victim traffic.

The blending phase is responsible for crafting messages that look like victim traffic, while hiding secret data within those messages. The blending phase takes as input the collected information and the secret data that the attacker wants to transmit, it merges them in a VAA message, and transmits the VAA message to a server controlled by the attacker (see step 2 in Figure 5.1).

5.4.2 Architecture

CHAMELEON takes as input a network trace, which represents the potential victim. From this trace, it generates another network trace containing traffic representing a VAA covert channel. Then, CHAMELEON merges the two network traces together. The merged trace can then be used to evaluate detection systems, because it contains both the normal traffic, from the original trace, and VAA covert channel traffic. An overview of CHAMELEON is shown in Figure 5.2.

CHAMELEON works in two steps. The first step is *Adaptive Traffic Generation*, which involves a client and a server. The client reads the network trace provided as

Figure 5.1: Overview of the process of generating VAA messages. Benign applications send messages over the network. These messages are collected by the attacker during the collection phase. In the blending phase, the attacker crafts her messages by using the information collected from the legitimate traffic and by hiding within them the secret data to exfiltrate. These messages are then sent to her controlled server.

input, and it generates adapted network messages by performing both collection and blending phase. Then, the adapted messages are sent to the server, which responds to the client. The second step is *Traffic Integration*, which involves a set of tools used to modify network traces. These tools are used to integrate the adapted traffic, stored in a temporary trace, into the original trace. CHAMELEON is implemented to work for HTTP traffic. We have implemented CHAMELEON in Python.

We decided to design CHAMELEON as a “dataset generator”, so we could generate datasets including VAA covert channels traffic by using only network captures. This allows us to obtain datasets from machines running different operating systems and applications, and to easily test defensive mechanisms in different settings. This design decision makes it also easier for other researchers to generate VAA covert channel datasets without requiring a dedicated machine on which CHAMELEON needs to run. Instead, previously captured network traffic can be injected with adapted traffic to analyze.

5.4.3 Adaptive Traffic Generation

The first step of CHAMELEON is to generate a trace containing VAA traffic. The VAA traffic mimics the traffic observed in a network trace that is provided as input (i.e., the victim traffic). The design of a VAA covert channel follows the high level steps discussed in Section 5.4.1. As in any covert channel, before any communication, client and server need to share a set of parameters, in order to be able to

Figure 5.2: Overview of the process followed by CHAMELEON, composed by two separate phases: *Adaptive Traffic Generation* (see Section 5.4.3) and *Traffic Integration* (see Section 5.4.4)

identify the hidden data from the messages. We discuss these setup parameters in the following subsection. Moreover, we also discuss the set of message parameters, which a VAA client may use in order to specify its way of communicating, and the details of our implementation, including how we perform the collection and blending phase.

As we mentioned in Section 5.3, our work focuses on unidirectional adaptive covert channels, where only the client adapts to the victim. Thus, in CHAMELEON, the server is a listener that provides default responses.

Setup Parameters

Setup parameters SP are shared between the client and the server before any communication is established. These parameters provide the basic information for the client and server to identify the hidden data within the network messages, and they are defined as $SP = \langle p, e \rangle$. For each network protocol p , the VAA client and server share a list of encoding algorithms e . The parameter e also contains a delimiter to identify the starting point of the hidden data. Our implementations uses HTTP and base64 as protocol and encoding, respectively.

Message Parameters

Message parameters MP are *not* shared between the client and the server. They are used to specify how messages should be crafted in the collection and blending phases such that they appear to be legit messages, while hiding covert information. As CHAMELEON only focuses on outgoing covert channels, the MP only have to be set at the client side. The MP define how the clients generate messages for their adapted communication. $MP = \langle c, s, l, i, b, a \rangle$, where c is the timeout of the collection phase; s is the maximum size of the data to embed in each message; l is the location where the exfiltrated data is embedded (e.g., URI, headers, body); i is the delay between sending out successive messages; b is the timeout for the

blending phase; and a is the type of application the attacker wants to mimic (e.g., Firefox). The values for i , l , and s may be learned in the collection phase. For example, CHAMELEON can learn from the collected traffic what is the average delay between messages, what is the location in network messages that often changes (e.g., measure how many unique values are associated to each header), what is the average size of network messages, such that it can adjust the amount of information to hide within its template.

Client's Implementation Details

The collection and blending phases depend on each other, meaning that the information gathered during the collection phase influences how the blending phase crafts the messages. During the collection phase, for a specific application a communicating over HTTP, we propose to store the order of header fields for each message and the values corresponding to each header field, which we refer to as the *header set* and the *header-value dictionary*, respectively (see Figure 5.3). A specific application can be identified using a string match in the **User-Agent** field. Once collection phase ends, the collected information and the secret data to exfiltrate are given as input to the blending phase.

The blending phase encodes the data using scheme e (from SP) and split it into chunks of size s (from MP), obtaining a set of *data items*. Before transmitting the data, we generate a template. A template represents a set of headers, and their associated values, that aim at mimicking the victim traffic. Headers are randomly chosen from the header set, while the associated values are randomly chosen from the header-value dictionary. The data item to transmit (i.e., the covert message) is inserted in the location, within the template, indicated by the parameter l , thereby generating the adapted message (see Figure 5.3). Thus, CHAMELEON *inserts data in a single message location, in order to maximize the mimicked parts of the message*, since the rest of the message is represented by the template which contains only victim's traffic information. Finally, the adaptive message is sent out to the server using interval i . After b seconds, the process restarts from the collection phase to mimic the most recent traffic. The server retrieves the message by identifying the delimiter described in the setup parameters e and by extracting the adjacent data.

Our implementation uses HTTP and base64 as setup parameters. Message parameters MP can be set before running the client and the server. Please note that whoever establishes a covert channel is also in control of both client and server behavior. Therefore, *an attacker may use the protocol in a way that is semantically wrong, but she can still communicate successfully*. For example, the client can generate HTTP GET requests for resources that do not exist on the server, but the server may still provide a valid response (e.g., 200 OK).

5.4.4 Traffic Integration

The second step CHAMELEON performs is the integration of the VAA traffic, observed between client and server, into the original trace provided as input. First,

Figure 5.3: Overview of VAA covert channel implementation by CHAMELEON. The left part of the Figure shows the collection phase, where CHAMELEON extracts HTTP messages from a victim’s trace (i.e., .pcap), it parses the messages, and stores the sequences of headers it has observed, and the list of values associated with each header. The right part of the Figure shows how CHAMELEON generates templates from the collected information (i.e., Header set and Header-value dict), and how it combines them with the secret data (i.e., data) in order to generate adaptive messages. Each dotted arrow indicates the parameter associated to each element.

the traffic of the VAA channel is captured using `tcpdump`, and it is stored into a temporary network trace. Then, the IP addresses in the temporary trace are rewritten according to the victim’s IP contained in the original trace using `tcprewrite`. Similarly, the timestamps are rewritten in order to fit the time period of the original trace, and we achieve this using `editcap`. Finally, now that both traces are consistent in terms of IP address and timestamps, we merge them using `mergecap`. These tools are part of the Wireshark suite. The final trace contains both victim’s traffic and the attack, and it is easily labeled, thereby being ideal for an evaluation of detection systems.

5.5 Evaluation

We evaluate the detection performance of three state-of-the-art solutions against a dataset generated by CHAMELEON. In this section we describe how we generated the dataset and its characteristics, give an overview of the three detection mechanisms, describe the evaluation setup, and show the results. Lastly, we summarize the lesson learned from this study and the limitations of CHAMELEON.

5.5.1 Dataset

For the generation of the dataset, we need network traces of benign traffic representing different hosts, and thus, potential victims. We choose the dataset published by Sharafaldin et al. [136] for this purpose. The reasons behind this choice are the following: 1) it contains traffic from multiple hosts over a time period of a week, allowing defenses to be trained with the traffic of the beginning of the week, and to be tested on the rest; 2) it contains traffic emulating user behavior and it contains different type of machines (e.g., servers and workstations); and 3) it is publicly available, so our generated dataset can be publicly released, without

compromising any user privacy, making our work easier to be reproduced.

We extracted from the dataset of Sharafaldin et al. [136] all outbound traffic generated by 9 different hosts, which represent, in our work, potential victims of VAA data exfiltration. For each host, we obtained 5 days of network traffic (from Monday to Friday), where each day of traffic is a single network trace. For each host, we choose a different set of message parameters MP , which represents a *strategy* a VAA covert channel may use to exfiltrate data. We ran CHAMELEON using each host’s network trace and defining the MP according to the strategy assigned to the host. Overall, the dataset resulted in 45 pcap traces containing both our VAA data exfiltration and hosts traffic. The dataset contains 16.6 GB of network traffic, of which 476 MB (120k HTTP requests) originate from the adapted communications.

Table 5.1 lists the message parameters of each strategy applied to each host. The column *file* indicates the size of the file we transmitted during the adaptive communication. The values of l are denoted by B for body, H for a random header, U for URI parameters and M for a random choice between header and URI. While the values of a are given as B for browser, N for non-browser, or M for a mix of both. The parameter s represents the *exfiltration rate* per message, in other words, the size of secret data that can be hidden in an adapted message. The strategies represents different exfiltration scenarios. Such scenarios include data exfiltration of files large 50 Kb and 500 Kb at different speeds, exfiltration locations, and application types. For instance, strategies for hosts 8 and 16 wait up to 5 seconds between sending messages, whereas strategies for hosts 5, 9, and 12 do not wait between attempts at all.

Our dataset is relevant for our experiment because it helps to test the detection performance against different scenarios of VAA data exfiltration. This choice is important because the traffic of data exfiltration is strongly influenced by the different parameters an attacker may choose. Nevertheless, we are aware that our dataset does not comprehend all possible exfiltration scenarios. However, it is still unclear whether a finite set of exfiltration scenarios can be defined.

5.5.2 Existing Detection Solutions

The three detection tools are: DUMONT [42], DECANter [1], and HED [53]. We selected these tools from the state-of-the-art for three reasons: 1) their detection models rely on network traffic characteristics that include payload and headers, and 2) their detection models leverage benign traffic information; and 3) they cover both semi-supervised (DUMONT and DECANter) and supervised learning (HED). We obtained the code of HED from the authors, while the implementations of the two other systems were available from our previous research, as discussed in Chapter 3, where descriptions of both DECANter and DUMONT are also provided. On the other hand, HED [53] is a supervised detection solution, which generates a binary SVM classifier from a training dataset containing “normal” background traffic and covert channel traffic. HED considers a set of more than a thousand features, such as entropy, length and n-grams for all string features. Feature selection is applied to identify the most relevant subset of features, which

Host	file [KB]	Message Parameters					
		c [s]	s [B]	l enum	i [s]	b [s]	a enum
5	500	5	128	H	0.00	∞	M
8	50	5	0-128	U	0-5.00	∞	B
9	50	5	128	B	0.00	5	M
12	500	5	128	H	0.00	∞	N
16	50	5	0-128	U	0-5.00	∞	N
17	500	5	0-128	B	0-1.00	∞	B
25	500	5	0-128	U	0-0.05	5	B
50	500	5	0-128	H	0-0.05	5	B
51	500	5	0-128	B	0-0.05	5	B

Table 5.1: Mimicking strategies applied in our dataset. Each row represents a strategy, and it uses the network traces of a specific host. The strategy is described as a combination of parameters (i.e., the columns).

is then used to train the model. In the testing phase, HED classifies a HTTP request as malicious or not according to the SVM trained decision function.

5.5.3 Evaluation Setup

We evaluate the classification performance of each detection tool in terms of *accuracy* (ACC), *false positive rate* (FPR), *true positive rate* (TPR), *detected attack* (DA) and *detected strategy* (DS). An attack is detected if at least one HTTP request is triggered as malicious within a single traffic sample. A strategy is detected if at least one HTTP request is flagged as malicious within *all* the four traffic samples of that specific strategy. We compute the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN), for the whole dataset. The accuracy is computed as $ACC = \frac{TP+TN}{TP+FP+FN+TN}$, the false positive rate as $FPR = \frac{FP}{FP+TN}$, and the true positive rate as $TPR = \frac{TP}{TP+FN}$.

DUMONT

As we discussed in Section 5.5.1, the traffic of each host is represented as five network traces labeled as days of the week (i.e., from Monday to Friday). For the evaluation of DUMONT, we first train a model for each host using the first half of the benign training data of the Monday sample. The second half, and the same amount of malicious data present in the Monday sample, is subsequently used for calibration². We test the remaining four samples, from the same host, against the model and we collect the results. The learning technique used by DUMONT is a One-Class SVM (OCSVM), which takes two parameters to define the non-linear

²Please note that in the original work [42] there is no clear guidance about the quantities of data needed for the calibration step.

model: the width of the Gaussian kernel γ , a value fp (i.e., soft margin) which represent an upper bound on the fraction of training errors, and a parameter α that helps DUMONT to further calibrate the soft margin, in order to be more or less restrictive against false positives. We specify the width of the Gaussian kernel as $\gamma = |\text{features}|^{-1}$, the default value of Scikit-learn’s One-class SVM [124]. In addition, our upper-bound on the false-positive rate was chosen as $fp = 0.001$ allowing for a maximum of 0.1% false-positive rate without calibration. To select the α value, we ran the experiments with $\alpha = 0.001, 0.01$, and 0.1 , through 0.9 . In our results we only show the outcome of $\alpha = 0.01$ as it gave the best accuracy in our tests.

DECANTeR

We train a model for each host using the Monday traffic sample and ignoring malicious requests, since DECANTeR requires only benign traffic for training. We test the remaining four samples from the same host against the model and collect the results. Our evaluation considers the classification performance in terms of HTTP requests, and not fingerprints. Thus, we consider all requests to be malicious if the fingerprint is classified as malicious, and benign otherwise. As discussed in Section 3.4.3, we also assume that an operator updates the fingerprints in case of false positives.

HED

We train a single model for all hosts using their Monday samples, and evaluate it against their remaining four samples, as described in the original work [53]. However, if we generate a supervised model from a set of known VAA malicious samples, we effectively train the model to detect the malicious samples in the context of their specific victims. Hence, *it is not guaranteed that the same samples are effectively detected in the future*, because their traffic may look different if they affect other victims. In order to give a full overview of the performance of HED, we evaluate it in two scenarios representing two different assumptions in its deployment.

We define the first scenario, HED_{pk} , assuming HED has *partial knowledge* of the malicious host traffic. Thus, the training model contains all benign host traffic, but only a subset of malicious traffic. In this way, we can verify if the trained model is able to effectively detect “unknown” malicious traffic (i.e., not used to train the model). In our evaluation we decided to randomly exclude the malicious traffic of half of the hosts to test this scenario. We excluded host 50, 12, 8 and 25. The second scenario, HED_{tk} , assumes HED has *total knowledge* of the malicious hosts’ traffic. Thus, the training model contains all benign and malicious host traffic. HED_{pk} is a more realistic scenario, because it assumes the defender may not know all the characteristics of a VAA attacker upfront.

5.5.4 Results

As we discussed at the beginning of this chapter, our goal is to evaluate the detection performance of existing detection solutions against adaptive data exfiltration. More specifically, we want to know whether the existing solutions are effective or not.

We consider a solution to be effective if it has a true positive rate higher than 70% and a false positive rate lower than 2%. A high TPR means that it is unlikely that the attack is going to be missed. Thus, in practice a low TPR can be very costly for a company. A low FPR means the detection solution rarely triggers false alarms. According to Sommer and Paxson, in their seminal work about IDSs and machine learning in real environments [76], limiting false positives must be a top priority of any IDSs. Table 5.2 shows the overall detection performance of the evaluated detection solutions for all the 9 strategies. Table 5.3 shows a detailed overview about detection performance per strategy.

Semi-supervised learning

DECANTeR and DUMONT are not effective in detecting VAA covert channels, as shown in Table 5.2. The main reason is that adaptation allows the malware to camouflage its characteristics with those of the victim. Thus, the malware fits the trained model and breaks the fundamental assumption of anomaly detection, namely that malware shows different patterns than benign traffic. The results of the evaluation are shown in Table 5.2.

DECANTeR is not effective against CHAMELEON because its classification system is based on exact matches of *some* header values or header names. Since CHAMELEON uses existing header values and names in its template, DECANTeR considers adapted messages as generated by existing applications. CHAMELEON can exfiltrate data through the **URI**, **Body**, **Accept**, and other headers, without being detected, as long as the “key” header values (i.e., **Host**, **User-Agent**, **Accept-Language**) match the profile values. If data is hidden within such “key” headers, then DECANTeR would likely detect the adapted traffic.

The most successful strategies involve mimicking background applications. The reason is that adapted messages have the same header sequences, **Host** values and **User-Agents**, these are three fundamental features for background application fingerprinting in DECANTeR. By adapting to the victim traffic, CHAMELEON completely avoided detection in two cases (strategies 12 and 16), as shown in Table 5.3. DECANTeR considered all adapted message to be benign, because the set of headers analyzed by DECANTeR matched exactly with benign traffic, while the exfiltration occurred in other message locations. DECANTeR performs poorly for strategies 5 and 9, when the malware randomly adapts to different applications. DECANTeR does not suffer as much when the adaptation involves a browser application. This can be explained by the Referer graph. Malicious requests do not always create a valid Referer graph despite being adapted, because they may adapt to requests without Referer field (e.g., OCSP requests). Therefore, these messages are considered to be generated by a background application, and a match with the correct application is not found. However, a VAA attacker could decide to always

include the **Referer** header with a value from previous requests.

DUMONT is not effective against **CHAMELEON** because its learning model (i.e., **OCSVM**) relies on statistical features describing the length and structure of different parts of a message (i.e., headers, **URI** and **Body**), their entropy, and the average length of header and **URI** values. The template created by **CHAMELEON** contains mostly the same data used by benign applications, excluding the element where secret data is hidden. Therefore, the data contained in the template has similar statistical features to the traffic used to train the **OCSVM** model. This similarity to benign characteristics helps the adapted traffic to hide its presence from detection. Furthermore, if **CHAMELEON** also hides data in locations that typically contain large amounts of data, such as **Body** and **URI**, then **CHAMELEON** is likely to fit the trained length characteristics learned by the model and, thus, to avoid detection. The use of base64 as an encoding technique allows to have entropy values not far from the values of **URI** or **Body** content, thereby reducing the impact of such features. Finally, temporal features used by **DUMONT** did not show enough impact in the detection. Hence, since adapted traffic share most of the characteristics with benign traffic, the feature vectors of adapted traffic fall within the boundary delimited by the decision functions, thus malicious requests are often misclassified as benign.

Overall, **DUMONT** misses almost a third of the samples. It misses all the samples for strategies 16 and 9, probably due to the small amount of data being exfiltrated. Although **DUMONT** detects some malicious requests for strategies 5, 8, 25 and 50, the vast majority of malicious requests are still not detected. This justifies the low TPR.

Supervised learning

HED yields the best results, as shown in Table 5.2. All the samples are detected, and the accuracy is higher compared to semi-supervised defenses for both scenarios HED_{pk} and HED_{tk} . Supervised learning leverages the extra knowledge about the attack to improve its performance.

HED leverages over a thousand statistical features of *both requests and responses* in its classification system (e.g., unigram, entropy, length). The static *responses* of **CHAMELEON** provide a detection advantage for **HED**, because the detection model can be trained using malicious static characteristics. Nonetheless,

Table 5.2: Overall performance of existing defenses in terms of accuracy (ACC), false positive rate (FPR) and detected attacks (DA).

Existing Defense	Performance			
	Acc	TPR	FPR	DA
DUMONT ($fp=0.001$, $\alpha = 0.01$)	49%	10%	2%	17/35
DECANTeR	70%	50%	3%	12/35
HED_{pk}	89%	85%	7%	35/35
HED_{tk}	96%	93%	7%	35/35

the FPR of both scenarios is 2 or 3 times higher than for the other solutions, making HED not effective against CHAMELEON. In the original paper [53], HED showed a FPR of 0.01% against non-adaptive covert channel. In our evaluation *the FPR increases two orders of magnitude*. This highlights the difficulties of detecting VAA covert channels. HED has a FPR of 7%, which is too high to be considered *practical*. Usually, 1% is already considered too much, since millions of events should be classified each day.

The reason behind the high FPR is the similar statistical representations of benign and adapted messages. Similarly as for DUMONT, templates contain data statistically similar to the one used in training, because it is data from the same application. Consequently, when the model is generated, it is difficult to find a decision function that can reliably separate the two classes of traffic. Moreover, due to high number of requests observed in HTTP traffic and the inherent heterogeneity of HTTP, HED often misclassifies benign HTTP traffic as malicious.

Table 5.2 shows that HED achieves 96% accuracy in its ideal settings HED_{tk} , where all strategies are known during training phase. However, the accuracy is 89% in the more realistic setting HED_{pk} , where only some strategies are known at training time. This 7% accuracy drop is due to the lack of knowledge about new adaptive strategies. As shown in Table 5.4 the true positive rate is lower for strategies 8, 25, and 50, which were excluded from the training dataset. Additionally, the templates of new adaptive messages are similar to the messages of the victims, which during the training phase were labeled as benign. Therefore, this makes it harder to correctly classify the data. The accuracy drop shows that HED does not have a stable detection performance, and that HED is sensitive to the data used in training. Furthermore, these drops occurred despite the static malicious responses across all strategies. This limitation of our implementation helps HED to achieve 89% accuracy, but it does not affect DUMONT and DECANter because they do not analyze server responses. It is possible that by creating more realistic server responses for the adaptive covert channel, the accuracy of HED could decrease even further.

Lessons Learned

Our evaluation shows that existing defensive mechanisms are not effective against detecting VAA covert channels, because they cannot effectively detect them while, at the same time, triggering few false alerts. Semi-supervised learning solutions are not effective against VAA attackers, because malicious traffic fits the model of benign communication, and thus, malicious connections are not flagged as anomalies. Supervised learning solutions are not effective for two reasons. First, the classes in the training dataset are represented by similar sets of data, thus there is not a clear distinction between the two classes features, which is a fundamental requirement for supervised approaches to be effective. Second, the VAA covert channel data used during the training of the system may not be representative for successive attacks, since the attacker mimics the traffic of new victims, which may show different network patterns. Both issues are present in HED: the first is highlighted by the high FPR, while the second is described by the accuracy drop between the

Strategies	DUMONT			DECANTeR		
	TPR	FPR	DS	TPR	FPR	DS
5	0.001	0.01	✓	0.50	0.02	✓
8	0.005	0.01	✓	0.18	0.05	✓
9	0.000	0.01	×	0.09	0.05	✓
12	0.133	0.03	✓	0.00	0.00	×
16	0.000	0.01	×	0.00	0.03	×
17	0.371	0.03	✓	0.51	0.03	✓
25	0.023	0.01	✓	0.61	0.02	✓
50	0.001	0.03	✓	0.78	0.31	✓
51	0.150	0.06	✓	1.00	0.03	✓

Table 5.3: Overall performance of DUMONT and DECANTeR per strategy, in terms of true positive rate (TPR), false positive rate (FPR) and detected strategy (DS) (× represents a missed strategy).

Strategies	HED_{pk}			HED_{tk}		
	TPR	FPR	DS	TPR	FPR	DS
5	0.99	0.06	✓	0.50	0.02	✓
8	0.50	0.05	✓	0.99	0.07	✓
9	0.98	0.07	✓	0.99	0.07	✓
12	0.99	0.07	✓	0.99	0.05	✓
16	0.98	0.07	✓	0.99	0.07	✓
17	0.99	0.10	✓	0.98	0.09	✓
25	0.55	0.12	✓	1.00	0.10	✓
50	0.68	0.04	✓	0.93	0.14	✓
51	0.99	0.10	✓	0.99	0.05	✓

Table 5.4: Overall performance of HED_{pk} and HED_{tk} per strategy, in terms of true positive rate (TPR), false positive rate (FPR) and detected strategy (DS) (× represents a missed strategy).

two HED scenarios HED_{tk} and HED_{pk} .

5.5.5 Performance Analysis of CHAMELEON

In this section we analyze two different aspects of CHAMELEON’s performance. First, we investigate the memory costs of collecting victim information for a VAA covert channel. Second, we evaluate the computational complexity of CHAMELEON operations in terms of big O notation.

Memory costs

During the collection phase, the client stores in memory the network information observed from the victim. We evaluated the memory footprint of CHAMELEON based on the amount of memory it needs to store for each collected message. Our analysis evaluates the memory costs for the first thousand messages. Please note that in our previous evaluations, as shown in Table 5.1, CHAMELEON performed collection phases lasting 5 seconds, where it collected a much lower number of messages.

We computed the memory costs for all the samples. The results show that, in the worst case scenario of our dataset, CHAMELEON would have stored 250Kb of information in memory after collecting a thousand messages. On average, the memory costs for a thousand messages is slightly above 100Kb. This fluctuation in memory costs is caused by the different usage of the protocol by the installed applications. An application using large unique values, such as long URLs, forces the client to store large strings, thereby increasing the costs. On the other hand, the client stores less information when applications repeat the content in their messages. The results are shown in Figure 5.4. Unsurprisingly, the memory size linearly increases with the increase of messages. The memory costs is also highly dependent on the traffic of each host. In case messages are repeated, the memory size does not increase, as shown by the orange series around the 700th message.

We consider the memory footprint of CHAMELEON to be acceptable, considering the large amount of memory available in modern systems. Especially, if we consider that the collection phase can run, periodically, every few seconds only, as we did in our experiments. In such setting, the memory footprint is almost negligible.

Computational Complexity

We analyze the two phases performed CHAMELEON separately. The collection phase has complexity $O(n)$, where n is the number of messages being collected. The collection is linear because each message is collected and parsed only once.

The complexity of the blending phase depends on the amount of data that should be exfiltrated to the server. Let’s assume the attacker wants to exfiltrate a file f , which is split in k data chunks. k represents the amount of adapted messages that must be transmitted. For each message, CHAMELEON chooses one list of headers h from the header set data structure, and, for each header, it chooses a random value stored in the header-value dictionary. Thus, the computational

Figure 5.4: CHAMELEON memory costs in a pessimistic scenario where the covert channel client collects a thousand messages. The orange line represents the worst case we observed in our dataset, while the blue line represents the average of memory costs across all hosts.

complexity of this step is $O(|h|)$, where $|h|$ is the amount of headers that should be associated with a value. The operations of choosing a header set or a value has constant complexity $O(1)$, since we only extract a random value from a set. Thus, the computational complexity to exfiltrate a file f is $O(k \cdot |h|)$. Considering that in a message the number of headers is often between 5 and 10 headers, the complexity is more dependent on the number of data chunks created to split the file. The amount of data chunks represents the stealthiness of the covert channel, meaning that more data chunks means slower and stealthier exfiltration. However, the higher is the number of data chunks, the higher is the computational complexity. An increase of computational complexity and a longer period of activity may potentially expose the presence of the attacker.

5.6 Limitations

CHAMELEON has three main limitations. First, it only mimics the content of network packets and not the interaction between client and server. A detection mechanism that models client-server interactions may effectively detect CHAMELEON. However, web traffic is heterogeneous and inconsistent client-server interactions may be frequent, considering the large volumes of web traffic. Thus, designing an effective heuristic is non trivial.

Second, CHAMELEON does not implement the concept of connection state. Thus, CHAMELEON can be easily detected if it is used to mimic stateful protocols such as FTP. To avoid detection in those protocols, CHAMELEON should be

extended to mimic also connection states. However, this limitation does not affect CHAMELEON over HTTP, because it is challenging to reliably monitor, for detection purposes, HTTP connection states (i.e., Cookies) due to the high heterogeneity of their usage and values across different web services.

Third, CHAMELEON can be detected using signatures or dedicated heuristics to identify specific patterns in the tool implementation. A simple example is to create a signature to the static server response. As discussed by Houmansadr et al. [127], this is always possible for defenders to detect mimicking techniques. However, these detection approaches are easy to evade since they rely on implementation details rather than on the underlying patterns of the exfiltration technique. Therefore, they cannot be considered a final solution to the problem of detecting VAA covert channels.

Finally, it may seem trivial to detect CHAMELEON using heuristics that identify replicated content in network messages. However, this is not the case due to some challenges that defenders must take into account: 1) benign applications regularly generate similar messages over time (e.g., scripts uploading data or downloading dynamic content), 2) defenders have to monitor large quantities of data and keep a detailed historical record for each host; and 3) CHAMELEON is a tool that can be configured with different parameters (e.g., increase delay to enforce larger historical analysis for defenders). These challenges make it non-trivial to effectively detect VAA covert channels using heuristics that rely on passive network traffic analysis.

5.7 Conclusions

In this chapter, we have proposed the design of a VAA covert channel for data exfiltration, in which an attacker mimics the legitimate traffic generated by the victim to hide information and to avoid detection from a security monitor. We have implemented a configurable toolchain called CHAMELEON, which can generate network traces containing VAA covert channels that can be used as groundtruth datasets. We used the output of CHAMELEON to evaluate three existing detection systems against VAA data exfiltration attacks. Our evaluation showed that these systems are not suitable to effectively detect these advanced attacks.

This chapter has shown that a detection solution for VAA data exfiltration is needed, thereby partially addressing **RQ2**. In the next chapter we continue to address **RQ2** by investigating a detection system for a specific case of VAA data exfiltration.

Chapter 6

Detecting Victim-Aware Adaptive Data Exfiltration via HONEYTRAFFIC¹

In this chapter we continue to address **RQ2** by investigating the problem of detecting victim-aware adaptive (VAA) data exfiltration. In the previous chapter, we have shown that existing detection techniques cannot effectively detect VAA data exfiltration attempts. The main cause of the ineffectiveness is that existing detection systems assume that malicious software creates network messages with general message characteristics that differ from benign traffic. This assumption can be broken by VAA attackers, as shown by CHAMELEON. Unless we assume some limitations about the attacker mimicking capabilities (e.g., the attacker introduces distinguishing and observable patterns), we believe *it is very difficult to design passive heuristics to effectively detect such threat*. Thus, we decided to avoid passive heuristics and investigate a different defensive mechanism: deception-based detection. The overall idea of these techniques is to lure the attacker to make some mistakes that expose her presence. Examples of deception-based techniques are honeypots and canary tokens. A honeypot is a purposefully vulnerable machine that waits to be compromised by greedy attackers, thereby exposing their presence. Similarly, in the setting of canary tokens, an attacker uncovers her presence by accessing decoy files. Following the same line of thoughts, we introduce a deception-based detection mechanism for VAA covert channels: HONEYTRAFFIC. The concept behind HONEYTRAFFIC is to generate fake messages from each monitored machine such that a VAA attacker, while mimicking network traffic, includes the content of fake messages thereby exposing herself to detection. Following [137], there are no comparable network-based deception techniques. We evaluate our approach under certain assumptions, and we show it is a practical technique against specific strategies of VAA data exfiltration.

¹This chapter is an edited version of a peer reviewed paper that will appear in SecureComm '19 [3]

HONEYTRAFFIC focuses on effectively detecting the type of VAA covert channels represented by CHAMELEON, where the attacker generates messages using “message templates” containing only victim’s data, and she embeds secret data to exfiltrate in one of the message headers. HONEYTRAFFIC does not detect all possible types of VAA covert channels. More importantly, HONEYTRAFFIC *is not intended to detect covert channels that do not adapt to the victim*, because this scenario is already covered by existing approaches (e.g., [42], [1], [53]). We discuss these limitations at the end of the chapter. We consider HONEYTRAFFIC to be a *complementary* solution to existing detection approaches. The combination of HONEYTRAFFIC with existing detection solutions makes it harder for VAA attackers to evade detection, but not impossible. Despite the improvements introduced by HONEYTRAFFIC to detect some VAA covert channels strategies, VAA covert channels remain a threat hard to detect.

6.1 Motivation

The main motivation for the need of a technique to detect VAA covert channels is the inadequacy of existing detection mechanisms against this type of threat. In the previous chapter 5, we have shown that both *semi-supervised* and *supervised* detection techniques fail to effectively distinguish between the traffic of an VAA attacker and the infected victim traffic. Summarizing Section 5.5.4, semi-supervised learning solutions are not effective against VAA attackers, because malicious traffic fits the model of benign communication, and thus, malicious connections are not flagged as anomalies. Supervised learning solutions are not effective for two reasons. First, the classes in the training dataset are represented by similar sets of data, thus there is not a clear distinction between the two classes features, which is a fundamental requirement for supervised approaches to be effective. Second, the VAA covert channel data used during the training of the system may not be representative for successive attacks, since the attacker mimics the traffic of new victims, which may show different network patterns. Due to these limitations of the aforementioned methodologies, we decided to investigate a new detection mechanism based on deception.

6.2 Intuition

The main idea is to leave the challenging task of detecting VAA covert channels to the attacker. Our approach consists of a client-side component that generates VAA traffic containing *honey tokens*. When an attacker mimics the generated benign traffic to hide malicious communication, she will end up mimicking the HONEYTRAFFIC. This produces inconsistencies in the network traffic that can be detected by a security monitor. The attacker has to distinguish between the honey messages (i.e., adapted messages including honey tokens) and benign messages. As discussed in the previous chapter, defensive techniques are not able to effectively use heuristics to distinguish between adapted and non-adapted messages. Consequently, the attacker should face the same difficulties.

6.3 Threat Model

We assume there exists a *honey client* installed on each machine (i.e., potential victim). The honey client generates network messages mimicking the machine's traffic, and sends them towards a server, which we call *honey server*. Moreover, we assume there exists a security monitor (e.g., next-generation firewall or NIDS). The monitor is usually already part of the company's infrastructure. The honey server can establish secure communications with both the honey client and the security monitor to provide the setup information. Please note that the honey server could be part of the security monitor, and it is not necessarily an independent entity. However, in the rest of the text, we treat it as an independent component. As discussed in Section 5.3, VAA attackers assume a strong defender capable of accessing all communication in plaintext. Thus, we assume HONEYTRAFFIC to be capable of accessing the communication in plaintext (e.g, using a TLS proxy).

We assume the attacker compromises a machine where a honey client is installed. We assume the attacker is aware of the presence of honey messages, but she does not know the setup information (e.g., list of honey tokens).

In order to avoid detection, the attacker may attempt to identify honey messages, such that she can only mimic legitimate application. However, the attacker must analyze the whole traffic to succeed. Since we assume the attacker does not know the setup information, she must analyze the content of each message and identify the honey messages. Essentially, the attacker tries to identify a covert channel, which is a complicated task even for defenders as we discussed in the previous chapters. Analyzing the content of host's traffic is computationally expensive, thus it could potentially expose the presence of the attacker. Since it is in the interest of the attacker to be stealthy, we assume the attacker to be not capable of running advanced detection heuristics such as running analysis on the messages' payload.

Finally, we assume the attacker mimics messages uniformly at random, as the victim-aware covert channels implemented by CHAMELEON. Messages are randomly picked so defenders cannot predict what messages the attacker may be attracted to.

6.4 HONEYTRAFFIC

The architecture of HONEYTRAFFIC is composed of three main components: a client, installed on each monitored host, a server and a security monitor. We refer to the client and server as *honey client* and *honey server*, respectively. The system works in two alternating phases: a setup phase and a detection phase. The *setup* phase is responsible for delivering the information necessary to run the detection mechanism to each component. During the *detection phase*, the system monitors the network to identify VAA covert channels. We discuss a detailed example in subsection 6.4.1. An overview of the architecture is shown in Figure 6.1.

In the rest of this section, we discuss in detail the setup and detection phase. Moreover, we discuss an example describing HONEYTRAFFIC. Finally, we discuss

(a) Overview of the setup phase. The honey server provides all the setup parameters to the NIDS and the honey client.

(b) Overview of the detection phase, where a VAA attacker is detected because its messages contain honey tokens but are not sent to random destination IPs known to the honey server.

Figure 6.1: Architecture of HONEYTRAFFIC. Green components are needed to run HONEYTRAFFIC. The red component represents the VAA attacker.

how honey tokens can be generated and how difficult it is, for an attacker, to distinguish the honey tokens from normal traffic.

Setup Phase

The *honey server* is the main component during the setup phase. It is responsible for the generation of all the information needed to run the detection system. The honey server generates a set of *honey tokens*, a set of random IP addresses, a set of *honey signatures* (e.g., regex) to identify the tokens in the network traffic, and the setup and message parameters (i.e., *SP* and *MP* discussed Section 5.4.3) to hide the tokens in the network message. The random IP generation process excludes existing network nodes that may trigger network errors from existing machines using the chosen IPs, or by network devices that know these IPs do not exist. This precaution is needed to avoid attackers being able to easily identify honey traffic. Then, the server securely communicates the tokens, IP addresses, and setup and message parameters to the honey client, while it sends the signatures to the security monitor. An overview of the setup phase is depicted in Figure 6.1a.

The setup phase is repeated after a specified time period (a day) in order to change tokens and signatures. In other words, the system updates regularly its

key detection material. These changes make it difficult for an attacker to reliably identify honey tokens.

Honey Client Communications

Upon receiving the information from the server, the *honey client* starts collecting information about the machine's traffic (i.e., collection phase). Once enough information has been collected, the client creates a template for its network messages that resemble the machine's traffic, and it embeds the honey tokens within the template (i.e., blending phase). In other words, the honey client follows the same process of the VAA covert channel client discussed in Section 5.4.3. In this process it uses the *SP* and *MP* parameters provided by the honey server. Lastly, the client spoofs the destination IP of its messages using one of the addresses provided by the server, and it sends the messages.

Detection Phase

Once the security monitor receives the signatures from the honey server, it starts monitoring the network traffic. Whenever the security monitor identifies a honey token (*signature hit*), it redirects the message to the honey server for further inspection.

Assuming the tokens are unique strings, which is a common assumption in signature-based detection, a signature hit can occur for two reasons: either the token was included in the message by the honey client or from a VAA malicious application.

The server can identify if the message is generated from the honey client, by verifying whether the destination IP address matches with one of those provided during the previous setup phase. If the address does not match, then the message was generated by a VAA malicious application. In fact, since the attacker's goals is to communicate with a remote server that she controls, the attacker's messages very likely do not present a destination IP address that matches one of those provided during the setup phase, which is a small set of randomly generated IP addresses.

In case the message is identified as originating from the honey client, the server can answer with a standard response, or it can even communicate additional data back to the client. It is important that responses are not static and look like legit traffic, to avoid that the attacker easily identifies honey communications. The response is needed because the IP is spoofed, and a real destination is not reached.

In case the message is considered malicious, the server triggers an alert. An operator could then decide to drop the connection or keep investigating the incident. Figure 6.1b represents an overview of the detection phase.

6.4.1 Example of HONEYTRAFFIC

Assume that during the *setup phase* the honey server generates three different honey tokens:

(real) dst IP: 120.13.21.2 GET /pypi/v/dpkt.svg HTTP 1.1 Host: img.shields.io User-Agent: Mozilla/5.0 ... Firefox 54.0 Accept: */* Accept-Language: en-US;q=0.5,en;q=0.3 Accept-Encoding: gzip, deflate Cookie: __cfduid=dce101b3e09a9a09d5dbf2afed17dcdc71413792427910 Connection: keep-alive	(fake) dst IP: 23.45.21.32 GET /pypi/adlogin_page/v/dpkt.svg HTTP 1.1 Host: img.shields.io User-Agent: Mozilla/5.0 ... Firefox 54.0 Accept: */* Accept-Language: en-US;q=0.5,en;q=0.3 Accept-Encoding: gzip, deflate Cookie: __cfduid=dce101b3e09a9a09d5dbf2afed17dcdc71413792427910; timestamp: 13 Nov 2018; 10:00 Connection: keep-alive	(real) dst IP: 123.23.67.97 GET /pypi/adlogin_page/v/dpkt.svg HTTP 1.1 Host: img.shields.io User-Agent: Mozilla/5.0 ... Firefox 54.0 Accept: */* Accept-Language: en-US;q=0.5,en;q=0.3 Accept-Encoding: gzip, deflate Cookie: __cfduid=RXVybyBJRUVFIFMmUCaYMD E4IFN1Ym1pc3Npb24h Connection: keep-alive
a)	b)	c)

Figure 6.2: Example of honey traffic. Figure 6.2 a) represents an example of a message from an existing application, and Figure 6.2 b) depicts a honey message, which mimics an existing application but also embeds the honey tokens (in red), and Figure 6.2 c) shows a malicious message that adapts to the victim's traffic, hides its secret data (in blue), but also includes the honey token (in red), which the malware did not detect.

1. `"/adlogin_page/"`, as a subfolder of a URI path;
2. `"timestamp: 13 Nov 2018; 10:00"`, as a COOKIE parameter;
3. `"686897696a7c976b7e"`, as a ETAG value.

The honey server also generates: i) a list of random destination IP addresses [192.168.1.100, 23.45.21.32, 142.59.23.1]; and ii) signatures for a NIDS to identify each honey token. For instance, using Snort syntax, the following can be a signature for the first token:

```
alert tcp any any -> any 80 (msg:"HONEY token 1";
content:"/adlogin_page/"; http_uri; sid:2000001;)
```

Please note that signatures can also specify IP addresses, however, in this work we treat the verification of the destination IP as an additional step. The generated honey tokens, signatures and list of random destination IPs are shared with the honey client and the NIDS, as shown in Figure 6.1a.

Now assume that during the *detection phase* the honey client mimics a message of an existing application (e.g., message shown in Figure 6.2a), and it embeds two different honey tokens in the message. The message is then sent over the network to one of the random addresses: 23.45.21.32. Figure 6.2b shows how the message generated by the honey client may look like.

At this point, the security monitor, which analyzes each single message, finds two signature matches with the honey client (i.e., one per token). The monitor forwards the message to the honey server, which verifies that the destination IP is part of the set of random destination IPs generated during the setup phase. Hence, the message is considered to be generated by the honey client. Finally, the server responds to the client with a different response for each message, such that it is harder for the malware to identify which messages contained honey tokens. In other words, static responses may introduce obvious patterns that could alert

the attacker and would let her identify honey messages, and possibly, the honey tokens.

Now let us assume a malware uses an VAA covert channel to communicate with its server with IP 123.23.67.97. The malware sniffs the victim's traffic, and it observes the messages shown in Figure 6.2a and Figure 6.2b. It generates an adapted message, and it embeds the data to exfiltrate in the **Cookie** (see Figure 6.2c), and it sends it to 123.23.67.97. In this scenario, the security monitor finds one signature, because the token in the **URI** (i.e., `"/adlogin_page/"` in red in Figure 6.2c) is present in the message. The monitor forwards the message to the honey server, which marks the message as malicious because it has a destination address that does not match with those created during the setup phase. An overview of this whole process is shown in Figure 6.1b.

6.4.2 Generating Honey Tokens

Honey tokens, and therefore honey messages, are easy to generate but hard to distinguish from normal traffic. For this reason HONEYTRAFFIC is an effective defensive mechanism against VAA attackers, because they cannot reliably detect which messages contain tokens and which do not. In this section we discuss why it is hard for an attacker to reliably identify honey tokens in network traffic. Since our evaluation focuses on HTTP, we only analyze the possibilities of generating honey tokens on HTTP.

A honey token must be a sequence of bytes that is unlikely to appear in traffic generated by other applications. These sequences are not difficult to generate. We can use typos in header fields (e.g., **USER-Agent** instead of **User-Agent**), new header fields (e.g., **IEEE:**), fake header values (e.g., non-existing Firefox User-Agent version, or a non-existing domain), random values or any combination of these techniques.

Although there are several possibilities to generate tokens, the best locations are those headers that are commonly used, but are often associated with different values. If a header value often changes in benign traffic, it becomes difficult, even for an advanced attacker, to identify when a sudden change indicates the presence of a honey token. The headers that are commonly used in HTTP and have such properties are: **Host**, **Cookie**, **Referer**, **Body** and **URI**. For example, for **Host** we can generate a fake domain (or subdomain), or for the **Referer** we can generate a fake URL that was never requested before. These values are likely to be unique in the traffic. The values associated to each header field follow certain patterns. For example, **Host** accepts strings representing DNS hostname. Therefore, honey tokens should be formatted according to the same patterns, such that the attacker is not able to easily recognize the token by identifying the wrongly formatted string. Other headers that can be used for honey tokens are: time-based headers (e.g., **If-Modified-Since**, **If-Unmodified-since**, **Date**), where the tokens are represented by timestamps, or headers related to check resources updates (e.g., **If-Match**, **If-None-Match**, **ETAG**), where the tokens are represented as "random" sequences of alphanumeric characters. Table 6.1 shows some example of tokens that can be associated to HTTP headers, and it compares them with header values

HTTP Header or Field	Example Normal Traffic	Example with Honey Token
URI	/pypi/v/dpkt.svg	/pypi/adlogin_page/v/dpkt.svg
Host	www.google.com	honiee .google.com
Referer	https://google.com	https://google.com/ submission2018/I
Cookie	cfduid=dce101b3e09a9a09d5dbf2afe	uid= sckji13e1nksad
If-Modified-Since	Sat, 29 Oct 1994 19:43:31 GMT	Mon, 13 Nov 2018 10:10:11 GMT
If-None-Match	737060cd8c284d8af7ad3082f209582d	245221fe8c484j3xz3ad3991j298421d

Table 6.1: Examples, for some headers, of what can be represented as a honey token (in bold).

commonly observed in normal traffic.

Honey tokens can be automatically generated by the honey server, as long as the server is aware of the expected syntax for the header value. Following the aforementioned example, `Host` accepts strings representing DNS hostname. Therefore, the honey server can generate tokens by selecting English words from a dictionary, and it can use these words as subdomains, second level domains or even top level domains. For time-based headers, the server can generate random timestamps with the correct format, while for `URI` it can embed words from a dictionary as part of the URI path, query parameter or query values.

Generating honey tokens requires limited computational capabilities. Tokens can be chosen as random words or strings according to a predefined format. Despite being easy to generate, they are hard to detect. Without knowing how honey tokens look like, for an attacker is difficult (if not impossible) to effectively distinguish between strings representing honey tokens and normal traffic, because there are no clear patterns that differentiate these two string classes. Furthermore, CHAMELEON has shown how difficult it is even for defensive mechanisms, which use advanced heuristics and have access to high computational capabilities, to identify hidden data within HTTP traffic.

6.5 Evaluation

HONEYTRAFFIC relies on the assumption that, sooner or later, an attacker adapts to messages including honey tokens. Thus, the detection of VAA covert channels is not deterministic but probabilistic. We are not aware of existing malicious software using VAA covert channels. Consequently, it is not possible to verify the effectiveness of HONEYTRAFFIC traffic with realistic data. Thus, we decided to evaluate our approach using an hypothetical but realistic scenario. In our evaluation we make a set of assumptions that defines the scenario, and we estimate the detection performance, the relation between false positives and tokens, and the network and memory overhead of HONEYTRAFFIC.

6.5.1 Assumptions

We make the following assumptions: 1) both the attacker and the honey client mimic the traffic of a browser application, namely the *target application*; and 2)

the honey client hides honey tokens into a fixed number of header values.

Attackers are more likely to adapt to browser traffic, because browsers communicate often and they generate a lot of traffic. This gives the attacker more chances to adapt to the victim's traffic. Moreover, browsers represent the worst-case scenario for the honey traffic in terms of traffic overhead. The large volumes of browser traffic force honey clients to generate more honey traffic in order to detect a VAA attacker with a high probability.

Applications can use different sets of headers during their communication. However, there is always a subset of headers that is used in all applications' network messages, as we discussed in Section 3.4.2. Thus, a honey client can use such headers to hide tokens in it. For example, browser traffic always contains a `URI` (i.e., needed to retrieve a web resource) and a `Host` value (i.e., domain to contact). Both headers contain data that changes very often, making it hard for an attacker to detect potential tokens. For this reason, we believe is realistic to assume that a honey client can hide tokens in a fixed set of headers. Specifically, following the aforementioned example, we assume the honey client hides tokens in two different headers for each message.

In this evaluation we analyze honey traffic for a specific application. However, in practice, honey traffic should be generated for all applications installed on the host.

6.5.2 Attacker Strategies

The probability of detecting a VAA covert channel depends on the strategy an attacker uses to mimic application traffic. The attacker can mimic a message according to two different strategies: the attacker copies one of the previously observed messages from the target application entirely, and hides her secret data in one of the message headers (S1); or the attacker creates her messages by combining the list of elements (e.g., headers and header values) that were previously observed from the target application, and substitutes one header value with her secret data (S2). The latter strategy, S2, is the same we used to generate adapted messages with CHAMELEON (see Section 5.4.3).

In our evaluation we evaluate the effectiveness of honey traffic against both strategies. Since we assume an attacker cannot distinguish honey traffic from normal traffic, the attacker chooses the values to adapt at random. For this reason, *our evaluation assumes that the attacker mimics messages, or elements, uniformly at random.*

In the rest of the section we use the following terminology: c represents the collection timeout (see Section 5.4.3); A is the number of messages the target application generated during c ; and H_{msg} is the number of honey messages h_{msg} mimicking the target application during c .

Adaptation per Message (S1) Let us assume the attacker runs a collection phase for a period of time c , where it collects all the messages for the target application. A is the total number of messages generated by the target application, and H_{msg} is the number of honey messages mimicking the target application.

The attacker cannot distinguish between A and H_{msg} , so she collects both. The attacker adapts to the target application by randomly choosing one of the collected messages. Then, the attacker hides the data and sends the message, according to the message parameters MP she uses.

The probability p that the attacker chooses one of the honey messages with this strategy, which allows us to detect the attack, is:

$$p_{S1} = \frac{H_{msg}}{A + H_{msg}}. \quad (6.1)$$

Consequently, the probability of detecting an attacker after sending x messages, can be modeled as a binomial distribution, and it can be defined as

$$P(\text{detection within } x \text{ messages}) = 1 - (1 - p_{S1})^x. \quad (6.2)$$

Please note that in a VAA covert channel, the secret data is hidden in a single header. Therefore, it is not possible for the attacker to overwrite all the tokens, since they are stored in two (or more) different headers.

Adaptation per Header (S2) Let us assume the attacker runs a collection phase for a period of time c , where it collects: a header set $L = \{L_1, \dots, L_j\}$ where L_i represents a list of headers; and a list of values V_{head} containing all the values associated with header $head$ in the observed traffic. V_{head} also contains the observed honey tokens generated for header $head$ during c , which we define as t_{head} . After choosing a list of headers L_i , the attacker generates an adapted message by randomly choosing a header value from V_{head} for each corresponding $head \in L_i$. Then, the probability p_{S2} to detect a VAA malicious message with this strategy is

$$p_{S2} = 1 - \prod_{head \in L_i} \left(1 - \frac{t_{head}}{|V_{head}|}\right). \quad (6.3)$$

Similarly as for S1, the probability of detecting an attacker after sending x messages, during blending phase, can be modeled as a binomial distribution, and it can be defined as

$$P(\text{detection within } x \text{ messages}) = 1 - (1 - p_{S2})^x. \quad (6.4)$$

A comparison between Eq. 6.1 and Eq. 6.3, already highlights how S2 is less favorable for the attacker, in case more tokens are associated to different headers, since there are more chances for her to include a token in her messages.

Parameters

We evaluate the effectiveness of honey traffic against the two adaptation strategies (i.e., S1 and S2) in different scenarios, assuming the target application is a browser.

In order to estimate how many requests a web browser may generate within a certain period of time, we collected one hour of browsing activities (e.g., streaming videos in the background and moderate web browsing) from a single machine

Figure 6.3: Probability of a VAA attacker being detected by the number of messages it sends, for both strategy 1 (red lines) and strategy 2 (blue lines). The percentage values represent the overhead of honey messages, compared with the amount of browser traffic.

controlled by us, using BurpSuite ²³. We observed that roughly 2,200 requests were generated.

Consequently, within a collection timeout $c = 30s$, the number of requests a browser may generate within c , which is also collected by the attacker, is 19. Therefore, we defined $A_c = 19$. We choose a short collection timeout because malware may want to communicate closely to the benign application to avoid suspicion. Since we assume that there are specific headers always present in application traffic, we define $|V_{head}| = A + t_{head}$, because the values observed from a single header contains all those generated by the normal application (i.e., A) and those generated by the honey client containing a token (i.e., t_{head}). Finally, we evaluate our mitigation according to different amounts of honey messages generated during c . We define four cases, where the honey client generates 1, 2, 5, and 10 messages. The network overhead introduced by the honey client for these values is approximately 5%, 10%, 20% and 50%, respectively (e.g., for 19 browser messages in c , we introduce 1 honey message, which represent roughly the 5% of the browser traffic).

We want to remark that, although we covered the two strategies an attacker may use to mimic applications traffic, the detection probabilities are still influenced by the number of messages generated by the target application and the honey client, the collection timeout, and how tokens are hidden. In this work, we evaluate our honey traffic approach only with the parameters mentioned above to estimate the practicality of this detection method.

²³<https://portswigger.net/burp/communitydownload>

³We installed a certificate on the machine to read also HTTPS messages.

Results

The results show that S2 is less convenient strategy for an attacker compared to S1, as depicted in Figure 6.3. Nonetheless, independently from the strategy, the attacker is detected after few messages. In case of S1, the attacker can be detected with 80% probability after she sends only 16 messages, costing only 2 honey messages every 30 seconds. If only one honey message is sent (i.e., 5% overhead), the attacker can still be detected with 80% probability after 30 messages. This makes it problematic for an attacker to persist on the host, because she cannot perform many malicious operations before she is detected. Another fundamental characteristic of HONEYTRAFFIC is that it generates almost no false positives, as long as tokens are properly chosen, because it works like a signature. Moreover, we can choose what patterns the signature should match.

Unsurprisingly, the increase of honey messages allows a faster detection. By multiplying the number of honey messages (within c) with the average size of an HTTP request, we can estimate how many kb/s our solution costs. Let us assume a pessimistic scenario, where the average size of a message is 1kB. By using 0.03kB/s (i.e., 5% overhead, 1 request in 30 seconds) of extra bandwidth, the attacker is detected within her first 43 messages in case of S1 and 23 messages in case of S2. The overall cost in a day of extra bandwidth is 2.59MB, for each browser we want to mimic in the network. Moreover, this is a worst-case scenario because this extra bandwidth is needed if the browser is active for 24 hours. If we consider the overhead to be 50%, which is the worst-case scenario for honey traffic due to the high number of messages it generates, the bandwidth cost in a day is 28.5MB, for a single browser.

The memory costs for the honey client are represented by the amount of information collected from the machine traffic, which is needed to establish VAA covert channels, and the list of honey tokens. The costs for collecting network information have been discussed in Section 5.5.5. The memory costs for storing the list of tokens are relatively low. We evaluate a scenario where the setup phase waits one week to refresh the tokens, each token is used only once and is 10 bytes long. These are pessimistic settings since many tokens should be generated and hold in memory for a long time. For a bandwidth overhead of 50%, storing the tokens approximately costs 4,032 kB. A lower bandwidth of 5% has 403 kB of memory costs. Table 6.2 shows the costs in terms of storage of the honey traffic. In case the setup phase is executed more often, the memory costs decreases. For instance, a setup phase refreshing every day would cost 576 kB in case of a 50% bandwidth overhead, and of 57 kB only in case of a 5% bandwidth overhead.

This evaluation assumes the attacker adapts every time she sends a message. In case the attacker would send multiple messages with the same collected data, our evaluation still holds, but the X-axis of Figure 6.3 should be interpreted as “adaptation attempts”, instead of messages.

Moreover, our analysis assumes browser messages are evenly spread across different collection phases. We do not expect this to happen in practice, because the browser generates bursts of messages. This implies that some collection phases are more favorable to the attacker, and some to the defender. Nevertheless, honey

Table 6.2: Memory costs for the honey tokens, assuming each token is used only once (i.e., one-time tokens) and the setup phase is repeated once a week. We consider tokens to have an average size of 10 bytes.

Bandwidth Overhead	Unique Tokens	Storage Size [kB]
5%	5760	403
10%	11520	806
20%	28800	2,016
50%	57600	4,032

clients can also generate bursts. Therefore, since the attacker is not able to recognize in which collection phase there are honey messages, the attacker cannot take advantage of it.

6.5.3 Tokens and Potential False Alerts

Honey tokens are supposed to be unique strings, because if they occur in normal traffic they would trigger false alerts. One of the main factor that can influence the uniqueness of tokens is their length, the shorter the token, the more likely it is to match strings in benign traffic. Thus, we decided to investigate what should be the length of a token in order to have low probabilities of false alerts.

We analyzed tokens from three to eight bytes long. Although there are many different ways of generating tokens, such as combining words from dictionaries or generating format-specific strings (e.g., timestamps), we decided to use the simplest method of all: generating random strings. Hence, for each of these lengths, we randomly generated 5000 unique tokens using alphanumeric characters and digits. We then verified how many of these tokens would trigger *at least* one false alert, for each of the 45 samples of the CHAMELEON dataset. For this evaluation we considered only the baseline traffic, and we excluded the VAA traffic from the dataset.

In other words, for each day of traffic we evaluated how many tokens were not unique. This would represent a lower bound of false alerts. Figure 6.4 shows the average amount of non-unique tokens we observed in the whole dataset for different token lengths. The results show that tokens longer than 6 bytes are all unique in our dataset, meaning that they would trigger no false positives. The uniqueness of tokens follows an exponential distribution, suggesting that the longer the token becomes, it becomes much harder to find a match in normal traffic. Our analysis shows that tokens cannot be shorter than 6 bytes, since many matches have been found in our small dataset. On the other hand, we cannot say that 6 bytes is the ideal length for a token, because our dataset was not large enough to represent a real network. However, considering that the uniqueness of tokens follows an exponential distribution, we can expect few matches in larger network for tokens of approximately 10 bytes. Finally, we should consider that in this evaluation we performed a string search in the whole message. However, in practice, honey tokens may be hidden in specific parts of the message (e.g., as URL query parameter),

Figure 6.4: Average amount of non-unique tokens observed in CHAMELEON traffic samples. For each token length, we randomly generated five thousand tokens. We counted how many of these tokens were present at least once in each traffic sample, which represents one day of traffic for a single host. Finally, we computed the average of occurrences of non-unique tokens.

reducing the chances of triggering false alerts.

6.6 Limitations

6.6.1 Full vs Partial VAA Data Exfiltration

As we explained throughout this chapter, HONEYTRAFFIC can detect VAA covert channels that use similar techniques as those used by CHAMELEON. More specifically, HONEYTRAFFIC *can detect VAA covert channels that generate messages using a template containing only data from the victim*. Let us call these VAA covert channels *full VAA* (F-VAA). In this setting, there is a chance for the attacker to include a honey token in its messages. Therefore, HONEYTRAFFIC can detect the attacker.

HONEYTRAFFIC may not work in case the attacker *partially* mimics the victim's traffic, meaning that she generates a template containing some victim's data and some data generated by herself (i.e., *partial VAA*). In this case, it is possible that honey tokens are never included in the adapted messages, because the attacker used her own data for that specific header value. Thus, HONEYTRAFFIC works as long as the headers values that should contain honey tokens are not overwritten by the attacker data.

Nonetheless, it remains unclear whether an attacker substituting several header values with randomly generated valid data reduces her chances to mimic honey tokens to zero. Moreover, it remains also unclear if the generation of these strings may

Detection Technique	Exfiltration Techniques			
	No Adapt.	APA	F-VAA	P-VAA
Existing Solutions	✓	✓	×	?
HONEYTRAFFIC	×	×	✓	?
Existing Solutions + HONEYTRAFFIC	✓	✓	✓	?

Table 6.3: Overview of what type of exfiltration techniques are addressed by existing detection techniques. We describe four types of adaptation: no adaptation, a priori adaptation (APA), full victim-aware adaptation (F-VAA), and partial victim-aware adaptation (P-VAA). All the rows contain a question mark for P-VAA, because now one has evaluated the case of partial adaptation.

introduce distinctive patterns that could be modeled by existing covert-channel detection solutions, thereby exposing the attacker to detection. Further investigation is required to understand what are the limits of partial VAA covert channels.

Due to these limitations against *partial* VAA covert channels, HONEYTRAFFIC cannot be considered a standalone solution to detect VAA covert channels. However, HONEYTRAFFIC can complement existing detection methods to make evasions more difficult for VAA covert channels and provide overall better network protection. Table 6.3 shows the threat scenarios covered by HONEYTRAFFIC compared with existing solutions, such as DECANTeR or HEADPRINT, and with the combination of both techniques.

6.6.2 Lack of Existing Malware Samples

The lack of malware samples using VAA covert channels, does not allow us to evaluate our solution with real data. An evaluation using a synthetic dataset containing both benign and honey messages can be performed, in order to verify after how many messages a VAA covert channel would have been mimicked. However, the only implementation we are aware of for this attacker is CHAMELEON, which is developed by us. Thus, we considered this option to be sub optimal. Consequently, we decided to make an estimate of HONEYTRAFFIC performance by evaluating with an hypothetical, but realistic, scenario. In this scenario, our estimates give an indication that HONEYTRAFFIC can be a practical solution with low network and memory overhead.

6.6.3 Evasion

Honey traffic cannot be proven secure, because we assume the attacker can run with high privileges, so, *in theory*, the malware can do anything on the compromised host. For instance, the attacker may be able to identify the process of honey client at system level and avoid mimicking packets that it generates. Nonetheless, we believe an attacker, especially if automated such as malware, faces severe difficulties in identifying the honey client, even more so if the client is properly hidden (e.g., by using rootkit techniques).

The setup phase of HONEYTRAFFIC is also a limitation, because the attacker can get to know the list of tokens that will be used. However, also in this case, the attacker must be aware of the presence of the honey client on the system. Moreover, she should also be able to intercept and interpret the messages provided by the server.

The attacker can try to evade our detection mechanism generating its own traffic. As we discussed in Section 6.6.1, partial VAA can be a potential solution for attackers to evade HONEYTRAFFIC. However, it is unclear whether a similar technique would also be effective against existing detection mechanisms. In case the attacker generates its own traffic, she may introduce patterns that could expose her to detection. Introducing patterns is exactly what VAA data exfiltration wants to avoid.

The attacker can try to mimic traffic according to a different distribution than by mimicking messages at random. For instance, the attacker may learn the set of most visited websites and mimic only messages sent towards those websites. The limitation of our work in this regard is that we did not analyze the scenarios of different mimicking distributions. However, such techniques can reduce the chances of detection, but cannot guarantee evasion. Since honey messages are randomly generated, they may also mimic a commonly visited website.

Lastly, an attacker can evade honey traffic by creating messages without using victim information. However, by doing so the attacker is not adapting, thus she would be detected by already existing techniques, such as DECANTeR or HEAD-PRINT.

6.7 Conclusions

In this chapter we proposed a novel detection strategy for a specific VAA data exfiltration attacks, which we call Full-VAA and this type of attack is the same implemented by CHAMELEON. Full-VAA data exfiltration attacks where the attacker tries to maximize the similarity of her messages with the victim's traffic by mimicking as much information as possible. We named our approach HONEYTRAFFIC, since it involves the generation of fake network messages that hide honey tokens, and aims at deceiving the attacker and let her adapt to the fake messages. The presence of a VAA malicious software can be identified by a security monitor that is aware of the honey tokens. We evaluated our approach under certain assumptions and we have shown that it can detect VAA data exfiltration with a negligible overhead on the network bandwidth. Since the approach is essentially a signature-based detection solution, it generates few false alerts.

In Chapter 3 and 4 we have proposed to use application fingerprinting to detect data exfiltration attempts performed by non-adaptive attackers. In Chapter 5 we have described the threat of VAA data exfiltration and we have shown that existing detection solutions are not adequate to effectively detect such threat. In this chapter we proposed a detection mechanism that can detect a specific type of VAA data exfiltration, namely Full-VAA, thereby addressing **RQ2**. The detection of other types of VAA attackers, such as Partial VAA, where the attacker mimics

only “some” victim’s network information, requires different detection methods.

This chapter concludes our work on the detection of data exfiltration. In the next chapter, we discuss how can we determine what has leaked after data exfiltration has been detected.

Chapter 7

Impact Evaluation of Data Exfiltration by Determining What Has Leaked¹

In the previous chapters we have addressed the research questions related to the detection of data exfiltration. This chapter focuses on the impact evaluation of data exfiltration. As we discussed in the introduction, existing solutions do not allow for a precise evaluation of the impact of data exfiltration. Solutions such as logging systems cannot be used for impact evaluation, because they can be tampered by an attacker. In this setting, an investigator has to assume that *all* data may have leaked, because he cannot prove otherwise. Alternatively, there exist secure cryptographic systems that allow computations over the data while never revealing the data in cleartext. Using these systems, *nothing* leaks. Unfortunately, these systems are practical only in very limited settings, due to their high computational costs.

In this chapter we address **RQ3**, and we propose an approach capable of determining data leakage that is not just *all-or-nothing*. Our system provides tamper-resistant logs against adversaries capable of taking complete control of a machine. The log also stores all data access information needed to determine what may have leaked.

The main limitation of existing logging solutions is that each machine is solely responsible for generating log content. Therefore, if the machine is compromised, there is no way to distinguish whether the new log content is tampered by an attacker or not. The main idea of our work is to shift the responsibility for the generation of log content from a single machine to a coordinated group of machines. Thus, even if one or a subset of machines is compromised, it is still possible to obtain a copy of a log that has not been tampered.

While this responsibility shift thwarts an attacker to tamper or to inject information in the logs, the nodes should also be aware of all accesses to cleartext data,

¹This chapter is an edited version of a peer reviewed paper that appeared in ACSAC '16 [4]

otherwise data can be leaked without being reported. The idea to solve this issue is to leave data encrypted at rest, and to decrypt it only when a computation is requested at a random node. Using secret sharing we ensure data can be decrypted only through a collaboration of machines, meaning that multiple nodes acts as a decentralized access control mechanism, thus, they are aware of all data accesses. Moreover, we randomly choose a node for the computation so an attacker cannot predict where data can be decrypted.

Our approach does not rely on trusted hardware, but on a combination of existing cryptographic primitives with a Byzantine consensus protocol. We demonstrate for the specific application setting of client-server authentication that our distributed log is feasible in practice, and that it can be integrated with existing services.

7.1 Motivation

The law requires that a company has reliable information in order to correctly understand what data has been leaked after a data breach. Existing work on data leakage focuses on prevention by providing Data Loss Prevention solutions [138]–[140]. However, these solutions cannot be used to determine the data leakage in case the prevention fails.

The most promising existing solutions are *provenance-aware* systems [141]–[143], which record the entire data lifecycle including its origin and where it has been sent. Unfortunately, these systems make strong assumptions on what the attacker cannot do (e.g., the attacker cannot access the OS kernel), which is unrealistic given the occurrences of sophisticated malware in data breaches [22], and the recent increase of rootkit development [23].

This lack of solutions against such powerful attackers, has only been recently addressed by Bates et al. [52] who proposed LPM, a Linux kernel module capable of collecting and protecting evidence against such a powerful attacker. Unfortunately, LPM’s security mainly relies on trusted hardware and it is platform dependent. Trusted hardware, which is invasive because it imposes significant changes to the infrastructure, often is not a feasible option for companies. This leaves companies without a solution to create evidence to determine data leakage.

On the other hand, there exist cryptographic techniques, such as fully homomorphic encryption (FHE) [50] and secure multi-party computation (MPC) [51], which allow data processing in the encrypted domain. Applications using these techniques do not require digital evidence since data is protected by cryptographic means. Therefore, determining the leakage is trivial, because *nothing* leaks to an attacker. However, these techniques are computationally expensive and only suitable in limited practical scenarios. Consequently, most of today’s data breach victims have no other option than considering *all* data to be leaked.

For these reasons, we decided to investigate an alternative logging solution that can reliably and precisely determine what data has leaked, without relying on trusted hardware.

7.2 Intuition

First let us assume that a central server is responsible for creating and maintaining a log file with all accesses to data. If this machine is compromised by a powerful attacker, for instance by using a kernel rootkit, there is no solution to protect the evidence from being tampered or deleted by the attacker, unless trusted hardware is used [52].

Now let us assume a decentralized logging setting with two machines, A and B . A sends its data access information to B and B stores the log for A . If an attacker compromises A , she can send fake information to B . There is no way for B to distinguish real from fake information, so the log may store fake records, which makes the log content not trustworthy. This is also true in case there are n machines and A sends its information to all the other machines $N_1, \dots, N_{n-1}$. As long as a machine is solely responsible for the content of its own evidence, and this machine can be compromised, it is not possible to rely on a logging system to determine data leakage. All common logging mechanisms work as just described.

Our approach *inverts* this process. It is not A to tell the $n - 1$ other machines what it had access to, but the $n - 1$ machines tell A what it is going to access. So, the $n - 1$ machines store in the log file that A had access to a specific data item. This approach can be considered a generalized digital analogue of the *four-eyes principle* (n -eyes in our case), which requires two individuals to approve some action before it can be taken. We enforce this approach by requiring data to be encrypted at rest and to decrypt it (making it accessible) only when it is needed for processing. The decryption step can be done only through the collaboration of multiple machines, which have to agree on the decryption and, if so, write in the log who is going to access the data.

7.3 Building Blocks

In this section we introduce the building blocks used in our approach and discuss the choice of the specific implementations.

7.3.1 Byzantine Fault Tolerant Consensus Protocol

Byzantine fault tolerant consensus protocols are a form of state machine replication, such that a set of n distributed nodes can behave consistently as a centralized implementation executing operations atomically one at a time. These protocols survive f Byzantine failures. We decided to use PBFT [144] because it is a widely accepted in practice and because it provides strong consistency guarantees despite a communication complexity of $\mathcal{O}(n^2)$. However, other Byzantine fault tolerant consensus protocols can be used (e.g., [145], [146]). In PBFT there are $n = 3f + 1$ nodes, and they move through a sequence of configurations called *views*. Within each view one node is the *primary* node. The primary node is responsible to forward client requests for operations to other nodes and to propose the order of their execution. A request is executed if at least $2f + 1$ nodes have accepted it. When a request is executed, a *reply* message is sent to the client. When the client receives

$2f + 1$ reply messages, she considers the operation to be successfully executed. Moreover, nodes periodically generate *checkpoints*, which are proofs confirming that at least $2f + 1$ nodes have executed the same number *seq* of requests. In case the primary node is considered compromised, all nodes start a *change-view* procedure, which elects a new primary node.

7.3.2 t -out-of- n Secret Sharing

Secret sharing (SS) is a cryptographic primitive that allows any party to split a data item s into a set of n shares, such that any combination of t shares, where $t \leq n$, can be used to reconstruct s . In our case we consider $t = f + 1$ and $n = 3f + 1$. The data item s can be recovered only if at least $f + 1$ shares are combined. Therefore, until at least one correct node reveals its share, the attacker learns nothing about s . A secret sharing scheme provides the following algorithms:

- $\text{Split}(s) \rightarrow \{s_1, \dots, s_n\}$ generates a set of n shares from a data item s .
- $\text{Combine}(\{s_i\}_{i=1}^{f+1}) \rightarrow s$ combines a set of shares to obtain the original data item s (or $\perp$ if there are less than $f + 1$ valid shares).

In this work we propose to use AONT-RS [147] for its performance. Compared to Shamir Secret Sharing [148], AONT-RS provides better storage efficiency² and faster sharing and reconstruction of secrets. AONT-RS is *computationally* secure (based on AES), and does not require a setup phase to distribute encryption keys, in contrast with other secret sharing schemes.

7.3.3 Commitment Scheme

A commitment scheme is a cryptographic primitive that involves two parties: a *committer* and a *receiver*. The committer can commit to a specific value x , without revealing x to the receiver (i.e., *hiding* property), unless the committer itself reveals it. Once the committer commits to x , she cannot change her value (i.e., *binding* property). A commitment scheme provides the following algorithms:

- $\text{Commit}(x) \rightarrow c$ generates a commitment message for a value x .
- $\text{Reveal}(x, c) \rightarrow 1/0$ verifies if x is the committed value for c .

In this work we use the simple hash-based construction that commits to a value x by computing $H(r||x)$ for a random r and a cryptographic hash function $H(\cdot)$ (modeled as a random oracle). However, other schemes might be used as well.

7.3.4 Digital Signature Scheme

A digital signature is a cryptographic primitive that allows a *signer* to authenticate a message, and a *verifier* to verify the authenticity of that message. A digital signature scheme provides the following set of algorithms:

²For example, given a 4 KB data block to divide 10-out-of-16 shares, Shamir Secret Sharing requires 64 KB, while AONT-RS needs 6,45 KB [147].

- $\text{KeyGen}(1^\lambda) \rightarrow (pk, sk)$ generates a signing key pair.
- $\text{Sign}(sk, m) \rightarrow \sigma_m$ generates a signature for a message m .
- $\text{Verify}(pk, m, \sigma_m)$ verifies a signature on a message m .

In this work we use RSA. Other digital signature scheme can be used as well.

7.3.5 Hash Chain

A hash chain is a consecutive application of a cryptographic hash function $\mathcal{H}$ to multiple data items that binds all of them to a single value. Given a family of events $\{E_i\}_{i=1}^l$, the value stored in the hash chain for each event E_i , is $L_i = \mathcal{H}(L_{i-1} || E_i)$, where $L_{i-1} = \mathcal{H}(L_{i-2} || E_{i-1})$. Each new entry is directly linked to the previous event, and the base hash L_0 is a predefined value.

7.4 System and Threat Model

Consider a scenario where users can upload data to a company infrastructure, which can run computations over this data either on the user's or system's demand. The company is responsible for the data storage and the computation. Moreover, it wants to deploy a mechanism capable of *reliably* determining the data leakage in case its infrastructure is compromised.

We consider the company's infrastructure to be a distributed asynchronous system composed of n nodes, which log and execute the operations requested by clients. The set of possible operations is determined by the specific application. Each node has an append-only log, which describes all operations executed by the system and can be used to determine the data leakage. The nodes are connected by a network that may: fail to deliver messages, deliver them out of order, duplicate them, or delete them. We assume a Byzantine failure model, where *faulty/compromised* nodes might behave arbitrarily (e.g., a compromised node does not necessarily follow the protocol). Nodes are *non-faulty/non-compromised* if they behave according to the protocol. Our approach guarantees that clients eventually receive replies to their request (*liveness* property) and those replies are correct according to linearizability (*safety* property) if we assume that: the attacker can compromise at most f machines, where $f = \lfloor \frac{n-1}{3} \rfloor$; the attacker is computationally bounded, so she cannot break cryptographic primitives; there is a delay T between the first time a message is transmitted and when it is received by a non-compromised node, where T does not grow indefinitely [149]. The latter assumption appears to be reasonable in practice if network faults are quickly repaired. These are common assumptions for Byzantine fault-tolerant consensus protocols [149]. Finally, we assume the company deploys security solutions that can detect compromised machines and determine the period of compromise (e.g., IDSs).

Attacker model. We consider a powerful attacker that can take complete control over the machines he compromises by using a kernel rootkit. However, we

assume compromising a machine is time consuming and a difficult task even for a skilled attacker. On these machines the attacker can tamper, disable or permanently delete any possible evidence. The attacker is allowed to compromise any machine and her choice may depend on previous information. The attacker may compromise up to f machines, where the total number of nodes of our system is defined as $n = 3f + 1$. This assumption is fundamental in order to achieve Byzantine consensus. Finally, the attacker can also coordinate the compromised machines to disrupt or to compromise the service.

7.5 Security Requirements for a Reliable Log

A log needs to fulfill the following security requirements to be considered *reliable* to determine the data leakage:

- **Tamper-Resistance:** any unauthorized modification of the log, such as addition, removal or content modifications of a log entry e_i , should be detectable.
- **Availability:** it should not be possible to remove all copies of the log from the system.
- **Trustworthiness:** the log of a non-compromised node should contain only events that have been requested by legitimate events. An attacker should not be able to convince non-compromised nodes to log events that were never requested. The difference to *tamper-resistance* is that the attacker may try to remotely influence the log of *other* nodes by hijacking the protocol.
- **Completeness:** the log should store all the accesses on plaintext data. An attacker should not be able to access any plaintext data item without being recorded by non-compromised nodes.

7.6 A Distributed Secure Log

As follows from our high level description in Section 7.2, we infer two important requirements: 1) a single node cannot access data without the consent of other nodes, and 2) a group of nodes should agree on when and what data a node can have access to. We address the first requirement using threshold cryptography, which enforces that t shares are needed to access the data. A lower number of shares does not reveal any information about the data. We address the second requirement using a Byzantine consensus protocol, guaranteeing consistency in the execution of operations among distributed nodes, despite (at most) f of them may be compromised. With this protocol, all nodes can agree on which data items a specific node can access, send the correct shares to the same node, and log the access to the data consistently with the other nodes.

As a consequence of the liveness and safety properties of a Byzantine consensus protocol (see Section 7.4) this setup requires $n = 3f + 1$ machines, where f is the

number of compromised nodes. Considering this strict requirement, we define the threshold for Secret Sharing as $t = f + 1$. This is the lowest value that guarantees that the attacker cannot read the data, since the system assumes at most f compromised machines.

We now discuss how the log is created and how we can determine the data leakage using the log.

7.6.1 Creating Evidence of Data Access

We assume a finite universe of clients that can request to upload data, or to perform computations over data stored in a distributed system $\mathcal{N} = \{n_1, \dots, n_n\}$ composed of n nodes, where $n = 3f + 1$, and f is the maximum number of nodes that can be compromised by an attacker. All clients and nodes have a public-private key pair $(pk, sk) \leftarrow \text{KeyGen}(1^\lambda)$. All public keys pk are known to each node. Node keys are securely shared when the system is deployed, while client keys are securely shared at the time of client registration to the system. We assume the attacker is not present at registration time, and we assume clients to be *trusted*. We note that this is a simplified setup that considers clients to be trusted and authenticated. In the next sections we also discuss our approach for scenarios with *untrusted* clients.

Requests Need to be Accepted and Logged

Clients interact with the distributed system by sending *upload* or *query* requests. These requests have to be accepted and logged by each node before being executed. This process is performed in the three-phase protocol of PBFT, as shown in Figure 7.1a.

In both cases, a client generates a request REQ , which describes the operation the client would like to perform. She generates a signature for the request $\sigma_{\text{REQ}} \leftarrow \text{Sign}(sk, \text{REQ})$, and she sends the signed request $\langle \text{REQ}, \sigma_{\text{REQ}} \rangle$ to the *primary* node, which then forwards it to all nodes (i.e., *pre-prepare* phase). Each node accepts $\langle \text{REQ}, \sigma_{\text{REQ}} \rangle$ if $\text{Verify}(pk, \text{REQ}, \sigma_{\text{REQ}}) = 1$ and if the message is formatted as specified by the protocol, otherwise it is discarded. Furthermore, depending on the application setting, nodes can apply additional checks (e.g., verify if the user is allowed to perform request on some particular data). Finally, PBFT requires additional verification steps during the execution of its protocol, which are independent from our solution. We refer to the original work for further details [144].

If $\langle \text{REQ}, \sigma_{\text{REQ}} \rangle$ is accepted, nodes broadcast their decision to other nodes (*prepare* and *commit* phases). When a node receives $2f + 1$ responses during the *commit* phase, it logs $\langle \text{REQ}, \sigma_{\text{REQ}} \rangle$. Each log entry $e_i = (i, \langle \text{REQ}, \sigma_{\text{REQ}} \rangle, L_i)$ has a sequence number i , the content of a request along with its signature $\langle \text{REQ}, \sigma_{\text{REQ}} \rangle$ and a recursive hash value $L_i = \mathcal{H}(L_{i-1} || \text{REQ})$. L_i links REQ to its previous request.

Once e_i is stored, each node sends a signed *reply* message back to the client. The client considers REQ to be successfully logged if it receives at least $2f + 1$ signed reply messages from distinct nodes. At this point, REQ is executed.

Figure 7.1: Overview of the interaction between a client and four nodes: a) shows the common process for each request, which is validated and logged through the PBFT protocol [144], where n0 is the primary node; b) shows how data is uploaded once the request has been logged, where the client t -out-of- n secret shares a secret s , and sends each share s_i to each node; c) shows how the computation is performed once the request has been logged, where the execution node (n1) receives the shares needed to reconstruct the data, performs the computation and sends back the result to the client.

Upload Requests

Clients can store new data items on the nodes or update existing items. An *upload request* $\text{REQ}_U = \langle \text{ID}, ts, \sigma_{\text{REQ}_U} \rangle$ has an identifier ID of the item the client wants to upload, a timestamp ts and a digital signature σ_{REQ_U} . The request is sent to the nodes for approval and logging, following the process discussed above.

After receiving $2f + 1$ *reply* messages, the client runs $\text{Split}(D) \rightarrow \{s_1, \dots, s_n\}$ and generates n shares for the item D that has ID as identifier. Finally, as shown in Figure 7.1b, the client sends to each node n_i a message $\langle \text{ID}, s_i, ts, \sigma_d \rangle$ containing: the identifier ID, a share s_i of D , a timestamp ts , and a signed digest σ_d where $d = \mathcal{H}(\text{ID}, s_i, ts)$.

Each node accepts the share s_i if the following verification steps succeed (otherwise, the node discards the share): 1) compute the digest d from the received message and check that $\text{Verify}(pk, d, \sigma_d) = 1$, and 2) check if a request REQ_U with same ID and from same client was already logged. If another share s_i is stored with the same ID, it is overwritten with the new share. Otherwise s_i is stored as a new item with identifier ID.

Although the information stored in REQ_U does not influence the determination of data leakage, it is stored in the log to avoid possible data inconsistency in case of concurrent updates. For instance, if two clients are concurrently trying to update the same data item, all non-compromised nodes can decide, based on the log, to

keep only the shares of the latest request they executed.

Replay attacks are possible, but do not affect the determination of data leakage and can be prevented by integrating orthogonal techniques in the protocol (e.g., nonce).

Upload by Nodes. Non-compromised nodes are not allowed to upload data, unless a client requests it. A non-compromised node uploads data only if there is a *query* request REQ_Q from a client that specifies a *write*(ID) command. The details of query requests are discussed below. A node upload request REQ_U is generated and processed through PBFT as a client's request. The identifier ID for the new item is the one specified in the *write*(ID) command.

A node accepts the request if: 1) after computing the digest d from the received message, $\text{Verify}(pk, d, \sigma_d) = 1$; 2) there exists a query request REQ_Q that specifies a *write*(ID) command such that ID matches the identifier in REQ_U ; and 3) the node that generated REQ_U is the same node that executed REQ_Q , and no upload requests have been accepted yet for that write command. The rest of the protocol follows as in the client case.

Upload failure. It is possible that REQ_U is logged, but the shares are not sent or are delayed. Once REQ_U is stored, each node starts a timer. All following requests that involve the uploaded item are logged and queued. Once the shares have been successfully received, requests are executed on the new item. If the timer reaches a globally set timeout value, the log is "rolled-back". REQ_U is removed from the log, the hash chain is updated and the following requests are executed on the old item.

Timeouts can be exploited by an attacker to degrade the performance of the system (e.g., wait for shares and delay computations), however these denial of service attacks do not influence the determination of data leakage.

Query Requests

Clients can perform computations over data stored on the nodes. A *query request* $\text{REQ}_Q = \langle \mathcal{D}, O, r, ts, \text{command}, \sigma_{\text{REQ}_Q} \rangle$ has a set $\mathcal{D}$ of unique data item identifiers, an operation O to apply to the data, a uniformly distributed random number r , where $0 \leq r \leq n - 1$, a timestamp ts , a command *read* or *write*(ID), and the signature σ_{REQ_Q} .

$\mathcal{D}$ is the set of items requested for the computation. O is any possible computation allowed by the application. r is a random number used to select the *execution node*. This node is responsible for reconstructing the data from the shares, performing the requested computation and returning the result to the client. The execution node for a request is the node $n_i = r$. The command *write*(ID) is used by a client when she wants to retrieve the result of the computation and then store it in the system with the specified identifier ID. A client uses a *read* command if she only wants to retrieve the result.

REQ_Q is validated and logged as discussed at the beginning of the section. If the request is accepted, each node retrieves the set of requested shares $\mathcal{S} =$

$\{s_1, \dots, s_m\}$ where $m = |\mathcal{D}|$ (i.e., one share per item). Each node sends $\mathcal{S}$ to the execution node. The execution node waits until it receives at least $f + 1$ set of shares (including its own). It reconstructs each requested item in $\mathcal{D}$, such that $\mathcal{D} \leftarrow \text{Combine}(\{s_i\}_{i=1}^{f+1})$. The execution node performs the requested operation O over the reconstructed set of items. Lastly, it sends the result of the computation to the client. This process is shown in Figure 7.1c. The execution node deletes all the received set of shares and the reconstructed data. Finally, it verifies whether a `write(ID)` command is specified in the query request or not. If so, the result needs to be uploaded through an *upload* request.

It is worth noting that if k shares are generated and uploaded instead of n , where $k < n$, the attacker may be able to block the execution of a query request by not sending the share she stores. We choose to generate n shares to guarantee the functionality of the system in a Byzantine model.

Execution Node Unavailable. In case the execution node is not available at the time the shares are being transmitted, all nodes run an instance of the collaborative protocol to select the execution node (discussed below). This selection protocol allows distributed nodes to agree on selecting a random execution node. The shares are then retransmitted to the new execution node.

The execution node might also fail to deliver the result of the computation to the client after it received the shares. For instance, the attacker controls the execution node of a specific request and wants to disrupt the service. Therefore, a client expects a response from the execution node within a predefined timeout. If the timeout is exceeded, the client generates a new query request with a different execution node.

A compromised execution node can send wrong computation results to the client. Overcoming this problem is not the goal of this work. Nonetheless, this does not affect the determination of leakage provided by our solution, because the nodes already logged all the data the (in this case) compromised execution node had access to. This happens independently from a successful or unsuccessful execution.

Untrusted Clients

In certain application scenarios, we cannot assume clients to be trusted. When assuming clients to be *untrusted*, we cannot rely on the authentication based on cryptographic keys.

This affects our approach in three ways: 1) a compromised primary node can successfully execute requests that were never generated by a client, because nodes cannot verify a request's authenticity; 2) a compromised node can generate fake client requests, and append them in the log while maintaining a consistent hash chain; 3) if an attacker compromises one node of the system, she can pretend to be any client and generate fake requests. She can modify r in REQ_Q , such that $r = n_i$ and n_i is the node she controls, so it gets always selected as execution node. These issues are related to the verification and logging process of requests, shown in Figure 7.1a.

The first problem breaks the *Trustworthiness* requirement discussed in Section 7.5. To solve this problem, untrusted clients should broadcast their requests **REQ** to all nodes, instead of forwarding them only to the primary node. Then, the primary can continue the PBFT protocol as usual. Non-compromised nodes accept **REQ** if they have received two copies of it, one from the primary node and one from the client. Thus, if a compromised primary node forwards fake requests, they are rejected by all non-compromised nodes.

The second problem breaks the *Tamper-resistance* requirement. This problem is solved by including the *checkpoints* generated by PBFT in the log. We briefly discussed checkpoints in Section 7.3. A more in depth description can be found in [144]. A checkpoint is included in a log entry along a hashed value that links it to the previous request **REQ**. Checkpoints are generated periodically after z requests. An attacker cannot forge a checkpoint because she cannot forge the signature of $2f+1$ nodes. She can forge at most $z-1$ requests without being detected. However, the attack is detected as soon as the z -th request is stored and the checkpoint is created.

The third problem affects the random number r in **REQ**_Q. Manipulating this value can guarantee plaintext data access to an attacker. We solve this problem by removing r from a query requests, and forcing nodes to agree on a random number *without* involving the client. After accepting **REQ**, all nodes run a distributed protocol to randomly select an execution node. The details of this protocol are discussed below. Hence, even if an attacker controls a node and can impersonate an untrusted client, she cannot deterministically analyze data in plaintext on machines she controls. However, there is no way to prevent a compromised client to access the result of the computation.

A collusion of an *untrusted* client and a compromised node, can lead an attacker to attempt accessing any other users data, and the lack of digital signatures impede the nodes to determine whether those requests are originated by the owner of the data or not. However, the attacker has no control over the random selection of the execution node, because, as discussed below, the attacker must control all nodes (i.e., n nodes) to manipulate the execution node selection and in our threat model the attacker can compromise at most f nodes (see Section 7.4). Since the execution node is the only node that can access the plaintext data, the attacker has no guaranteed data access.

The attacker can send (or replay) multiple requests to increase her chances to have her compromised node selected as execution node. This misuse can be limited by enforcing security policies such as limiting the number of requests on certain data.

Lastly, although the attacker can access more data by compromising clients, the log is still able to *reliably* determine the data leakage, and it is not affected by the collusion.

Collaborative Selection of the Execution Node

We now describe a protocol that allows distributed nodes to agree on a specific random number, which is then used to select the execution node for *query* requests.

The selection is guaranteed to be random as long as the attacker cannot compromise more than f machines. This protocol is used in two occasions: in presence of untrusted nodes, or in case the execution node of a request is not available and a new node should be selected.

Each node n_i generates a value x_i uniformly distributed at random. x_i is the value used to select the execution node. Each node computes a commitment $c_i \leftarrow \text{Commit}(x_i)$. In our specific scenario $\text{Commit}(x_i) = H(R_i || x_i)$ for a cryptographic hash function H and some random value R_i . Each node sends c_i to the primary node. The primary node selects a set $\mathcal{F} = \{c_1, \dots, c_{f+1}\}$ of commitments, and forwards it to each node. Each node n_i check if its commitment c_i is in $\mathcal{F}$. If so, it sends x_i and R_i to the other nodes. Each node verifies the commitments $\text{Reveal}(x_i, c_i) = 1, \forall c_i \in \mathcal{F}$. If they are all correctly verified, the execution node is computed as $r = \sum_{i \in \mathcal{F}} x_i \bmod n - 1$.

In case the primary node does not forward $\mathcal{F}$, it is considered compromised and all nodes start an instance of the *change-view* protocol (cf. Byzantine Fault Tolerant Consensus Protocol in Section 7.3) to select a new node as primary node. Nodes selected to reveal their committed value can also be compromised and may not reveal x_i . For this reason, there exists a timeout that limits the waiting time for revealing the commitments. In case the timeout is exceeded, the primary node sends another set $\mathcal{F}'$ of commitment values, that differs from $\mathcal{F}$ only for those commitments that were not revealed within the timeout.

The Benefit of using a Commitment Scheme. Suppose every node n_i in the system broadcast a random value x_i to all other nodes, and the sum of all those values is the execution node. A compromised node can wait until all other $n - 1$ nodes broadcasted their value. At this point, the attacker can compute the sum S of these values and she can choose a value x_i such that $x_i + S = n_i$, where n_i is a compromised node. This attack cannot be detected given that the network is asynchronous.

The commitment scheme thwarts a compromised node from learning what other nodes have committed to, and it impedes an attacker to change her value after she has committed to it. Therefore, the aforementioned attack is mitigated by the commitment scheme. Our approach also resists the adversary model described in Section 7.4 because, even though an attacker controls f nodes including the primary node, she has to select a set of $f + 1$ commitments. Therefore, there is always at least one value that is uniformly distributed at random, which guarantees r to be random as well.

7.6.2 Determining Data Leakage using the Distributed Log

As discussed in Section 7.4, we assume the company knows the set $\mathcal{M}$ of compromised machines, and the period of compromise $(T_{begin} - T_{end})$. This information is often the result of a breach investigation. Typically, IDS are deployed in order to detect whether a machine is compromised, as we discussed previous chapters.

Algorithm 2 describes how to determine the data leakage from a log V . First, it verifies the integrity and authenticity of the content of each log entry e_i (Line 4).

Figure 7.2: Overview of the collaborative protocol to select the execution node. In the first phase each node i sends their commitment c_i to the primary node (i.e., n_0 in this case), which then selects $f + 1$ commitments (i.e., c_2 and c_0 in this example). The selection is then forwarded to all nodes. The selected nodes n_2 and n_0 reveal their values sending x_i and y_i . All nodes verify that x_2 and x_0 match with c_2 and c_0 , respectively. Finally, they sum the committed values and compute the module $n - 1$ to retrieve r , which represents the execution node.

Second, it verifies the integrity of the hash chain (Line 5). If any of these two steps fails, V is considered tampered and discarded. Otherwise, Algorithm 1 checks if e_i describes a computation over a compromised node within the period of compromise in Line 6. If so, the set $\mathcal{D}$ of items is added to a list of leaked items. This process is repeated recursively on each entry in the log. Lastly, Algorithm 1 returns the list of leaked items. The data leakage is determined by identifying all data items, within the period of compromise, accessed by compromised nodes.

For scenarios where clients are trusted, it is also possible to determine the data leaked to a specific compromised client. This can happen in case of malicious *insiders*, which is also a severe threat scenario in data breaches.

7.6.3 Analysis

In this section we discuss how the distributed log satisfies the security requirements discussed in Section 7.5.

Tamper-resistance. Each log entry $e_i = (i, \langle \text{REQ}, \sigma_{\text{REQ}} \rangle, L_i)$ contains a signature σ_{REQ} and a hashed value L_i that links the i th entry with the $(i - 1)$ th. An attacker is not able to break cryptographic primitives, therefore she cannot modify REQ and generate a valid signature σ_{REQ} . Any addition or removal of entries in the log is detected by the hash chain values $L_0, \dots, L_{i-1}, L_i$, where L_0 is a root value, because an attacker cannot generate a new REQ'_i such that $\mathcal{H}(L_{i-1} \parallel \text{REQ}'_i) = \mathcal{H}(L_{i-1} \parallel \text{REQ}_i)$. Lastly, an attacker cannot *forge* the history of the log (e.g., providing a log with a consistent hash chain but containing *fake* entries), because each entry e_i contains the signature of a client and the attacker cannot forge it. The only entry that does not require a client signature is the upload of an execution node. However, this operation is considered valid only if a client request containing a `write(ID)`

Algorithm 2 Determination of Data Leakage

Require: $\mathcal{M}$ is the set of malicious machines, where $|\mathcal{M}| \leq f$; T_{begin} is the time when the breach started; T_{end} is the time when the breach finished; a log V from a non-compromised machine. Let us consider j as the client who signed the request REQ .

```

1: procedure DATA LEAKAGE( $V, \mathcal{M}, T_{begin}, T_{end}$ )
2:   leaked_items  $\leftarrow []$ 
3:   for all  $(i, \langle \text{REQ}, \sigma_{\text{REQ}} \rangle, L_i)$  in  $V$  do                                 $\triangleright$  i.e., for each log entry
4:     if Verify( $pk_j, \text{REQ}, \sigma_{\text{REQ}}$ ) = 1 then
5:       if  $L_i = \mathcal{H}(L_{i-1} || \text{REQ})$  then
6:         if  $\text{REQ}.r \in \mathcal{M}$  and  $T_{begin} \leq \text{REQ}.ts \leq T_{end}$  then
7:           leaked_items.add( $\text{REQ}.\mathcal{D}$ )
8:         end if
9:       else
10:        return  $\perp$ 
11:      end if
12:    else
13:      return  $\perp$ 
14:    end if
15:  end for
16:  return leaked_items
17: end procedure

```

has been accepted, which cannot be forged by the attacker. This attack is also detectable in case of *untrusted* clients as discussed in Section 7.6.1.

Availability. Our solution is based on PBFT and it survives Byzantine failures when the total number of nodes is $3f + 1$. Therefore, even if an attacker compromises f machines (i.e., worst-case scenario) and additionally f other machines fail (e.g., hardware failure), it is still possible to access $f + 1$ *untampered* logs from the remaining *non-faulty* machines.

Trustworthiness. An attacker wants to remotely forge the log of non-compromised nodes by convincing them to append requests that have never been requested by clients. The attacker can achieve her goal only by being the *primary node* and if she fulfills one of the following requirements: 1) forge a client's private key or 2) control $2f + 1$ nodes. In the former case, an attacker cannot forge a client signature, because she has no access to her private key sk and she cannot break cryptographic primitives. In the latter case, the attacker can compromise at most f nodes, which is not enough to convince all non-compromised nodes that fake events really happened (the *safety* property of PBFT).

Completeness. The distributed log is complete if all the accesses to plaintext data are recorded by *non-compromised nodes*. Every time a request is processed,

and data is reconstructed in plaintext to an execution node, all *non-compromised* nodes log the event. If an attacker wants to access data in plaintext without being recorded, she needs to collect enough shares s_i in order to reconstruct the data. She can access at most f shares for each item in the system. However, $f < f + 1$, where $f + 1 = t$, so $\text{Combine}(\{s_i\}_{i=1}^f) \rightarrow \perp$. Therefore, she cannot access any data item without any help from at least a non-compromised node, which always logs when shares are sent to other nodes, and it does that only upon valid requests.

Communication and Computation Complexity

Our approach describes two types of requests: the upload of data and the computation over data. Both request types use PBFT to verify and to log requests, so their communication complexity is $\mathcal{O}(n^2)$ per request. This is caused by the nodes broadcasting the request in the *prepare* and *commit* phase, as shown in Figure 7.1a. The remaining phases for both query types, shown in Figure 7.1b and Figure 7.1c, are responsible for transmitting shares over the network and their communication complexity is $\mathcal{O}(n)$. The computational complexity to split a data item in t -out-of- n shares is $\mathcal{O}((n - t)t)$ [147]. In our case, we have $t = f + 1$. The communication complexity of the distributed protocol used to select the execution node is $\mathcal{O}((f + 1)n)$, because $f + 1$ (or t) nodes broadcast their values to every node once their commitment has been selected.

The above complexity analysis only considers the additional overhead imposed by our approach when used on top of an existing service. The plaintext operations of the specific underlying service that are performed by the execution node are excluded in our analysis.

Quality of Leakage Determination

Recall that we have shown in Section 7.6.3 that our approach *reliably determines* data leakage, which was the goal of this work. So, in principle, we could stop here as we have achieved our goal. However, it does *not prevent* leakage because data is processed in the clear by the execution node (which might be compromised). Looking at our construction, we notice that it can still happen that *all* data leaks or that *nothing* leaks to an attacker. For instance, let us assume that an attacker manages to access all data in our system. In this case, our mechanism accurately and reliably determines the data leakage as 100%. Therefore, in order to evaluate where our approach lies between “*all* data has leaked” as opposed to “*nothing* has leaked”, we would like to evaluate the expected data leakage in our system. More precisely, we want to evaluate the *quality of leakage determination*, which we define as the expected leakage in the system in the *worst-case* attack scenario. Recall that, following our adversary model (as defined in Section 7.4), our worst-case attack scenario is met when an attacker has fully compromised exactly f (out of n) nodes.

While the worst-case attack scenario is fixed in our adversary model, the expected leakage clearly depends on the concrete application setting we deploy our system in. Due to this dependence, evaluating the quality of leakage determination

is not feasible in general terms. However, there is a specific category of applications for which the quality of leakage determination can be accurately evaluated without looking at individual applications within this category. We do our analysis of the quality of leakage determination for this category of applications which can be described by the following representative setting.

Consider a setting in which users are associated with a specific set of data items. The users can run computations only on their own dataset, and the computations are independent events. A very important example is a client-server authentication as discussed in Section 7.6.4. Moreover, let us assume an attacker controls f nodes during the whole system lifetime, which is the worst-case scenario of our approach. For this setting, we can derive a formula that allows us to compute the expected data leakage. Specifically, we can compute the number of distinct clients/users that would have their data leaked. We assume each user i requests m_i computations on his data during the whole system lifetime. A user's dataset is considered leaked if at least one computation out of m_i is reconstructed on a compromised node. Since the execution node is selected uniformly at random, a user's dataset is leaked with probability $p = 1 - (1 - \frac{f}{n})^{m_i}$, where n is the total number of nodes and f is the number of compromised nodes. The leakage of a user's dataset is an event (X) with two possible outcomes: either it is leaked (1) or it is not leaked (0). The expected data leakage for a user's dataset matches with the probability of its dataset to be leaked. So, if M is the entire population of users in the system, the expected data leakage is:

$$E[X] = \sum_{i=1}^M 1 - (1 - \frac{f}{n})^{m_i} \quad (7.1)$$

Overall, this shows that our approach is, on average, better than just saying that all or nothing has leaked. In Section 7.6.4, we evaluate a specific real-world application scenario using equation (7.1).

7.6.4 Application: Client-Server Authentication

In this section we evaluate the efficiency and the quality of our leakage determination approach in the setting of authentication. We have chosen this type of application because credentials are often a major target of data breaches. A specific example is the case of LinkedIn that, despite knowing they have been compromised in 2012, they were not able to determine the data leakage. Their estimate was around 7 million records. Only in 2016, when the attacker sold the data online, they realized that the leakage affected around 160 million records [24]. In biometric authentication, the data is even more sensitive because it cannot be changed, and determining its leakage is essential for customers to prevent identity theft.

We evaluate password-based, biometric-based and PUF-based (Physical Unclonable Function) authentication. These authentication mechanisms represent the three authentication categories: something we know, something we are and something we have, respectively. All three types of authentication involve a registration

phase, where users or devices register their authentication token. In the specific case of biometrics, the token is a biometric template of a physiological or behavioral user characteristics, whereas in the PUF case, the token is a challenge-response pair [150].

Efficiency of Our Approach

Our distributed log is based on two main building blocks, AONT-RS [147] and PBFT [144], whose performance and functionalities have been already verified in literature and are adopted in practice [147], [151]. We believe *an implementation would not improve the understanding of our solution*. However, we estimate the efficiency of our approach to show its feasibility. Our analysis considers the case of normal operations, where there are no compromised nodes. We consider nodes to run operations in sequence and we do not consider protocol optimizations, especially for PBFT. We are aware that this estimate does not reflect the real efficiency. However, we believe the estimate suffices to give an indication on the feasibility of integrating our approach.

We estimate the efficiency our approach in a biometric-based authentication such as iris recognition [152]. Let us consider a scenario where iris scanners (on company premise) are trusted clients, which have a RSA 2048-bit key pair and communicate through a 100Mbps network link with n nodes. Nodes are also connected with a 100Mbps network link. A biometric-based authentication, with our approach, would work as follows: first the client uploads the iris feature vector, then it requests a comparison between the recently uploaded vector and the one uploaded at registration time. If the feature vector would not be uploaded in a separate step, but sent in the clear within the query request, it would automatically leak to any node. Thus, each phase of the PBFT consensus mechanism (Figure 7.1a) is run twice, while phases of data upload (Figure 7.1b) and computation (Figure 7.1c) are run only once. We calculate the estimate by summing the time to transfer each message from one party to another, and the time to validate the signatures. We consider all operations to be in sequence, for instance if four nodes have to validate a signature, we sum four times the validation time.

Table 7.1 shows that PBFT is the bottleneck of our approach. When there are four nodes, during one instance of PBFT 29 messages are transmitted over the network. Messages are upload or query requests. Since an authentication attempt needs two requests, the number doubles³. Due to the communication complexity of $\mathcal{O}(n^2)$, the performance of PBFT degrades quadratically as the number of nodes increases. The upload and computation phases, which send shares⁴ over the network, do not influence the performance much and their overhead linearly increases with the number of nodes. The low overhead is a consequence of the small number

³For this use case, we assumed the upload requests to be 276 bytes, and the query requests to be 288 bytes. In both requests, 256 bytes represents the digital signature. The different size between the two request is due to the larger amount of arguments in the query requests.

⁴The data that we secret share is composed by 256 bytes of the iris feature vector, plus additional 64 bytes needed for AONT-RS. For this use case shares have the following size: 160 bytes (4 nodes), 106 bytes (7 nodes), 80 bytes (16 nodes), because the size of shares is dependent to the threshold value t , such that $\frac{\text{data}}{t}$ [147].

Nodes	PBFT (2x)	Upload	Computation	Total time	Op/s
n=4, f=1	1.5 ms	0.1 ms	0.1 ms	1.7 ms	588
n=7, f=2	3.7 ms	0.2 ms	0.2 ms	4.1 ms	243
n=16, f=3	21.2 ms	0.5 ms	0.4 ms	22.1 ms	45

Table 7.1: Estimates of the additional overhead (per authentication) caused by our approach for an iris recognition application. The column “PBFT” represents the protocol steps in Figure 7.1a, while ‘Upload’ and ‘Computation’ represent the steps depicted in Figure 7.1b and Figure 7.1c, respectively. Op/s is an estimate of the number of authentication operations that can be performed. Our estimates are computed for different number of nodes.

of messages, the small size of the reconstructed item (i.e., an iris feature vector of 256 bytes [152]), and the encoding performance of AONT-RS [147].

Considering the performance values shown in Table 7.1, we believe this approach is efficient enough to be integrated in existing biometric-based authentication applications. Moreover, our estimation does not take into account PBFT optimizations, the parallelization of operations, and the fact that only $f + 1$ shares are needed to perform computations and not n . This approach is also practical in password-based and PUF-based solutions given the fact that they run the same number of requests and transmit smaller shares.

The Quality of Leakage Determination in a Real-World Application

In this section we show by using a real-world example that our approach is not an *all-or-nothing* approach. Undoubtedly, a *nothing* approach is the ideal scenario. However, this is not (yet) achievable in practice, since required cryptographic techniques (e.g., FHE or MPC) are not efficient enough in most practical applications. The remaining option is to consider *all* data to be leaked, which is the other extreme case.

Our approach explores a middle-ground between the two approaches, where the company might still suffer a leakage of data but it can *precisely show* what has been leaked. We show that our approach is not *all-or-nothing* by determining the data leakage using equation (7.1) in the setting of a real-world application. The same equation can be used by operators to understand the behavior of their application under the proposed distributed systems and understand its possible benefits a priori.

Real-world Example. We consider a currently centralized real-world application such as the internal ePrint web service of the entire faculty of Computer Science at an international university. We evaluate the quality of leakage determination in case this service would be integrated with our solution. We analyze the authentication log, which contain over 10 years of login information. We consider the time of compromise to be 256 days, which, according to a recent study on data breaches [25], has been identified as the average time to identify a data breach.

(a) Quality of leakage determination with $n = 4$ and $f = 1$ in 5 years, for a compromise time of 256 days [25].

(b) Quality of leakage determination in 2013 with number of nodes: $n = 4, 7, 10, 16$.

Figure 7.3: Representation of our quality of leakage determination in the setting of the ePrint service of CS faculty.

We apply the model discussed in Section 7.6.3 to our password-based authentication dataset, where we consider the user’s dataset as the user’s password.⁵ A user computation is a login, in which the user requests the system to compare whether the provided password matches the stored password or not. We extracted the amount of logins (m_i) for each of the 2330 users, and we computed the overall expected data leakage with equation (7.1).

Figure 7.3a represents the average quality of leakage determination in case an attacker had compromised f nodes (i.e., $n = 4$ and $f = 1$) for a period of 9 months (i.e., 256 days), which is the average duration of a data breach before being detected [25]. We computed this value over the last 5 years data and it is possible to see how the leakage of the entire user base is determined to be lower than 9%. If we consider only the set of users that logged in at least once, the leakage is at most 68%. In both cases, it is clear that our solution is not an *all-or-nothing* approach.

We have also analyzed the *unrealistic* worst-case scenario where we consider an attacker to compromise f nodes for the entire 10 years. In this case, 32,5% of the entire dataset is leaked, while if we consider only users who logged in at least once, the leakage is 86,4%. Therefore, even in the unrealistic setting of 10 years of compromise, our solution proves to not be all-or-nothing.

Figure 7.3b depicts the quality of leakage determination from the log data of 2013 considering a different numbers of nodes. As expected, a lower number of compromised nodes results in less data leakage. Finally, it is worth noting that the determination of leakage in the worst-case scenario, where f nodes are controlled by the attacker, slightly increases with the number of nodes. This is due to the different ratio of $\frac{f}{n}$ (e.g., $\frac{1}{4} = 0.25$, $\frac{5}{16} = 0.31$).

This evaluation holds also for biometric-based and PUF-based authentication scenarios, because also in these cases one token is reconstructed for each authentication.

Finally, in password-based and PUF-based cases, it is possible to update of authentication tokens after a fixed period of time, such that leaked data becomes useless. This computation can help in defining such policy. For instance, an operator can learn from the computation the subset of users whose passwords are more prone to be leaked in case of compromise. With this very specific information, the operator can define a policy to enforce a change of password for this specific set of users, so that the leaked data can become useless for the attacker.

7.7 Discussion

Other Applications. Our approach is not limited to authentication applications. *Any* service running computations over data, can be integrated with our approach. It is important to notice that our system moves the data over the network every time a computation is requested, therefore applications that are often

⁵The term password can be substituted with other authentication tokens such as fingerprints for a biometric-based scenario, or the one-time challenge-response pair for the PUF-based scenario.

computing on large data items (e.g., hundreds of megabytes) can suffer severe performance issues. A possible solution for those application could be to encrypt the data with a symmetric encryption scheme and secret share the secret key. The storage requirements would increase (e.g., an encrypted copy of each item on each node), but the network performance would significantly improve.

Practicality. Compromising a central server containing sensitive data is often a complicated task and it requires time, because the infrastructure is often well protected. Nonetheless, central servers are still being compromised. A distributed solution, protected with the same degree of attention, makes it harder for an attacker to achieve her goal, because compromising one machine is not enough. This preserves the functionality of the system for a longer period (i.e., determination of leakage), and it exposes the attacker to a higher risk of detection because she has to try additional attacks. This holds if the system is heterogeneous, which implies that the system possibly runs different OSs, uses different passwords, software implementation, etc.

The major limitations of our system is inherent to the quadratic overhead of PBFT. Scaling to a huge number of nodes would make it incredibly difficult for an attacker to compromise the entire system, making the system very secure. Unfortunately, despite the fact that there are more efficient Byzantine fault tolerant consensus protocols than PBFT, there are still scalability issues. The new approaches to consensus used in cryptocurrencies, such as Proof-of-Work [26] and Proof-of-Stake [27], might be an alternative that can solve the problem of scalability. However, these techniques introduce new problems (e.g., consumption of resources) that can limit the practicality of our approach. Nonetheless, investigating these alternatives is a topic for future research.

Finally, considering the performance estimate of our approach, we believe our system is practical for a small number of nodes. We also consider it practical when it comes to keeping the system heterogeneous for few machines.

7.8 Related Work

Most of the work on the topic of data leakage focuses on prevention. One of the most obvious approaches for leakage prevention is cryptography [50], [51]. Data Leakage Prevention (DLP) [138]–[140] is another category of techniques. All these approaches are not capable of determining the data leakage in case the prevention fails (in case they prevent it, the leakage is nothing).

Other research areas such as secure storage and data provenance can help achieving data leakage determination. Strunk et al. [153] proposed a self-securing storage that tracks and logs all the modifications to the data stored in the system, unless the attacker can compromise the host operating system. However, to determine the data leakage is important to log the access to all files and this information is not logged. If an attacker opens a file and reads its content, the operation is not logged as evidence. Other secure versioning solutions such as [154] have the same problem, and it is not known how much this would impact the system performance,

especially on platforms performing a lot of computations.

Provenance-aware systems are promising solutions for data leakage determination because they aim to collect, store and manage the history of every single data object in the system. The term *provenance* refers to the history of ownership of an object. The vast majority of works on data provenance [141], [142], [155]–[157] are not designed to resist malicious adversaries, which makes them unusable in case of system compromise (one of our main assumptions). Pohly et al. [143] proposed Hi-Fi, a provenance-aware system that collects provenance through the Linux Security Module, which mediate any data manipulating operation on kernel objects. Hi-Fi protects the integrity provenance data against adversaries in user-space by using additional hardware such as read-only storage. Although this allows for the investigation or detection of a compromised system, it does not help the determination of data leakage because an attacker can append new fake entries to the provenance log. Additionally, protecting from user-space attacker is a weaker model. Zhou et al. [158] introduced secure network provenance (SNP), a technique that provides forensic capabilities to an untrusted networked system, where a subset of nodes can be fully compromised. SNP allows an administrator to identify faulty nodes and to explain any unexpected state change on any node. SNP focuses more on tracking faulty nodes and providing evidence for it, rather than determining what data has been accessed and possibly leaked. Bates et al. [52] have presented the only solution capable of collecting data provenance against a strong adversary, which could be used to determine what data has been leaked. The Linux Provenance Module (LPM) allows a secure provenance collection of processes, network activities and the kernel itself. Through the analysis of the provenance graph, it would be possible to determine the data leakage. However, the security of the whole system mainly relies on trusted hardware, and it is known that the storage and analysis of months (or years) of collected provenance data is still a severe practical problem.

7.9 Conclusions

We presented an approach to determine data leakage using a reliable distributed log. Our proposed log works even in the presence of powerful attackers that are capable of taking complete control of a limited number of machines. In contrast to previous work, we do not require any trusted hardware, despite the strong threat model. We have demonstrated the quality in leakage determination of our solution on real-world data; showing that it is not an *all-or-nothing* approach. Given that we use well-known building blocks that have been demonstrated to work in practice [147], [151] and based on the values obtained from a worst-case estimation of our protocol, we believe our solution can be integrated with existing (authentication) services.

This chapter addresses **RQ3**, and shows that it is possible to evaluate the impact of a data breach. Although the approach is practical in specific settings (e.g., few nodes), it is not legacy compliant, meaning that it is not possible to easily adjust existing logging methods with our system. In the next chapter, we discuss

the conclusions of this thesis, summarizing our contributions, the limitations of our proposed systems and the future works.

Chapter 8

Concluding Remarks

We conclude this thesis by discussing the contributions of our work in relation with the research questions discussed in the introduction. Furthermore, we discuss some remaining open problems.

8.1 Contributions

Today almost any business depends on digital technologies and data is becoming a fundamental asset. At the same time, data can damage a business if it is not properly protected. The disclosure, or loss, of customer information, company secrets and employee information can severely harm the reputation of a company and its finances. This type of security incident is called data breach. Historically most of the security research has investigated how to prevent security incidents, and less attention has been paid towards their detection and mitigation. As a consequence, companies can rely on fewer technologies to address the detection and mitigation of data breaches. The difficulties in discovering and mitigating a data breach are also highlighted by recent research [10]. Thus, in this thesis we decided to provide technical improvements for both these aspects, specifically, the detection of data exfiltration and the evaluation of its impact (i.e., determine what data has leaked).

The main research question of this thesis was:

How to detect a data breach and how to analyze its impact?

To address this research question, we have proposed, implemented and evaluated different systems. The contribution of this thesis is threefold:

1. We proposed *passive application fingerprinting* for anomaly-based detection, a technique to detect data exfiltration over the network. With this approach we address **RQ1**:

How to detect data exfiltration without predefined threat knowledge?

We first introduced the concept of passive application fingerprinting with DECANter, in Chapter 3. We have implemented and compared DECANter

with the current state-of-the-art anomaly-based detection solution for covert channels over HTTP, and we have shown that passive application fingerprinting performs significantly better. Our approach performs better because it relies on fine-grained models (i.e., application fingerprints), rather than coarse grained models (i.e., a single model for the whole HTTP traffic). Additionally, passive application fingerprinting is effective because, currently, malware does not adapt its traffic to its victim.

Then, we refined the fingerprinting technique introducing HEADPRINT in Chapter 4. HEADPRINT automatically infers the distinguishing characteristics of application traffic, instead of relying on a predefined feature set. Compared with DECANter, HEADPRINT is more accurate in correctly identifying application traffic. This improvement allows HEADPRINT to generate fewer false alarms in the setting of anomaly detection, thereby making passive application fingerprinting more practical. The implementations and (part of) the datasets used in these research works are publicly available.

2. The fact that malware is currently not adapting (to the best of our knowledge) to its victim’s traffic, does not mean that it will not do so in the near future. We have investigated the topic of victim-aware adaptive (VAA) attackers by addressing **RQ2**:

How can we detect victim-aware adaptive (VAA) data exfiltration?

First, we have evaluated how existing HTTP covert channel detection techniques would perform against *VAA covert channels*. We achieved this by introducing CHAMELEON in Chapter 5, a toolchain to generate synthetic datasets representing VAA data exfiltration attacks. We used these datasets to evaluate the performance of existing detection solutions, and we have shown that they are not effective in detecting adaptive data exfiltration. Then, in Chapter 6, we proposed HONEYTRAFFIC, a deception-based detection technique that is able to effectively detect specific cases of VAA data exfiltration attacks (i.e., those implemented by CHAMELEON) with a negligible network overhead and a low false positive rate.

3. Finally, we investigated how to evaluate the impact of data exfiltration by addressing **RQ3**:

How to reliably determine what data was exfiltrated despite the host being compromised?

In this work we proposed a distributed secure log to reliably determine what data is exfiltrated after a data breach. The system, discussed in Chapter 7, combines existing cryptographic primitives and Byzantine consensus protocols to generate a distributed system that collaboratively log and provide data access. We have evaluated our system in a realistic scenario, and we have shown it can reliably identify an upper bound of what data may have leaked. In contrast with existing work, our system is not an all-or-nothing

approach and does not rely on trusted hardware assumptions. Lastly, we discussed how it can be integrated with some existing services.

The contributions of our work to the security community are not limited to the presentation of detection and impact evaluation techniques. During our research, we have also developed and collected artifacts to perform our analyses, which have been partly publicly released. The release of our artifacts is not only for research reproducibility, but also to foster future research in data exfiltration detection, which, so far, has mainly focused on non-adversarial use-cases.

In the context of the security lifecycle used by organizations to protect their infrastructure, this thesis reduces the technological gap for detecting and mitigating data breaches.

Passive application fingerprinting (PAF) is a new technique that companies can integrate in their arsenal of defensive tools in order to detect more quickly ongoing data breaches. PAF helps to partially solve the problem of data exfiltration detection over HTTP by providing better detection performance and by being more practical than the current state-of-the-art solutions. We are aware that it is impossible to completely solve the problem of data exfiltration detection, especially for low bandwidth channels. However, we believe that it is important to propose new techniques to maximize the chances of detecting such attacks. Moreover, an interesting property of PAF is that it detects anomalous outbound traffic, therefore it can potentially identify threats other than data exfiltration, such as command and control channels and other types of malicious communication. Most importantly, the idea behind PAF is independent from the protocol itself, and it leaves some important open problems related to its applicability in other settings. We discuss in the next section.

At the same time, our work gives insights regarding the inadequacy of traditional network-based detection tools against sophisticated attacks. VAA covert channels break the assumption of detection tools that rely on the passive analysis of network traffic. After this study, we have realized that passive network monitoring solutions are inherently limited against such threats and that there is a big technical gap to fulfill. In our opinion, the combination of host and network information can improve the detection capabilities. With our deception-based solution, we have started to reduce this technological gap, because we propose a detection solution effective against a specific of VAA covert channels, namely Full VAA covert channels, where the attacker maximizes the similarity of her messages to the victim's traffic by copying as much information as possible within her network messages. CHAMELEON showed how this type of VAA covert channel cannot be handled by existing detection solutions. However, the problem of effectively detecting VAA covert channels is not solved, because our approach does not detect other types of VAA covert channels, such as Partial VAA, where the attacker partially mimics the victim's traffic and partially generates message content by herself. Finding a solution that can effectively tackle all types of VAA covert channels remains a hard problem.

Lastly, our distributed logging system can help companies to better mitigate data breaches. Under certain assumptions, our system solves the problem of deter-

mining leakage in adversarial setting. The practical limitations of our solution are mainly inherited from the consensus protocols. Nonetheless, the core idea behind our solution is independent from the underlying consensus protocol. Thus, in case more efficient protocols will be proposed in the future, our solution will likely be more efficient as well. Nonetheless, it remains still an open problem how to exactly identify what data has been leaked. Our approach only provides an upper bound.

8.2 Open Problems

In this section we discuss some of the open problems that relate to our work. Since passive application fingerprinting is a new technique proposed for anomaly detection, there are still several questions that should be answered. In this section we discuss three important aspects to investigate about this technique: its applicability to other protocols, its applicability to mobile traffic and how fingerprints can be securely updated in adversarial settings. Moreover, HONEYTRAFFIC is a detection technique that works on some cases of VAA covert channel, namely Full VAA (e.g., CHAMELEON), however it does not cover other cases such as Partial VAA. Thus, the investigation of new detection techniques remains a relevant issue. Finally, we discuss the problem of how to exactly determine the data leakage in adversarial, which is still an open problem in literature.

Generalization to other protocols. An important open problem related to our work is the investigation on how passive application fingerprinting can be applied to *encrypted protocols*, such as TLS. A solution to this problem would make the whole approach more practical, because it could be applied to most of the networks without relying on any expensive proxy. One way to tackle this problem is to identify distinguishing patterns in the TLS Handshake protocol, which is in plaintext. JA3¹ fingerprints are a good starting point, because they show that it is possible to fingerprint TLS clients from Handshake protocol information. However, JA3 fingerprints are mainly being used as signatures for TLS-based malware. It is unclear how well JA3 can track the connections of a single application, and how well it reacts to software updates. These characteristics are important for “whitelisting” approaches such as passive application fingerprinting. Perhaps, one way to start investigating this issue would be to study and compare JA3 with HEADPRINT. The method used in HEADPRINT is protocol agnostic, so, in principle, HEADPRINT may be able to identify distinguishing characteristics in the Handshake protocol. After a thorough study, it could be possible to find a solution in between that can correctly fingerprint applications over the encrypted traffic. A challenge for this research is also the collection of a large labeled dataset with traffic from different applications.

Another important problem is the generalization of passive application fingerprinting to other protocols. An attacker may use different channels for the communication, which are not necessarily covered by our solution. Among others, known protocols used for malicious activities are SMB, SMTP, and SSH. Therefore, it is

¹<https://github.com/salesforce/ja3>

important to extend the coverage also to other protocols. Ideally, a single solution can be applied to model the usage of any protocol. To achieve this we see two main challenges. First, the solution should work only on the byte sequences observed in the messages (i.e., assuming no parser is available for the specific application protocol). Solving this challenge would increase the processing speed of network messages. This is important in case all the network protocols would be monitored. Second, the solution should automatically infer the invariants and distinguishing characteristics of the usage of a network protocol. Solving this challenge would allow to apply the same method to multiple protocols.

Mobile Traffic. Our work mainly focused on enterprise environments, with a special focus on workstations and server traffic, which often contains the most sensitive data. On the other hand, our solution is not suitable for monitoring mobile devices, which install and run several applications over a short period of time. In this setting, triggering alerts when new applications are installed is not a viable option, because it would likely cause a surge of FPs. Thus, the system should be able to identify new applications, to automatically generate fingerprints for them, and to alert the operator only when there are signs of malicious behavior. The main challenge lies in the generation of new application fingerprints, because in mobile traffic it can occur that many different applications are talking at the same time.

Secure Fingerprint Update. Ideally, a passive application fingerprinting system would be able to automatically update the fingerprints over time. Software gets updated and new functionalities are added, thus, the network traffic may change accordingly. In our work, we purposely do not try to automatically and systematically update the content of the fingerprints over time, other than in rare situations (see fingerprints update in Section 3.4.4). The reason for this choice is that we are in an adversarial setting, and the attacker may abuse the updating mechanism to evade the detection. We consider the investigation of secure update procedures a relevant open problem, in order to achieve an *evolving* system without human intervention.

Detecting Victim-Aware Adaptive Attackers In literature there are no effective detection mechanisms that can effectively identify victim-aware attackers. Our proposed solution, HONEYTRAFFIC, is a solution to detect attackers as described by CHAMELEON (i.e., *full victim-aware adaptive attackers*), but we *do not know* whether HONEYTRAFFIC is effective against other types of VAA attackers, such as *partial victim-aware adaptive attackers* (P-VAA). We briefly discussed this topic in Section 6.6.1. A more in depth study on the capabilities of P-VAA attackers may help to answer this question. Moreover, it remains unclear how difficult it is for a VAA attacker to hide from a detection system analyzing both host and network information. The effective detection of VAA attackers remains a challenging open problem.

Determining the Exact Data Leakage. The distributed system we have proposed for the impact evaluation of a data breach provides an upper bound of what data may have leaked. However, it cannot tell exactly what has leaked. The precise identification of leaked data items would be ideal for data breaches investigations, however it is difficult to achieve in adversarial settings. A potential approach to achieve this is to monitor how data is manipulated by the node. For instance, by verifying if the performed operation on the data is the one requested by the client. However, the threat model should be also revisited, because we currently assume that the attacker can fully compromise the host (i.e., including a tool monitoring the host). An alternative assumption could be to consider a virtual environment, where the attacker is assumed not to compromise the hypervisor, which can be used as a monitor.

Author's Publications

- [1] R. Bortolameotti, T. van Ede, M. Caselli, M. H. Everts, P. H. Hartel, R. Hofstede, W. Jonker, and A. Peter, "Decanter: Detection of anomalous outbound HTTP traffic by passive application fingerprinting," in *Proceedings of the 33rd Annual Computer Security Applications Conference, ACSAC 2017, Orlando, FL, USA, December 4-8, 2017*, 2017, pp. 373–386.
- [2] R. Bortolameotti, T. van Ede, A. Continella, T. Hupperich, R. Rafati, M. H. Everts, W. Jonker, P. Hartel, and A. Peter, "Headprint: Detecting anomalous communications through header-based application fingerprinting," *Manuscript in submission, version 15th September 2019*, 2019.
- [3] R. Bortolameotti, T. van Ede, A. Continella, M. H. Everts, W. Jonker, P. Hartel, and A. Peter, "Victim-aware adaptive covert channels," *Security and Privacy in Communication Networks - 15th International Conference, SecureComm 2019, Orlando 2019, USA, October 23-25, 2019*, 2019.
- [4] R. Bortolameotti, A. Peter, M. H. Everts, W. Jonker, and P. H. Hartel, "Reliably determining data leakage in the presence of strong attackers," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, 2016, pp. 484–495.
- [5] R. Bortolameotti, A. Peter, M. H. Everts, and D. Bolzoni, "Indicators of malicious SSL connections," in *Network and System Security - 9th International Conference, NSS 2015, New York, NY, USA, November 3-5, 2015, Proceedings*, 2015, pp. 162–175.

Online Resources

- [6] M. Department for Digital Culture and Sport, *Cyber Security Breaches Survey 2018*, <https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2018>, Report, Accessed: 2018-09.
- [7] M. McGuire, *Into The Web of Profit: Understanding the Growth of the Cybercrime Economy*. <https://www.bromium.com/resource/into-the-web-of-profit/>, type, Accessed: 2018-09.
- [8] T. Micro, *Data Breach*, <https://www.trendmicro.com/vinfo/us/security/definition/data-breach>, Accessed: 2018-09.
- [9] B. L. Index, *Data Breach Database*. <https://breachlevelindex.com/data-breach-database>.
- [10] P. Institute, *Cost of a Data Breach Study*, <https://www.ibm.com/security/data-breach>, Accessed: 2018-09.
- [11] N. C. of State Legislatures (NCSL), *Security Breach Notification Laws*, <http://www.ncsl.org/research/telecommunications-and-information-technology/security-breach-notification-laws.aspx>, Accessed: 2018-09.

- [12] Verizon, *2018 Data Breach Investigations Report*, <http://www.verizonenterprise.com/verizon-insights-lab/dbir/tool/>, Accessed: 2018-09.
- [13] *Evading NIDS, revisited*. [Online]. Available: <https://www.symantec.com/connect/articles/evading-nids-revisited>.
- [14] M. Cruz, *Data Exfiltration in Targeted Attacks*, <https://blog.trendmicro.com/trendlabs-security-intelligence/data-exfiltration-in-targeted-attacks/>, Accessed: 2018-11.
- [15] M. Mimoso, *Information-Stealing Stegoloader Malware Hides in Images*, <https://threatpost.com/information-stealing-stegoloader-malware-hides-in-images/113337/>, Accessed: 2018-11.
- [16] N. Villeneuve, *Hiding in Plain Sight: The FAKEM Remote Access Trojan*, <https://blog.trendmicro.com/trendlabs-security-intelligence/hiding-in-plain-sight-the-fakem-remote-access-trojan/>, Accessed: 2018-11.
- [17] A. Nadler, *Effectively Detecting Low Throughput and Malicious DNS Exfiltration*, <https://blogs.akamai.com/2018/04/effectively-detecting-low-throughput-and-malicious-dns-exfiltration.html>, Accessed: 2018-11.
- [18] V. Valeros, *In plain sight: Credential and data stealing adware*, 2016. [Online]. Available: <http://blogs.cisco.com/security/in-plain-sight-credential-and-data-stealing-adware>.
- [19] Electronic Frontier Foundation, *Kaspersky User-Agent Strings - NSA*. [Online]. Available: <https://www.eff.org/it/node/86529>.
- [20] *EMOTET Returns, Starts Spreading via Spam Botnet*. [Online]. Available: <https://blog.trendmicro.com/trendlabs-security-intelligence/emotet-returns-starts-spreading-via-spam-botnet/>.
- [21] *PNFilter Malware Now Exploiting Endpoints, Not Just Routers*. [Online]. Available: <https://duo.com/decipher/vpnfilter-malware-now-exploiting-endpoints-not-just-routers>.
- [22] D. McMillen, *Wiper malware analysis*. [Online]. Available: <http://www.ibm.com/developerworks/library/se-wiper-analysis/index.html>.
- [23] McAfee, *Rise of rootkits*. [Online]. Available: <http://www.mcafee.com/us/security-awareness/articles/rise-of-rootkits.aspx>.
- [24] *LinkedIn lost 167 million account credentials in data breach*. [Online]. Available: <http://fortune.com/2016/05/18/linkedin-data-breach-email-password/>.
- [25] *2015 Cost of data breach study: Global Analysis*. [Online]. Available: <http://www-03.ibm.com/security/data-breach/>.
- [26] *Proof of Work. bitcoin wiki*. [Online]. Available: https://en.bitcoin.it/wiki/Proof_of_work.
- [27] *Proof of Stake. bitcoin wiki*. [Online]. Available: https://en.bitcoin.it/wiki/Proof_of_Stake.

Literature

- [28] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed nist standard for role-based access control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, no. 3, pp. 224–274, 2001.
- [29] A. Shabtai, Y. Elovici, and L. Rokach, *A survey of data leakage detection and prevention solutions*. Springer Science & Business Media, 2012.
- [30] K. Scarfone and P. Mell, "Guide to Intrusion Detection and Prevention Systems (IDPS)," *NIST special publication*, vol. 800, no. 2007, p. 94, 2007.
- [31] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM computing surveys (CSUR)*, vol. 44, no. 2, p. 6, 2012.
- [32] P. Chen, L. Desmet, and C. Huygens, "A study on advanced persistent threats," in *IFIP International Conference on Communications and Multimedia Security*, Springer, 2014, pp. 63–72.
- [33] S. Axelsson, "Intrusion detection systems: A survey and taxonomy," Technical report, Tech. Rep., 2000.
- [34] N. Idika and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, vol. 48, 2007.
- [35] R. Zuech, T. M. Khoshgoftaar, and R. Wald, "Intrusion detection and Big Heterogeneous Data: a Survey," *J. Big Data*, vol. 2, p. 3, 2015.
- [36] D. Bolzoni, "Revisiting anomaly-based network intrusion detection systems," PhD thesis, University of Twente, Enschede, Netherlands, 2009.
- [37] C. Rossow, C. J. Dietrich, H. Bos, L. Cavallaro, M. van Steen, F. C. Freiling, and N. Pohlmann, "Sandnet: Network traffic analysis of malicious software," in *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, BADGERS@EuroSys 2011, Salzburg, Austria, April 10, 2011*, 2011, pp. 78–88.
- [38] A. Zaras, A. Papadogiannakis, R. Gawlik, and T. Holz, "Automated generation of models for fast and precise detection of http-based malware," in *2014 Twelfth Annual International Conference on Privacy, Security and Trust, Toronto, ON, Canada, July 23-24, 2014*, 2014, pp. 249–256.
- [39] A. K. Sood, S. Zeadally, and R. J. Enbody, "An empirical study of http-based financial botnets," *IEEE Trans. Dependable Sec. Comput.*, vol. 13, no. 2, pp. 236–251, 2016.
- [40] K. Borders and A. Prakash, "Web tap: Detecting covert web traffic," in *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, 2004, pp. 110–120.
- [41] M. Dusi, M. Crotti, F. Gringoli, and L. Salgarelli, "Tunnel hunter: Detecting application-layer tunnels with statistical fingerprinting," *Computer Networks*, vol. 53, no. 1, pp. 81–97, 2009.

- [42] G. Schwenk and K. Rieck, "Adaptive detection of covert communication in HTTP requests," in *Seventh European Conference on Computer Network Defense, EC2ND 2011, Gothenburg, Sweden, September 6-7, 2011*, 2011, pp. 25–32.
- [43] Y. Baychev and L. Bilge, "Spearphishing malware: Do we really know the unknown?" In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2018, pp. 46–66.
- [44] S. Vömel and F. C. Freiling, "A survey of main memory acquisition and analysis techniques for the windows operating system," *Digital Investigation*, vol. 8, no. 1, pp. 3–22, 2011.
- [45] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014.
- [46] S. Soltani and S. A. H. Seno, "A survey on digital evidence collection and analysis," in *Computer and Knowledge Engineering (ICCKE), 2017 7th International Conference on*, IEEE, 2017, pp. 247–253.
- [47] E. S. Pilli, R. C. Joshi, and R. Niyogi, "Network forensic frameworks: Survey and research challenges," *Digital Investigation*, vol. 7, no. 1-2, pp. 14–27, 2010.
- [48] K. D. Bowers, C. Hart, A. Juels, and N. Triandopoulos, "Pillarbox: Combating next-generation malware with fast forward-secure logging," in *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2014, pp. 46–67.
- [49] D. Ma and G. Tsudik, "A new approach to secure logging," *TOS*, vol. 5, no. 1, 2:1–2:21, 2009.
- [50] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, 2009, pp. 169–178.
- [51] A. C. Yao, "Protocols for secure computations (extended abstract)," in *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, 1982, pp. 160–164.
- [52] A. M. Bates, D. Tian, K. R. B. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 319–334.
- [53] J. J. Davis and E. Foo, "Automated feature engineering for HTTP tunnel detection," *Computers & Security*, vol. 59, pp. 166–185, 2016.
- [54] J. P. Anderson, "Computer security threat monitoring and surveillance," *Technical Report*, James P. Anderson Company, 1980.
- [55] D. Denning and P. G. Neumann, *Requirements and model for IDES-a real-time intrusion-detection expert system*. SRI International, 1985.

- [56] D. E. Denning, “An intrusion-detection model,” in *Proceedings of the 1986 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 7-9, 1986*, 1986, pp. 118–133.
- [57] P. Garcia-Teodoro, J. E. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, “Anomaly-based network intrusion detection: Techniques, systems and challenges,” *Computers & Security*, vol. 28, no. 1-2, pp. 18–28, 2009.
- [58] M. Z. Raffique and J. Caballero, “FIRMA: Malware Clustering and Network Signature Generation with Mixed Network Behaviors,” in *Research in Attacks, Intrusions, and Defenses - 16th International Symposium, RAID 2013, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings*, Springer, 2013, pp. 144–163.
- [59] R. Perdisci, W. Lee, and N. Feamster, “Behavioral clustering of http-based malware and signature generation using malicious network traces,” in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*, USENIX Association, 2010, pp. 391–404.
- [60] R. Perdisci, D. Ariu, and G. Giacinto, “Scalable Fine-grained Behavioral Clustering of HTTP-based Malware,” *Computer Networks*, vol. 57, no. 2, pp. 487–500, 2013.
- [61] T. Nelms, R. Perdisci, and M. Ahamad, “Execscent: Mining for new c&c domains in live networks with adaptive control protocol templates,” in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, 2013, pp. 589–604.
- [62] K. Bartos, M. Sofka, and V. Franc, “Optimized invariant representation of network traffic for detecting unseen malware variants,” in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 807–822.
- [63] M. Caselli, E. Zambon, J. Amann, R. Sommer, and F. Kargl, “Specification mining for intrusion detection in networked control systems,” in *USENIX Security Symposium*, 2016, pp. 791–806.
- [64] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou, “Specification-based anomaly detection: A new approach for detecting network intrusions,” in *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, 2002, pp. 265–274.
- [65] H. Debar, M. Dacier, and A. Wespi, “A revised taxonomy for intrusion-detection systems,” in *Annales des télécommunications*, Springer, vol. 55, 2000, pp. 361–378.
- [66] W. Lee and S. J. Stolfo, “Data mining approaches for intrusion detection,” in *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*, 1998.

- [67] C. Warrender, S. Forrest, and B. A. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *1999 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 9-12, 1999*, 1999, pp. 133–145.
- [68] A. K. Ghosh, A. Schwartzbard, and M. Schatz, "Learning program behavior profiles for intrusion detection," in *Proceedings of the Workshop on Intrusion Detection and Network Monitoring, Santa Clara, CA, USA, April 9-12, 1999*, 1999, pp. 51–62.
- [69] A. G. Pennington, J. D. Strunk, J. L. Griffin, C. A. N. Soules, G. R. Goodson, and G. R. Ganger, "Storage-based intrusion detection: Watching storage activity for suspicious behavior," in *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, 2003.
- [70] M. B. Salem and S. J. Stolfo, "Modeling user search behavior for masquerade detection," in *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2011, pp. 181–200.
- [71] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An overview of IP flow-based intrusion detection," *IEEE Communications Surveys and Tutorials*, vol. 12, no. 3, pp. 343–356, 2010.
- [72] R. Hofstede, P. Celeda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow monitoring explained: From packet capture to data analysis with netflow and IPFIX," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 4, pp. 2037–2064, 2014.
- [73] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson, "The security impact of HTTPS interception," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.
- [74] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "Blindbox: Deep packet inspection over encrypted traffic," *Computer Communication Review*, vol. 45, no. 5, pp. 213–226, 2015.
- [75] S. Dharmapurikar and V. Paxson, "Robust TCP stream reassembly in the presence of adversaries," in *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*, 2005.
- [76] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*, 2010, pp. 305–316.
- [77] F. Valeur, G. Vigna, C. Krügel, and R. A. Kemmerer, "A comprehensive approach to intrusion detection alert correlation," *IEEE Trans. Dependable Sec. Comput.*, vol. 1, no. 3, pp. 146–169, 2004.

- [78] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. A. Kemmerer, C. Kruegel, and G. Vigna, "Your botnet is my botnet: Analysis of a botnet takeover," in *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, 2009, pp. 635–647.
- [79] A. Continella, A. Guagnelli, G. Zingaro, G. D. Pasquale, A. Barengi, S. Zanero, and F. Maggi, "Shieldfs: A self-healing, ransomware-aware filesystem," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, 2016, pp. 336–347.
- [80] A. Continella, M. Carminati, M. Polino, A. Lanzi, S. Zanero, and F. Maggi, "Prometheus: Analyzing webinject-based information stealers," *Journal of Computer Security*, vol. 25, no. 2, pp. 117–137, 2017.
- [81] C. J. Dietrich, C. Rossow, F. C. Freiling, H. Bos, M. van Steen, and N. Pohlmann, "On botnets that use DNS for command and control," in *Seventh European Conference on Computer Network Defense, EC2ND 2011, Gothenburg, Sweden, September 6-7, 2011*, 2011, pp. 9–16.
- [82] S. Wendzel, S. Zander, B. Fechner, and C. Herdin, "Pattern-based survey and categorization of network covert channel techniques," *ACM Comput. Surv.*, vol. 47, no. 3, 50:1–50:26, 2015.
- [83] T. G. Handel and M. T. S. II, "Hiding data in the OSI network model," in *Information Hiding, First International Workshop, Cambridge, UK, May 30 - June 1, 1996, Proceedings*, 1996, pp. 23–38.
- [84] S. Zander, G. J. Armitage, and P. Branch, "A survey of covert channels and countermeasures in computer network protocols," *IEEE Communications Surveys and Tutorials*, vol. 9, no. 1-4, pp. 44–57, 2007.
- [85] D. Llamas, C. Allison, and A. Miller, "Covert channels in internet protocols: A survey," in *Proceedings of the 6th Annual Postgraduate Symposium about the Convergence of Telecommunications, Networking and Broadcasting, PGNET*, vol. 2005, 2005.
- [86] K. Cabaj, L. Caviglione, W. Mazurczyk, S. Wendzel, A. Woodward, and S. Zander, "The new threats of information hiding: The road ahead," *IT Professional*, vol. 20, no. 3, pp. 31–39, 2018.
- [87] S. Hammouda, L. Maalej, and Z. Trabelsi, "Towards optimized TCP/IP covert channels detection, IDS and firewall integration," in *NTMS 2008, 2nd International Conference on New Technologies, Mobility and Security, November 5-7, 2008, Tangier, Morocco*, 2008, pp. 1–5.
- [88] J. Zhai, G. Liu, and Y. Dai, "A covert channel detection algorithm based on tcp markov model," in *Multimedia Information Networking and Security (MINES), 2010 International Conference on*, IEEE, 2010, pp. 893–897.
- [89] H. Zhao and Y. Shi, "Detecting covert channels in computer networks based on chaos theory," *IEEE Trans. Information Forensics and Security*, vol. 8, no. 2, pp. 273–282, 2013.

- [90] S. J. Murdoch and S. Lewis, "Embedding covert channels into TCP/IP," in *Information Hiding, 7th International Workshop, IH 2005, Barcelona, Spain, June 6-8, 2005, Revised Selected Papers*, 2005, pp. 247–261.
- [91] V. Paxson, M. Christodorescu, M. Javed, J. R. Rao, R. Sailer, D. L. Schales, M. P. Stoecklin, K. Thomas, W. Z. Venema, and N. Weaver, "Practical comprehensive bounds on surreptitious communication over DNS," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, 2013, pp. 17–32.
- [92] A. Nadler, A. Aminov, and A. Shabtai, "Detection of malicious and low throughput data exfiltration over the DNS protocol," *CoRR*, vol. abs/1709.08395, 2017.
- [93] E Cambiaso, M Aiello, M Mongelli, and G Papaleo, "Feature transformation and mutual information for dns tunneling analysis," in *Ubiquitous and Future Networks (ICUFN), 2016 Eighth International Conference on*, IEEE, 2016, pp. 957–959.
- [94] I. Homem, P. Papapetrou, and S. Dosis, "Entropy-based prediction of network protocols in the forensic analysis of dns tunnels," *arXiv preprint arXiv:1709.06363*, 2017.
- [95] W. Ellens, P. Zuraniewski, A. Sperotto, H. Schotanus, M. Mandjes, and E. Meeuwissen, "Flow-based detection of DNS tunnels," in *Emerging Management Mechanisms for the Future Internet - 7th IFIP WG 6.6 International Conference on Autonomous Infrastructure, Management, and Security, AIMS 2013, Barcelona, Spain, June 25-28, 2013. Proceedings*, 2013, pp. 124–135.
- [96] M. Aiello, M. Mongelli, and G. Papaleo, "DNS tunneling detection through statistical fingerprints of protocol messages and machine learning," *Int. J. Communication Systems*, vol. 28, no. 14, pp. 1987–2002, 2015.
- [97] D. J. Pack, W. Streilein, S. Webster, and R. Cunningham, "Detecting http tunneling activities," Massachusetts Institute of Tech Lexington Lincoln Lab, Tech. Rep., 2002.
- [98] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli, "Traffic classification through simple statistical fingerprinting," *Computer Communication Review*, vol. 37, no. 1, pp. 5–16, 2007.
- [99] K. Borders and A. Prakash, "Quantifying information leaks in outbound web traffic," in *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA, 2009*, pp. 129–140.
- [100] M. Marchetti, F. Pierazzi, M. Colajanni, and A. Guido, "Analysis of high volumes of network traffic for advanced persistent threat detection," *Computer Networks*, vol. 109, pp. 127–141, 2016.
- [101] G. He, T. Zhang, Y. Ma, and B. Xu, "A novel method to detect encrypted data exfiltration," in *Second International Conference on Advanced Cloud and Big Data, CBD 2014, Huangshan, China, November 20-22, 2014*, 2014, pp. 240–246.

- [102] G. Gu, J. Zhang, and W. Lee, “Botsniffer: Detecting botnet command and control channels in network traffic,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008.
- [103] G. Gu, R. Perdisci, J. Zhang, and W. Lee, “Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection,” in *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, 2008, pp. 139–154.
- [104] F. Giroire, J. Chandrashekar, N. Taft, E. M. Schooler, and D. Papagianaki, “Exploiting temporal persistence to detect covert botnet channels,” in *Recent Advances in Intrusion Detection, 12th International Symposium, RAID 2009, Saint-Malo, France, September 23-25, 2009. Proceedings*, 2009, pp. 326–345.
- [105] S. Venkataraman, J. Caballero, P. Poosankam, M. G. Kang, and D. X. Song, “Fig: Automatic fingerprint generation,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*, 2007.
- [106] C. Neasbitt, R. Perdisci, K. Li, and T. Nelms, “Clickminer: Towards forensic reconstruction of user-browser interactions from network traces,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, 2014, pp. 1244–1255.
- [107] H. Zhang, D. D. Yao, N. Ramakrishnan, and Z. Zhang, “Causality Reasoning about Network Events for Detecting Stealthy Malware Activities,” *Computers & Security*, vol. 58, pp. 180–198, 2016.
- [108] G. Xie, M. Iliofotou, T. Karagiannis, M. Faloutsos, and Y. Jin, “ReSurf: Reconstructing Web-Surfing Activity from Network Traffic,” in *IFIP Networking Conference, 2013, Brooklyn, New York, USA, 22-24 May, 2013*, 2013, pp. 1–9.
- [109] V. Paxson, “Bro: A System for Detecting Network Intruders in Real-time,” *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.
- [110] E. Zielinska, W. Mazurczyk, and K. Szczypiorski, “Trends in Steganography,” *Communications of ACM*, vol. 57, no. 3, pp. 86–95, 2014.
- [111] S. Alneyadi, E. Sithirasanen, and V. Muthukkumarasamy, “Detecting data semantic: A data leakage prevention approach,” in *IEEE TrustCom/ Big-DataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, IEEE, 2015, pp. 910–917.
- [112] X. Shu and D. Yao, “Data Leak Detection as a Service,” in *Security and Privacy in Communication Networks - 8th International ICST Conference, SecureComm 2012, Padua, Italy, September 3-5, 2012.*, Springer, 2012, pp. 222–240.

- [113] X. Shu, D. Yao, and E. Bertino, "Privacy-Preserving Detection of Sensitive Data Exposure," *IEEE Trans. Information Forensics and Security*, vol. 10, no. 5, pp. 1092–1103, 2015.
- [114] M. Hart, P. K. Manadhata, and R. Johnson, "Text Classification for Data Loss Prevention," in *Privacy Enhancing Technologies - 11th International Symposium, PETS 2011, Waterloo, ON, Canada, July 27-29, 2011. Proceedings*, Springer, 2011, pp. 18–37.
- [115] X. Shu, J. Zhang, D. Yao, and W. Feng, "Fast Detection of Transformed Data Leaks," *IEEE Trans. Information Forensics and Security*, vol. 11, no. 3, pp. 528–542, 2016.
- [116] G. Katz, Y. Elovici, and B. Shapira, "CoBAN: A Context-based Model for Data Leakage Prevention," *Information Sciences*, vol. 262, pp. 137–158, 2014.
- [117] A. Al-Bataineh and G. White, "Analysis and detection of malicious data exfiltration in web traffic," in *7th International Conference on Malicious and Unwanted Software, MALWARE 2012, Fajardo, PR, USA, October 16-18, 2012*, IEEE Computer Society, 2012, pp. 26–31.
- [118] T. Dougan and K. Curran, "Man in the browser attacks," *IJACI*, vol. 4, no. 1, pp. 29–39, 2012.
- [119] S. Boriah, V. Chandola, and V. Kumar, "Similarity measures for categorical data: A comparative evaluation," in *Proceedings of the SIAM International Conference on Data Mining, SDM 2008, April 24-26, 2008, Atlanta, Georgia, USA, 2008*, pp. 243–254.
- [120] D. Bakkelund, "An LCS-based string metric," *Oslo, Norway: University of Oslo*, 2009.
- [121] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-resilient privacy leak detection for mobile apps through differential analysis," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.
- [122] J. Nazario and T. Holz, "As the net churns: Fast-flux botnet observations," in *3rd International Conference on Malicious and Unwanted Software, MALWARE 2008, Alexandria, Virginia, USA, October 7-8, 2008*, 2008, pp. 24–31.
- [123] J. W. Ratcliff and D. E. Metzener, "Pattern-matching-the gestalt approach," *Dr Dobbs Journal*, vol. 13, no. 7, p. 46, 1988.
- [124] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

- [125] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, “FP-STALKER: tracking browser fingerprint evolutions,” in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, 2018, pp. 728–741.
- [126] D. A. Wagner and P. Soto, “Mimicry attacks on host-based intrusion detection systems,” in *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, 2002, pp. 255–264.
- [127] A. Houmansadr, C. Brubaker, and V. Shmatikov, “The parrot is dead: Observing unobservable network communications,” in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, 2013, pp. 65–79.
- [128] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton, “Protocol misidentification made easy with format-transforming encryption,” in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS’13, Berlin, Germany, November 4-8, 2013*, 2013, pp. 61–72.
- [129] C. V. Wright, S. E. Coull, and F. Monrose, “Traffic morphing: An efficient defense against statistical traffic analysis,” in *NDSS*, 2009.
- [130] H. M. Moghaddam, B. Li, M. Derakhshani, and I. Goldberg, “Skypemorph: Protocol obfuscation for tor bridges,” in *ACM CCS 2012*, 2012, pp. 97–108.
- [131] Z. Weinberg, J. Wang, V. Yegneswaran, L. Briesemeister, S. Cheung, F. Wang, and D. Boneh, “Stegotorus: A camouflage proxy for the tor anonymity system,” in *ACM CCS 2012*, 2012, pp. 109–120.
- [132] P. Fogla, M. I. Sharif, R. Perdisci, O. M. Kolesnikov, and W. Lee, “Polymorphic blending attacks,” in *USENIX Security 2006*, 2006.
- [133] P. Fogla and W. Lee, “Evading network anomaly detection systems: Formal reasoning and practical techniques,” in *ACM CCS 2006*, 2006, pp. 59–68.
- [134] M. Casenove, “Exfiltrations using polymorphic blending techniques: Analysis and countermeasures,” in *7th International Conference on Cyber Conflict: Architectures in Cyberspace, CyCon 2015, Tallinn, Estonia, May 26-29, 2015*, 2015, pp. 217–230.
- [135] F. V. Yarochkin, S. Dai, C. Lin, Y. Huang, and S. Kuo, “Towards adaptive covert communication system,” in *14th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2008, 15-17 December 2008, Taipei, Taiwan*, 2008, pp. 153–159.
- [136] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, “Toward generating a new intrusion detection dataset and intrusion traffic characterization,” in *Proceedings of the 4th International Conference on Information Systems Security and Privacy, ICISSP 2018, Funchal, Madeira - Portugal, January 22-24, 2018.*, 2018, pp. 108–116.
- [137] X. Han, N. Kheir, and D. Balzarotti, “Deception techniques in computer security: A research perspective,” *ACM Comput. Surv.*, vol. 51, no. 4, 80:1–80:36, 2018.

- [138] V. Pappas, V. P. Kemerlis, A. Zavou, M. Polychronakis, and A. D. Keromytis, "Cloudfence: Data flow tracking as a cloud service," in *Research in Attacks, Intrusions, and Defenses - 16th International Symposium, RAID 2013, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings*, 2013, pp. 411–431.
- [139] I. Papagiannis and P. R. Pietzuch, "Cloudfilter: Practical control of sensitive data propagation to the cloud," in *Proceedings of the 2012 ACM Workshop on Cloud computing security, CCSW 2012, Raleigh, NC, USA, October 19, 2012.*, 2012, pp. 97–102.
- [140] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Tainteraser: Protecting sensitive data leaks using application-level taint tracking," *Operating Systems Review*, vol. 45, no. 1, pp. 142–154, 2011.
- [141] K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems," in *Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA, May 30 - June 3, 2006*, 2006, pp. 43–56.
- [142] A. Gehani and D. Tariq, "SPADE: support for provenance auditing in distributed environments," in *Middleware 2012 - ACM/IFIP/USENIX 13th International Middleware Conference, Montreal, QC, Canada, December 3-7, 2012. Proceedings*, 2012, pp. 101–120.
- [143] D. J. Pohly, S. E. McLaughlin, P. D. McDaniel, and K. R. B. Butler, "Hi-fi: Collecting high-fidelity whole-system provenance," in *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, 2012, pp. 259–268.
- [144] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.
- [145] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, 2009, pp. 153–168.
- [146] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong, "Zyzyva: Speculative byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, 7:1–7:39, 2009.
- [147] J. K. Resch and J. S. Plank, "AONT-RS: blending security and performance in dispersed storage systems," in *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, 2011, pp. 191–202.
- [148] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

- [149] M. J. Fischer, N. A. Lynch, and M. Paterson, “Impossibility of distributed consensus with one faulty process,” in *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 21-23, 1983, Colony Square Hotel, Atlanta, Georgia, USA, 1983*, pp. 1–7.
- [150] G. E. Suh and S. Devadas, “Physical unclonable functions for device authentication and secret key generation,” in *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*, 2007, pp. 9–14.
- [151] A. N. Bessani, J. Sousa, and E. A. P. Alchieri, “State machine replication for the masses with BFT-SMART,” in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, 2014, pp. 355–362.
- [152] J. Daugman, “How iris recognition works,” in *Proceedings of the 2002 International Conference on Image Processing, ICIIP 2002, Rochester, New York, USA, September 22-25, 2002*, 2002, pp. 33–36.
- [153] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger, “Self-securing storage: Protecting data in compromised systems,” in *4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000*, 2000, pp. 165–180.
- [154] J. Wires and M. J. Feeley, “Secure file system versioning at the block level,” in *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*, 2007, pp. 203–215.
- [155] P. Macko and M. Seltzer, “A general-purpose provenance library,” in *4th Workshop on the Theory and Practice of Provenance, TaPP’12, Boston, MA, USA, June 14-15, 2012*, 2012.
- [156] R. Hasan, R. Sion, and M. Winslett, “The case of the fake picasso: Preventing history forgery with secure provenance,” in *7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA, USA. Proceedings*, 2009, pp. 1–14.
- [157] K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. L. MacLean, D. W. Margo, M. I. Seltzer, and R. Smogor, “Layering in provenance systems,” in *2009 USENIX Annual Technical Conference, San Diego, CA, USA, June 14-19, 2009*, 2009.
- [158] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr, “Secure network provenance,” in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, 2011, pp. 295–310.