SAPIENZA
UNIVERSITÀ DI ROMA

Critical Infrastructures Security: Improving Defense Against Novel Malware and Advanced Persistent Threats

Department of Computer, Control, and Management Engineering Antonio Ruberti, Sapienza – University of Rome

Ph.D. Program in Engineering in Computer Science – XXXII Cycle

Candidate

Giuseppe Laurenza
ID number 1192406

Thesis Advisor

Prof. Roberto Baldoni

Co-Advisors

Prof. Riccardo Lazzeretti
Prof. Leonardo Querzoni

February 2020

Thesis reviewed by (in alphabetically order):

Prof. Alessio Merlo
Prof. Camil Demetrescu (Internal Reviewer)
Prof. Corrado Aaron Visaggio

Critical Infrastructures Security: Improving Defense Against Novel Malware and Advanced Persistent Threats
Ph.D. thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: February 2020

Author's email: laurenza@diag.uniroma1.it

Abstract

Protection of Critical Infrastructures (CIs) is vital for the survival of society. Any functionality reduction or interruption can cause heavy damages to people. *Stuxnet* and *Wannacry* are clear proofs that the world is changed and now attackers target CIs mainly through cyber-space. The rapid evolution of adversaries' skills provokes an overwhelming raising in the difficulty of defense. Tons of malware are released every day and malware analysts cannot be fast enough to analyze all of them and react in time. Moreover, classical security software, such as anti-virus, cannot help, due to the huge knowledge required to recognize threats.

In this thesis, we present our ideas to reduce the problem and consequently improve Critical Infrastructures security. We observe that the main attack vector is malware, therefore we propose a semi-automatic architecture for malware analysis, which can help human analysts giving useful information and heavily reducing their workloads by prioritizing the *cutting-edge* and most dangerous malware.

Moreover, we focus on malware belonging to new *malware families* or developed by *Advanced Persistent Threats* (APTs), which pose a serious risk to CIs and hence deserve deeper inspection. We have hence developed useful tools, to be integrated into our architecture, able to group malware in families and recognize malware developed by APTs. We implement the first task through clustering and *online clustering*. This module can help to highly reduce the number of malware to be analyzed. Malware labeled as known families do not need additional investigation since their behavior is already studied. Moreover, it is possible to study only a small number of representatives from new groups to further reduce the workload. We fulfill the second task through a Triage approach. This task is fundamental to detect very dangerous malware. Being APTs the most threatening adversaries of CIs, detecting their activities as soon as possible is the only way to diminish the damage and possibly stop the attack.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Critical Infrastructure (CI) is a common term for referring to an *asset* or a *system* that is essential for the life of societal functions. CIs are so vital to a country that the incapacity or destruction of such systems and assets would have a debilitating impact on security, national economic security, national public health or safety, or any combination of those matters[1]. Each country or union has its list of sectors that are considered *critical* and produced a *Protection Program* like the US Presidential Directive PDD-63[2] or the European Program for Critical Infrastructure Protection (EPCIP)[3]. However some sectors, instead, are widely recognized as critical by many organizations, for example, Health-care or Energy services.

The protection of Critical Infrastructures is hence an important matter. Any damages or interruption of services can be very dangerous to affected people. It is easy to imagine the problem extent of having, for example, not working hospitals or black-outs in a large portion of the cities.

While in the past adversaries needed to act physically in the infrastructures, for example sabotaging devices, in recent times, the preferred mean for attacking a CI is through the cyber-space. It is easy to understand the advantages of acting remotely: it is less risky because an attacker can easily remain anonymous and he can perform multiple attacks to multiple victims with less efforts. Deep investigating remote attacks, it results clear that most of them rely on *malware infection* as the main attack vector and this trend is not changing during last year[4].

Malware is a short-term for *malicious software* and it groups all computer programs designed to infiltrate other computer systems, damage the target's infrastructure or gather sensible information. This definition suggests why malware are fundamental for attacks. An attack is carried out through different actions that can be *easily* made through various malicious software. Moreover, malware can be modified or specifically developed to attack a particular vulnerability in the target, making possible to overcome also the most advanced defenses. Malware exist mainly because there are vulnerabilities to exploit. These vulnerabilities can be related to different *characteristics* of the device, like at the firmware level of the machine, or

[1] https://csrc.nist.gov/glossary/term/critical-infrastructure

[2] https://fas.org/irp/offdocs/pdd/pdd-63.htm

[3] https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32008L0114&from=EN

[4] https://www.checkpoint.com/downloads/resources/cyber-attack-trends-mid-year-report-2019.pdf

the operative system and they can be even related to the *human users*. CIs systems are composed of various classes of devices, starting from PC with Windows to server with Linux distributions, from *management* interfaces to specialized machines both with custom firmware and operative systems. All these devices can be victims of an attack, even only as a *step* to reach other elements of the network. Thus, hardening should be applied to all the machines in order to avoid the spreading of an infection. CIs' adversaries try to attack each exposed point, thus fixing vulnerabilities should be one of the first steps to perform. However, it is not possible to patch all of them, first because they can be not discovered by *the good ones*, second because the patch can be not already available or even not currently applicable. The latter case is typically related to patches that required shutdowns or reboots. In fact, in complex systems, due to system availability or dependability, it is not always possible to interrupt one of the components without interrupting the entire service. Moreover CIs include also systems that are *morphologically* different from classical computer systems. For example, a power grid is composed of a huge number of different devices that need to be connected and remotely monitored or managed. Thus, the number of possible vulnerabilities and hence of *fixing strategies* is very high and it is also continuously growing with the technology evolution. In fact, with new inventions and discoveries, also CIs are changing. Following the previous example, in recent times, there is the introduction of the *smart grids*, that are classical power grids with, also, a variety of operation and energy meters including smart meters, smart appliances, renewable energy resources, and energy-efficient resources to strongly improve the quality of the service. All of these additions can be sources of other vulnerabilities. So CIs need a continuous monitor and analysis of vulnerabilities of their systems.

However, as explained before, this could not be enough, because it is not possible to discover and fix all of them, or attackers are exploiting some unknown vulnerability. Thus we also need to observe attack vectors that can take advantage of vulnerabilities and, as previously demonstrated, the main attack vectors against CIs are malware. One recent example of cyber-attack carried by malware to CIs was the famous Wannacry [75]. In 2017 Wannacry hit dozens of nations, encrypting computer data on millions of devices and asking for ransom, blocking the activities of various Critical Infrastructures. One of the most affected countries was Britain[5] that was forced to close even wards and emergency rooms, leaving many people without medical treatments. Wannacry takes advantage of the *EternalBlue*[6], an exploit suspected to be developed by United States National Security Agency (NSA) that take advantage of a vulnerability of Server Message Block (SMB) protocol.

Infections can arrive in many ways, for example, some years ago an attack was carried by a *forged* PowerPoint document sent through emails. As it is possible to see in Figure 1.1, this document contained a macro that downloaded the *second-step* malware and started the infection, opening even a **SSH backdoor**. This malware, called BlackEnergy [54], was used to take control of a PC connected to the energy management system and to turn off-line about 30 electric transformation

[5] `https://www.cbsnews.com/news/hospitals-across-britain-hit-by-ransomware-cyberattack/`

[6] `https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Trojan:Win32/EternalBlue&ThreatID=-2147239042`

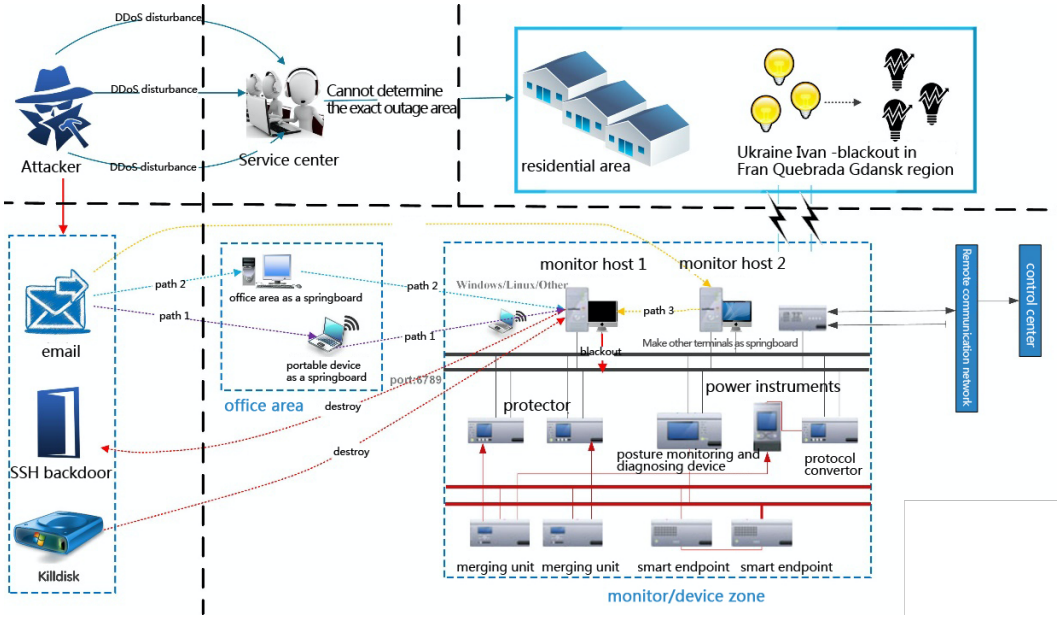


Figure 1.1. Schema of Blackenergy attack.

substations, resulting in about 230000 residents in the *dark*. Luckily, there were no victims, but people had taken a great risk for their life with half of a city blacked out due to the cyber-attack. An interesting characteristic of this attack was that it involved also a Distributed Denial of Service to the Call Service Center of the infrastructure to prevent them to rapidly intervene.

Another demonstration of the skills of CIs attackers was Stuxnet[23]. It is a threat that was primarily written to target an industrial control system or set of similar systems. Industrial control systems are used in gas pipelines and power plants. In its first version, it was used to attack the Iranian Nuclear Plant to sabotage the uranium enrichment facility at Natanz. In there, the malware was used to damage the centrifuge work, tampering the monitor systems at the same time, in order to make the modifications undetected. Fortunately, it limited its effort reducing only the performance of the facility in the *uranium enrichment* phase, but with similar access, it was possible to achieve more dangerous objectives, like cause even a new *Chernobyl*. Figure 1.2 shows how the Stuxnet malware worked. It attacked a *Closed system* that was not reachable from outside using an infected USB stick of one of the engineers working at the target as a vector. Then it compromised victims controllers using four different zero-day vulnerabilities to traverse many different machines in order to arrive in the *SCADA* devices. This *journey* was required because there was not a way to directly infect the target system, however, using other vulnerabilities it has been possible to compromise other devices that lead to the final target, one step at a time. The last phase of the infection is to check if the target is reached and then start the real *dangerous* operations.

As demonstrated by all these examples, CI can be put at risk and need to be protected. One way of improving CI security is enhancing protection against malware. The main difficult when facing malicious software infections is the incredibly rapid evolution of malware development, that lead to the impressive number of new

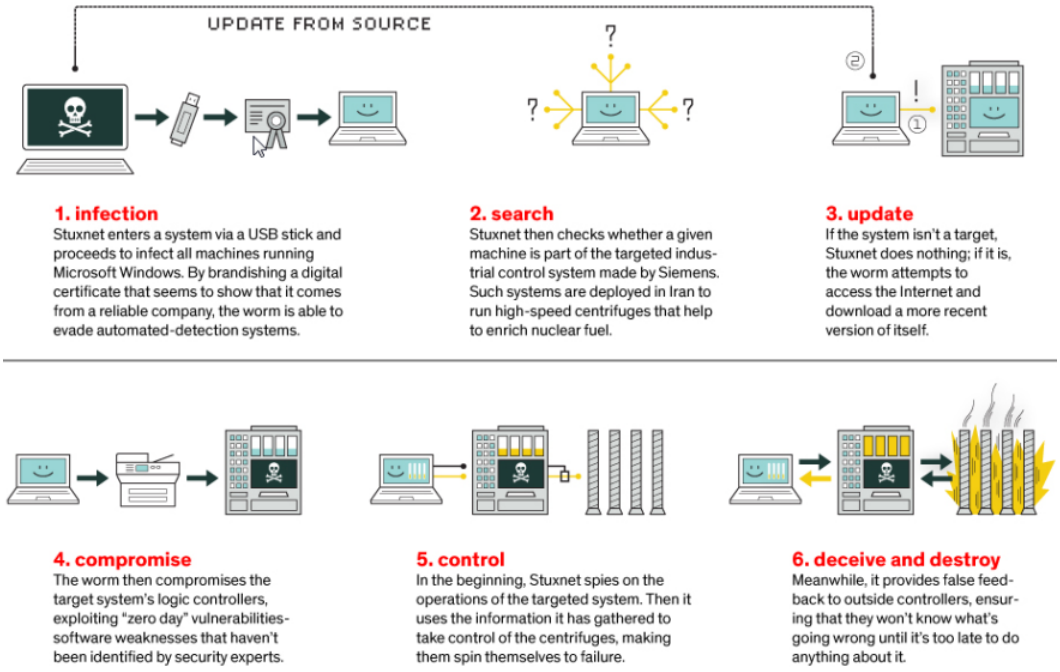


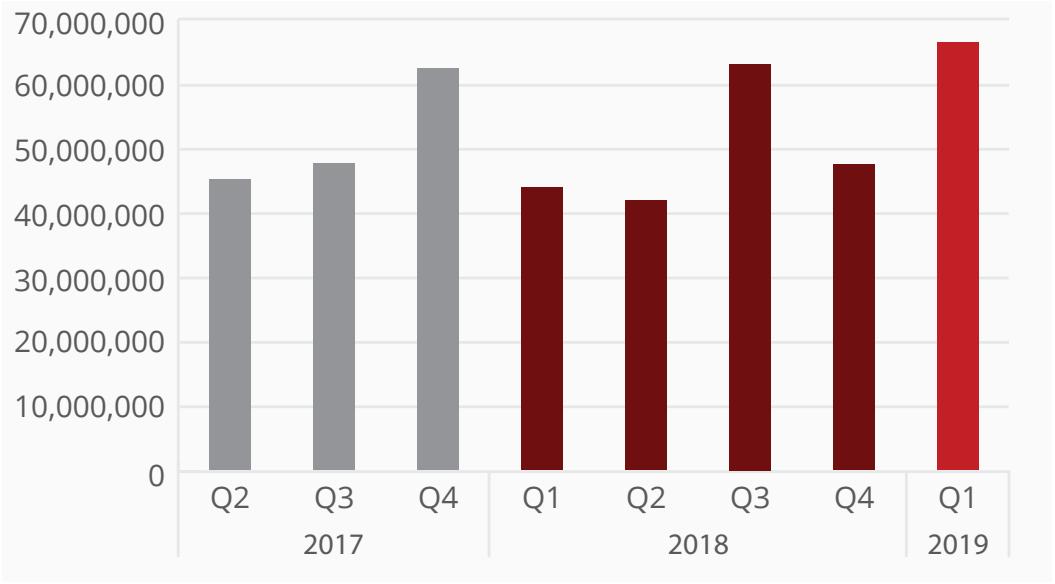
Figure 1.2. Schema of Stuxnet infection.

malware discovered every day. Figure 1.3 shows the magnitude of the problem, more than 60 millions of new malware were discovered in the first four months of 2019, a release speed of 6 new malware each second.

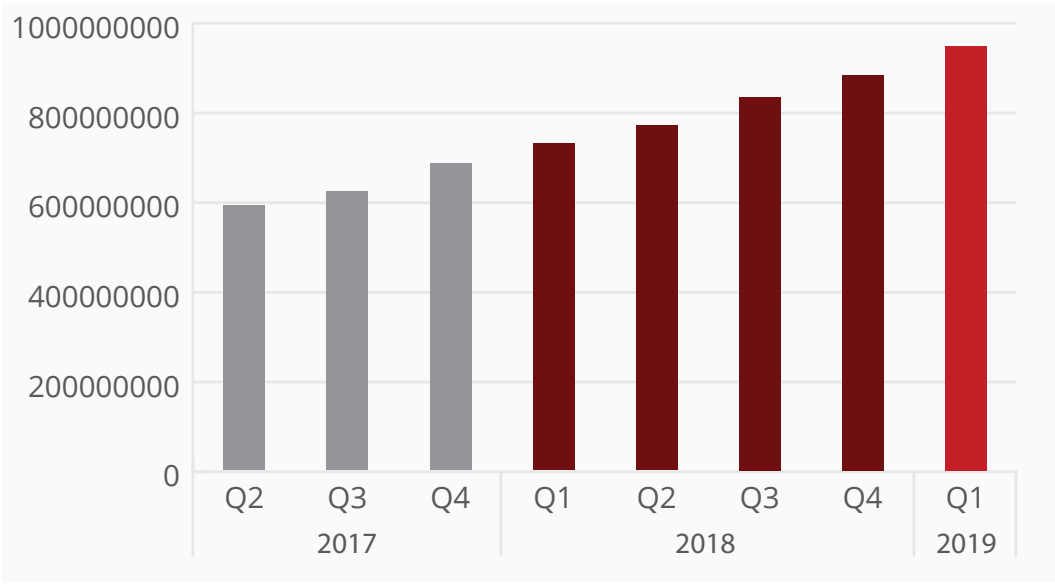
A deep investigation is required to correctly understand the behavior and dangerousness of malware and this task can be fulfilled only by human experts, but security analysts are a limited resource. To overcome the lack of experts there is the need for automatic and semi-automatic tools to reduce the number of samples requiring manual analysis. Relying on classical anti-virus tools is not enough. Common anti-virus software are developed with one or more engine based on approaches that can be roughly divided into three categories:

- static signature analysis;
- static heuristic analysis;
- dynamic analysis.

All of them present several limitations [69]. Static signature requires the observation of the malicious samples to produce the signature of the binary that is necessary to recognize it in the future. Heuristic analysis can be easily bypassed ensuring that all the malicious code is hidden, for example through code encryption. Dynamic analysis model can be hindered in different ways due to the limits of the analysis environment. In fact, the analysis of the AV can't last too long without a burden on the users and it is difficult to make realistic simulation environments without using many resources. Thus there are some limitations in which and how many operations the AV can emulate during the analysis. Moreover, many malware are



(a) Number of new discovered malware



(b) Total known malware

Figure 1.3. Malware Statistics. Source: McAfee Labs, 2019[63].

environmentally conscious and they do not perform any harmful operations if they detect fake environments. Thus anti-virus can not recognize them.

An alternative can be malware prioritization, analyzing first the most *interesting* ones. Defining interesting can be tricky because it can vary depending on the context. When dealing with the security of Critical Infrastructures, two criteria should lead the choice: **Novelties** and **Dangerousness**. Observing malware with novel characteristics is important because it can present new techniques to which currently there are not any defenses. Dangerousness is also an important criterion

because it is obviously safer to focus first on the most critical threats.

Thus the problem becomes how to rapidly prioritize malware satisfying these two criteria. For the first criterion, one way can be to group them and then analyze only a small number of *representative samples* and also, analyze only malware belonging to completely new groups, because the old ones were already been deep investigated. A typical approach is to rely on the concept of *malware families* to divide malicious software into groups of similar elements; using the definitions proposed by Ghanaei *et al.* [25]:

- A malware **X** is a *variant* of another malware **Y** if X can be obtained from Y by applying some mutations (malware X and Y share considerable portions of critical code);

- Variants of the same malware belong to the same *family*.

Thus, considering only these representative samples from new families as *interesting* is possible to strongly reduce the number of malware those require human attention. For example, Sophos reported that there exist more than 12000 unique variants of Wannacry[7], thus through family identification it is possible to avoid manual analysis of most of them because they are part of the same group. Ugarte-Pedrero *et al.* [96] report that about 70% of malware discovered daily by security companies belong to known families, leaving only a small part of completely unknown samples. Moreover, it is possible to group in families also the remaining malware, thus reducing, even more, the number of malware requiring manual analysis.

Instead, the dangerousness prioritization criterion can be quite easy to define. In fact, some malware are part of sophisticated and target-oriented cyber-attacks, which often leverage customized malicious software to remotely control the victims and use them for accessing valuable information inside an enterprise or institutional network target. According to NIST Glossary of Key Information Security Terms[8], such "*adversary that possesses sophisticated levels of expertise and significant resources which allow it to create opportunities to achieve its objectives by using multiple attack vectors (e.g.,cyber, physical and deception)*" is known as Advanced Persistent Threat (APT). APTs are the most dangerous attackers that can target a CI, the name is self-explaining:

- **Advanced**: groups are composed mainly of very skilled people. They usually take advantage of zero-day vulnerabilities discovered by them or bought on the black market.

- **Persistent**: one of their main distinguishing characteristics is that an APT attack can last for years. They keep updating their malware to maintain a persistent foothold in the target. For example, Kasperksy traces the beginning of the *Octopus* APT activities back to 1990 and this group is active also in these days.

[7] `https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/WannaCry-Aftershock.pdf`

[8] `http://nvlpubs.nist.gov/nistpubs/ir/2013/NIST.IR.7298r2.pdf`

- **Threat**: they perform strong attacks, that are perfectly tailored to the chosen victims. This is another one of their main dangerous characteristics. In fact, they invest a considerable amount of resources to study how to overcome target defenses.

The previously described Stuxnet is a perfect example of an APT attack: it infected an employee USB stick to infect the first machine, then they took advantage of four zero-day vulnerability to bypass network protections and obtain access to the other devices. Moreover, they developed the Stuxnet malware to only reproduce itself on all devices, leaving the *damaging* component disabled until the binary found itself on some particular model of controlling devices made by a particular brand, of course, these machines were the ones of the targeted nuclear plant.

Therefore, our thesis is focused on improving Critical Infrastructures defenses against malware infections. Taking into account previously described issues, we focus on providing an architecture for malware analysis to CI, designing also methodologies and modules to reduce the effort required from human analysts. In our work, we wanted to make possible for CI to semi-automatically analyze a large portion of the received samples, leaving only malware representative of new families (the novel ones) or developed by APTs (the most dangerous ones) to manual inspection. In our work, we design the architecture without any requirement related to the operating systems of the malware. However, the other contributions of this thesis are focused mainly on Portable Executable (PE) malware, the Windows executable file format. We decided to focus our research on them because, even in CIs, most infections start from Windows systems. In fact, Windows is *more vulnerable* than other operative systems for several reasons. First of all, it should be executable on a very wide set of hardware. Thus, due to this compatibility requirement, it is difficult to cover all the vulnerabilities. Moreover, thanks to economic reasons and being user-friendly, it is the most used operating system. Thus, operators that are not experts in computer topics, have usually Windows installed on their machines. Such employees are also the easiest vulnerabilities to exploit [19]. In fact, using various techniques, like social engineering and phishing, it is possible to *attack* them and infect their machines, starting the intrusion in the CI's system.

1.1 Contributions

Our contributions can be mainly organized in:

- Design and development of an architecture for a malware analysis framework to support CIs in such a challenging task. Given the huge number of new malware produced daily, the architecture is designed to automate the analysis to a large extent, leaving for human analysts only a small and manageable part of the whole effort. Such a non-automatic part of the analysis requires a range of expertise, usually contributed by more analysts. The architecture enables analysts to work collaboratively to improve the understanding of samples that demand deeper investigations (intra-CI collaboration). Furthermore, the architecture allows sharing partial and configurable views of the knowledge base with other interested CIs, to collectively obtain a more complete vision of

the cyber threat landscape (inter-CI collaboration). Chapter 3 describes it in details.

- Design of two datasets:
  - *MalFamGt* is a dataset for *family detection* and *identification*. It contains a set of malware grouped by family labels obtained with two different approaches, one based on the *elaboration* of anti-virus labels, the other based on malware behavior.
  - *dAPTaset* is a database that collects data related to APTs from existing public sources through a semi-automatic methodology and produces an exhaustive dataset. At the time of the publication was the first publicly available dataset that easily provides APT-related information (*e.g.*,binary hashes, networking data, etc.) to researchers involved in Advanced Persistent Threats studies

  Both datasets have been used to perform various studies and experiments. We publicly released them to allow other researchers to easily compare their methodologies and researchers to ours. Chapter 4 contains all the information on their structure and the methodology to build them.

- A novel solution for the identification of malware families that is based on the usage of the Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH) algorithm, an efficient clustering algorithm for very large datasets. Chapter 5 details the proposed methodology and also shows a comparison with other clustering algorithms, where it is made evident that BIRCH outperforms all the others, achieving an FMI greater than 95%.

- MalFamAware, a novel approach to *incremental automatic family identification and malware classification*, where samples are efficiently assigned to families as soon as they are fed to the algorithm, and the families (i.e. the clusters) are updated accordingly. We demonstrated how *online clustering* can be used to implement family identification and malware classification at the same time, cutting the cost of a periodically re-training of a classifier. Chapter 6 contains details of the approach, showing how MalFamAware achieve an F1-score greater than 92%.

- A *Triage Approach* to perform APT-related malware detection and identification in one step. We base this approach on static features that can be extracted rapidly and we focus on maximize the precision score in order to not waste analysts' time with *False Positive* cases. We design an initial approach that solves the problem and then we improved it with a more modular structure. Chapter 7 contains the explanation of the approach and the comparisons of both models, showing how we achieve a weighted accuracy and weighted precision score over 99% in both detection and identification.

- A heavy improvement of the *Triage Approach* based on a multi-step methodology to combine two different classifier models, each one dedicated to a single analysis phase. In Chapter 8 we will explain how this novel methodology works and show the improvements in detail.

Publications

1. G. Pitolli, G. Laurenza, L. Aniello, L. Querzoni "MalFamAware: Automatic Family Identification and Malware Classification Through Online Clustering", *International Journal of Information Security (IJIS)*, 2019, submitted.

2. G. Laurenza, R. Lazzeretti, L. Mazzotti "Malware triage for early identification of Advanced Persistent Threat activities", *Digital Threats: Research and Practice (DTRAP)*, preprint available at *arXiv:1810.07321*, submitted.

3. G. Laurenza, R. Lazzeretti, "dAPTaset: a Comprehensive Mapping of APT-related Data", accepted to *Security for Financial Critical Infrastructures and Services (FINSEC)*, 2019.

4. E. Petagna, G. Laurenza, C. Ciccotelli, L. Querzoni "Peel the onion: Recognition of Android apps behind the Tor", accepted to *International Conference on Information Security Practice and Experience (ISPEC)*, preprint available at arXiv:1901.04434, 2019.

5. G. Pitolli, L. Aniello, G. Laurenza, L. Querzoni, R. Baldoni "Malware Family Identification with BIRCH Clustering", accepted at *International Carnahan Conference on Security Technology (ICCST)*, 2017.

6. G. Laurenza, L. Aniello, R. Lazzeretti, R. Baldoni "Malware Triage Based on Static Features and Public APT Reports.", accepted at *International Conference on Cyber Security Cryptography and Machine Learning(CSCML)*, 2017.

7. G. Laurenza, D. Ucci, L. Aniello, R. Baldoni "An architecture for semi-automatic collaborative malware analysis for CIs.", accepted at *Reliability and Security Aspects for Critical Infrastructure protection (RESA4CI)*, 2016.

1.2 Outline

The remaining part of the Thesis is organized as follows: Chapter 2 shows the current state of the art. Chapter 3 presents our proposal for an analysis architecture. Chapter 4 contains the details of the datasets used in our works. Chapter 5 presents our methodology to perform *family detection* and *family identification*. Chapter 6 shows an improvement of the previous methodology in order to reduce time requirements. Chapter 7 presents our Triage approach to detect and identify APT-related malware. Chapter 8 shows how we improve the previous approach, outperforming the previous results. Finally, Chapter 9 presents our conclusions about the entire work.

Chapter 2

Related Works

In this chapter, we describe the current state of the art in Critical Infrastructures protection. This chapter is organized as follows: Section 2.1 shows the various frameworks developed to *measure* and manage CI security. Then, in Section 2.2 we describe the most representative researches on methodologies for analyzing malware. Section 2.3 details works related to malware family detection and identification. Finally, Section 2.4 presents the defenses proposed against Advanced Persistent Threats.

2.1 Frameworks for CI Protection

Due to the importance of protecting Critical Infrastructures, even countries and organizations develop their methodologies to attest the security level of their infrastructures. In this way, it is possible to understand and measure the grade of security of the various structures. Moreover, these frameworks usually contain methodologies and suggestions on how to fix the weakness found.

The U.S. *National Institute of Standard and Technologies* (NIST) is the first to design such framework [72]. They recognize the need of *helping organizations to better understand and improve their management of cybersecurity risk*[1]. This framework is created through the collaboration between American industries and their government. They develop this *voluntary guidance* that is based on the existing standards and contains guidelines and best practices to better manage and reduce the cybersecurity risk. It consists of three main components: the *Core*, *Implementation Tiers* and *Profiles*. The first one provides a set of desired cybersecurity activities using an easy to understand language. Implementation Tiers assist organizations by providing context on how an organization views cybersecurity risk management. Finally, the latter component is a set of profiles used to identify and prioritize opportunities for improving cybersecurity at an organization.

After NIST effort, many countries acknowledge the importance of such work. Thus, they decide to *translate* this framework in their own native language to spread the adoption in their national organizations. Two of the firsts countries were Japan[2]

[1] `https://www.nist.gov/cyberframework`
[2] `https://www.ipa.go.jp/files/000071204.pdf`

and Portugal[3]. Moreover, other countries find very useful the idea of a Cyber Security Framework adopted at the national level, but they realize that the American organizations' structures are too different from their ones. For this reason, they do not limit the effort to only translate NIST Framework, but they develop a complete adaption more suited for their needs. An example of such countries is Italy that, in a conjunctive work with Italian universities, presents its own national framework (*Framework Nazionale* [5]). The first version of this framework is very similar to the NIST original one, with a focus on large national organizations. After that, Italy proposes an enhanced version that takes into account that most of the Italian organizations are Small and Medium Enterprises (SMEs). SMEs require a completely different approach, thus the second version of the framework contains many suggestions on how to contextualize it in a smaller environment.

The common weakness of these approaches is that they do not cover specifically the malware aspect, which currently is the most used attack vector against CIs. Instead, in this thesis, we focus on improving CIs defenses against malicious software.

2.2 Malware Analysis Architectures

As explained before, malware are the preferred attack vector when dealing with Critical Infrastructures. Thus, several architectures for CI protection based on malicious software have been proposed.

An example is the one presented by Bayer *et al.* [7]. They propose TTAnalyze, a framework for automating the process of analyzing malware. Running a binary under TTAnalyze produces a detailed report about the purpose and the functionality of the analyzed sample. This report includes detailed data about modifications made to the Windows registry and the file system, information about interactions with the Windows Service Manager and other processes, as well as a complete log of all generated network traffic. TTAnalyze uses emulation to run the unknown binary. Unlike solutions that use virtual machines, debuggers, or API function hooking, the presence of TTAnalyze is practically invisible to malicious code. The analysis is comprehensive because their framework monitors calls to native kernel functions as well as calls to Windows API functions. It also provides support for the analysis of complex function call arguments that contain pointers to other objects. Moreover, this framework allows also *function call injection*, thus it is possible to alter the malware execution and run custom code in its context. Authors take advantage of the open-source emulator QEMU[4] to develop their custom analysis framework.

Another interesting malware analysis architecture is the one presented by Yin *et al.* [105]. They propose TEMU, a new generic dynamic binary analysis platform. Similarly to TTAnalyze, it is based on QEMU. This architecture allows analyzing malware running them in a fully virtual environment, and recording its behaviors in a fine-grained manner (*i.e.*,at instruction level). The registration is made completely from outside the environment, in order to not trigger any malware alert.

[3] `https://www.uschamber.com/sites/default/files/intl_nist_framework_portugese_finalfull_web.pdf`

[4] `https://www.qemu.org`

Moreover, TEMU includes a new technique, called *layered annotative execution*, for annotating certain memory locations or CPU registers or update existing annotations during the execution of each instruction in the emulated system, depending on the instruction semantics and the analysis purpose. Authors also developed a series of new techniques for detecting and analyzing various aspects of malware, released as TEMU plugins. The first one is Renovo [39], that extracts unpacked code and data. The second is Panorama [106], able to characterize abnormal information access and processing behavior of privacy-breaching malware. Another interesting technique is HookFinder [104], which has the goal of identifying and understanding malware's hooking. Last malware analysis technique presented in their work is MineSweeper [11], aiming to uncover hidden behaviors and identify trigger conditions.

Kong *et al.* [43] propose another interesting malware analysis system. They present a framework that automatically classifies malware instances according to their structural information, such as their function call graphs and basic block graphs. This framework extracts the function call graph from each malware program and collects various types of fine-grained features at the function level, such as which system calls are made and how many I/O read and write operations have been made in each function. Then it evaluates malware similarity iteratively applying two criteria: *discriminant distance metric learning* and *pairwise graph matching*. The first one consists of projecting the original feature space into a new one such that malware belonging to the same family are closer than others that formed different families. The latter, instead, consists of finding the right pairwise function-level matching between the function call graphs of two malware in order to measure their structural similarity. For the classification component, they rely on an individual classifier for each type of features, trained on the pairwise malware distances. However their framework is open to any classifier algorithm, the only requirement is a set of anchor instances, which are usually the subset of labeled samples in the original dataset. For example, for a kNN classifier, the anchor instance set can be the k closest malware from the test set. Instead, for an SVM classifier, the support vectors already contain all the anchor instances.

Nari *et al.* [68] propose a completely different approach. They present a framework for the automated classification of malware samples based on their network behavior. They abstract the network behavior of malware samples to high-level behavioral profiles that contain communication information including IP addresses, port numbers, and protocols as well as dependencies between network activities. Then they model these profiles as graphs and introduce a set of graph features that they found to be the most effective in classifying malware samples. They start the analysis from the PCAP file obtained by the malware execution in order to obtain network flows. They base their network flows extraction on port numbers and used protocols. From these flows, they generate malware graphs. In the graph generation, they also use data on dependencies between network flows in addition to flow information. Thus, for each malware sample, they create a graph that represents the network flows resulted from the execution of that sample and the dependencies between flows. From them, they extract various graph features such as the size of the graph, the out-degree of the root node which represents the number of independent flows, and the maximum out-degree which shows the maximum number of

dependent flows. With the feature vectors generated, they finally classify malware in families. In their work they test different classification algorithms, finding that J48, an implementation of Quinlan's C4.5 algorithm, is the one that performs better in this scenario.

AMAL [67] is a more complex system. It is a large-scale behavior-based solution for malware analysis and classification (both binary classification and clustering). It addresses many limitations and shortcomings of the existing academic and industrial systems. AMAL consists of two sub-systems, *AutoMal* and *MaLabel*. The first is based on many previous works in literature for characterizing malware samples by their memory, file system, registry, and network behavior artifacts. AutoMal is intended for a variety of users and malware types, thus it supports processing prioritization, multiple operating systems and format selection, run time variables and environment adjustment, among other options. Also, it comprises of several components, allowing it to scale horizontally for parallel processing of multiple samples at a time. Instead, MaLabel tries to address the shortcomings and limitations of the prior work in practical ways. For example, unlike previous works such as [43] and [105], MaLabel uses low-granularity behavior artifacts to better characterizing differences between variants of the same malware family. Moreover, MaLabel is also capable of binary classification and clustering. It incorporates several techniques and automatically chooses among the best of them to produce the best results. To do that, MaLabel relies on labels accurately made by expert malware analysts to train classifiers and assist in labeling clusters grouped in unsupervised learning. Unlike labeling (for training and validation) in the literature, which is subject to errors, analysts who are domain experts did labeling for MaLabel. In the authors' opinion, in this scenario human errors in their labeling are negligible. Thus AMAL workflow can be separated in AutoMal that provides tools to collect behavioral artifacts, while MaLabel uses those artifacts to create representative features and then, through supervised or unsupervised learning algorithms, divide malware samples into families similar in behavior.

All these works focus on the development of an automatic system for analyzing malware. Unluckily, this approach is not enough to protect CIs. They need a continuously updated Knowledge Base (KB) to face the most recent malware. Moreover, some advanced malware should be manually analyzed to be fully understood. Thus a more complete architecture, that includes modules to collect new malware and maintain the internal KB update, offering at the same time tools and suggestions to security experts for manual inspection is required to strongly improve Critical Infrastructures security.

2.3 Malware Family Detection and Identification

The problem of classifying malware in families has been extensively addressed in the literature, as widely reported by some recent surveys on the topic, such as those by Ye *et al.* [103] and Ucci *et al.* [95].

Mainly, two different approaches can be outlined. The first one is based on *Supervised Learning* and requires an initial ground truth to build a model. The latter, instead, is based on *Unsupervised Learning*. It is usually less accurate than

the other one, but it can work without any previous knowledge. There is also another possible approach, the hybrid one. However, the hybrid approach works on family identification and it usually adopts the hybrid concepts to enhance hyperparameters of unsupervised algorithms, instead of other advanced methodologies.

Researches based on supervised learning usually rely on classification algorithms with a model trained on a recognized ground truth which is often manually built. An interesting example of this approach is the work presented by Kawaguchi *et al.* [40]. They propose a new family identification method based on features extracted from APIs/system calls. They record all the system calls performed by samples in the first 90 seconds of executions. From them, they produce the feature vector to be fed to classification algorithms. They tested the goodness of their proposal identifying families with a decision tree, random forest, k-NN, and naive Bayes as classifiers. Although interesting, their proposal appears too limited for a real-world scenario. In fact, one widely used *obfuscation technique* employed by real-world malware is remaining *silent* at the beginning of the infection. Moreover, the dataset used is small and contains only six different families, a number that is not enough to confirm the efficacy of their solution.

An impressive idea is the one proposed by Dahl *et al.* [20]. They based their features on byte sequences and APIs/system calls. The main novelty is the heavy reduction of the original input space dimensionality, allowing a neural network to be trained on the high-dimensional input data. Their initial set of features extracted by their analysis engine is composed of 50 million possible elements. To reduce the input space to a reasonable set of features, they first apply features selection using *mutual information* [18], resulting in a new set of 179 thousand sparse binary features. It is more reasonable but still impracticable with neural networks. Thus they apply *random projections* [50][51] reducing the features set by 45 times, reaching a more usable set of 4000 elements. To show the good performances of neural networks, they provide a comparison with a logistic regression classifier. However, although there is a large improvement in the accuracy, the test error percentage remains too high to be used in a real-world scenario.

An alternative idea is proposed by Raff *et al.* [79]. They evaluated the *Normalized Compression Distance* (NCD), a measure used in a large number of domains to compare objects with varying feature types. Applying NCD in malware scenario can be really interesting since it can be used on raw bytes, requiring the use of little, if any, domain knowledge. The minimization of domain knowledge is a great advantage in malware classification. In fact, it can reduce the obvious dependability between malware changes over time and features extraction adaptation. NCD cannot be directly applied in this scenario due to some performance limitations. In fact, the nature of how compression algorithms work causes problems for large input sequences [8], making the computation time for the NCD not practicable. To solve this runtime issue, the authors design an alternative metric called *Lempel-Ziv Jaccard Distance* (LZJD). In practice, they found a way to simplify LZMA, the most accurate compression algorithms for NCD, and thus speeding up the entire distance measuring process. Then they apply k-NN classification algorithm to perform family identification. They compare the performance both in time and accuracy of NCD and LZJD, showing how the latter performs better in each test.

Ahmadi *et al.* [2], instead, propose a novel paradigm to group malware variants

into different families. In this work, authors give more importance to feature extraction and selection methods than classification algorithms. For each malware sample, they compute features on byte sequences, opcode, APIs/system calls, Windows registry operations, CPU register modifications, and PE file characteristics. Features are extracted from content and the structure of the sample, so the proposed method will even work on packed and obfuscated sample. They suggest to group features according to different characteristics of malware behavior, and their fusion is performed according to a per-class weighting paradigm. The aforementioned feature engineering method is experimented with the Microsoft Malware Challenge dataset and achieves a very high accuracy rate of 97%. However, the problem of this approach is that many features come from observing system calls and their *effects* on the operative systems and the CPU. Thus, malware that use obfuscation techniques to evade system calls observation can deceive a large set of features.

Islam *et al.* [36] also present a methodology to integrate static and dynamic features for family identification. In this work, they combine discoveries from their previous works in order to improve the final accuracy. Moreover, in this way they extract features in different ways, making the approach more robust to possible evasion techniques. First in [92] they demonstrate the effectiveness of using function lengths as a mean of malware classification. Then they propose the adoption of *printable string information* [91] and their combination with function lengths features [35] to distinguish among malware families. Both these classes of features are statically obtained through a disassembler. Another step in their study about features extraction for malware family analysis is their work on *dynamic log information* [93], although they initially used these features in a malware detection scenario. In this work they use a trace tool, called *HookMe*, to actively monitor the state changes in the system and generate a trace report which reflects the behaviors of the files in terms of API function calls. Last, they test features vector obtained with all the previous approach and the combined vector with multiple classification algorithms (SVM, random forest, gradient boosting and decision tree) confirming that the fused one performs better than all the others.

Another interesting work is proposed by Nataraj *et al.* [70]. In this paper, they follow a completely different and novel approach with respect to more *classical* ones to characterize and analyze malware. It is based on the idea that, at a broader level, a malware executable can be represented as a binary string of zeros and ones. Thus this vector can be reshaped into a matrix and viewed as a simple image. More in detail, they *read* any malware as a vector of 8-bit unsigned integers and then organized into a 2D array. This array can be visualized as a grayscale image in the range [0,255] (0: black, 255: white). The width of the image is fixed and the height is allowed to vary depending on the file size. Observing these images they found significant visual similarities in image texture for malware belonging to the same family, probably caused by the common practice of code reuse when generating malware variants. This visual similarity can so be exploited through techniques from computer vision, where image-based classification has been well studied, to perform malware family detection and identification. To compute image features, they use the *GIST* computation, typically used in the feature extraction from image textures [94][73]. The resulting representation is analyzed with k-nearest neighbors with Euclidean distance as classification algorithm.

Unsupervised learning, instead, has been adopted by other researchers that propose mainly works based on clustering algorithms. As explained before, the advantage of this approach is the possibility of working without previous knowledge. However, many times authors use part of the testing dataset to tune the algorithm and achieve better results, using a partial hybrid methodology.

An example is the work presented by Bailey *et al.* [4]. Their proposal is a methodology for family identification based on behavior similarity. They divide malware into groups of samples with similar behavior in order to reduce the number of unknown malware. A novelty of their approach is that they avoid the classical approach of recording individual system calls to build the features vectors. Instead, they focus their attention on *what the malware actually does* in order to obtain a more invariant e directly useful information. System calls, in fact, maybe at a level that is too low for semantically abstracting meaningful information. Authors define the behavior of malware in terms of non-transient state changes that the malware causes on the system. Observing state changes avoids many common obfuscation techniques that foil static analysis as well as low-level signatures. In particular, they extract simple descriptions of state changes from the raw event logs obtained from malware execution. For example, they record spawned process names, modified registry keys, modified file names, and network connection attempts. The list of such state changes becomes a behavioral profile of a malware sample. Then they apply a hierarchical clustering algorithm to detect and identify malware families. However, they need to properly choose the distance metrics to avoid the problems of *overemphasizing size* and *behavioral polymorphism*. The first one is related to the fact that, when the size of the number of behaviors is large, the edit distance is effectively equivalent to clustering based on the length of the feature set. Thus the algorithm will overemphasize differences over similarities. The second problem is related to the fact that many observed clusters had few *exact* matches for behaviors, due to simple behavioral polymorphism, like the use of random file names. To overcome these problems they rely on NCD, as the distance metric. Intuitively, NCD represents the overlap in information between two samples. As a result, behaviors that are similar, but not identical, are viewed as close (*e.g.*,two registry entries with different values, random file names in the same locations). Normalization addresses the issue of different information content.

Bayer *et al.* [6] proposed a similar work. Authors enrich and generalize the collected malware execution traces, then they summarize the behavior of a sample in a behavioral profile. These profiles express malware behavior in terms of operating system (OS) objects and OS operations. Moreover, profiles capture a more detailed view of network activity and how a malware program uses input from the environment. This allows their system to recognize similar behaviors among samples whose low-level traces appear very different. Their approach can be seen as an improvement of [4], previously described. In fact, they apply a similar methodology, but they also include *taint tracking* into the execution trace. Taint tracking consists of attaching labels to certain interesting bytes in memory and propagate these labels whenever they are copied or otherwise manipulated. This allows to better capture the behavior of a malware program. For example, they can determine that a file name depends on the current date and changes with every malware execution. As a result, the filename is generalized appropriately. Finally, they apply a hierarchical

clustering algorithm to profiles to detect malware families. To make their approach more scalable, they employ locality sensitive hashing (LSH), introduced by Indyk *et al.* [33].

Rather, Kinable and Kostakis [42] investigate call graph representation as a mean of abstraction of malware. A call graph models a binary executable as a directed graph whose vertices, representing the functions the executable is composed of, are interconnected through directed edges that symbolize function calls [82]. Then they apply a clustering algorithm to group malware in families. For this purpose, authors have defined a graph edit distance (GED) to group graph representations. They find that Simulated Annealing algorithm[44] is the best choice for this scenario. To verify the goodness of their proposal, they tested it with k-medoids and DBSCAN as clustering algorithms. k-medoids needs the number of clusters the algorithm should deliver as input, thus they rely on two quality metrics to analyze the natural number of clusters. These metrics are Sum of Error (SE)[90] and the Silhouette Coefficient (SC)[81]. SE measures the total amount of scatter in a cluster. SC is is a combination of cohesion, that expresses how similar the objects inside a cluster are, and separation, that reflects how distinct the clusters mutually are. At the end of the experiments, the authors determine that DBSCAN is the algorithm that performs better using their call graph representation as input.

Jang *et al.* [37], instead, propose a system for large-scale malware similarity analysis and clustering called BitShred. The key idea behind BitShred is to use feature hashing to dramatically reduce the high dimensional feature spaces that are common in malware analysis. In their work, they demonstrated that hashed features vector, called *malware fingerprint*, provide a near-optimal approximation of the true Jaccard index. Thus they can use Jaccard distance as a similarity metric among fingerprints. Using their optimized representation they make Jaccard a fast and CPU-friendly logic operation, speeding up clustering operations. The chosen clustering algorithm is the agglomerative hierarchical clustering, as it is impossible to know the number of clusters *a priori*. BitShred has also another great advantage, it is agnostic to the particular sample analysis system, even when the extracted feature set has a very large feature space. As confirmation, authors tested BitShred including it in two different previously proposed sample analysis approaches [6][1].

Hu *et al.* [30] proposed a novel way to extract static features that is less susceptible to mutation schemes frequently employed by malware writers to evade binary analysis. With their approach, it is possible to obtain simultaneously the advantage of a full-coverage provided by static analysis and the robustness guaranteed by dynamic analysis. Moreover, dynamic analysis is usually more resource-intensive than the static one, thus this approach can perform family detection and identification a lot quicker than other approaches. They called this system *MutantX-S*. It features a unique combination of techniques to address the weakness of static malware analysis. First, it tailors a generic unpacking technique to handle runtime packers without any previous knowledge of the particular packing algorithm. Second, it employs an efficient encoding mechanism, based on the IA32 instruction format, to encode a program into opcode sequences that are more robust to low-level mutations. Moreover, it applies a hashing-trick and a close-to-linear clustering algorithm in order to efficiently work with huge malware datasets with large features vectors. Rieck *et al.* [80] propose a system, called Malheur, for the automatic anal-

ysis of malware behavior using machine learning. As far as we know, it is the only open-source project publicly available. Malheur allows for automatically identifying novel classes of malware with similar behavior (clustering) and assigning unknown malware to these discovered classes (classification). They present an incremental approach for behavior-based analysis that relies on *prototypes*, i.e., reports being typical for a group of homogeneous behaviors. They use the hierarchical clustering with complete linkage, by processing the incoming reports of malware behavior in small chunks. To carry on an incremental analysis, there is the need to keep track of intermediate results and the concept of prototypes enables them to store discovered clusters in a concise representation.

The lack of reliable ground truth, that is also continuously updated, makes approaches based on classification algorithms insufficient to protect CIs. At the same time, widely used clustering algorithms have the disadvantages of requiring very high computational cost in order to perform their activities. Moreover, they need to be periodically executed to update the currently known clusters with the next batches of malware. In Chapter 5 and in Chapter 6, we will propose solutions to these weaknesses.

2.4 Defenses against Advanced Persistent Threats

As explained in Chapter 1, Critical Infrastructures main adversaries are Advanced Persistent Threats. Therefore, many works in literature try to improve CIs defenses against APTs. Some of them focus only on analyzing publicly disclosed APT cases but do not provide any useful countermeasures other than some general ones. For example, Virvilis *et al.* [99] present a technical analysis of the activities of Stuxnet, Duqu, Flame and Red October. They explain how Stuxnet operates against the Iranian nuclear program, more specifically Natanz uranium enrichment plant, as we already explained in Chapter 1. They also describe Duqu APT, showing how it was detected in 2011, but it was probably operating since the year before. Differently from Stuxnet, which aims to sabotage, the main goal of Duqu was espionage. Flame, instead, was first detected in 2012, but it is believed that it had been already active for 5-8 years. An interesting characteristic of Flame malware is that its size is over 20MB, something that is very uncommon for an APT-related malware that is usually less than 1MB in size. Interestingly, Flame malware is based on a minimalistic architecture, in fact, it expands its functionalities through the download of specific modules among a discovered set of 1000 different elements. The last APT presented in this work is Red October. It was discovered in 2012, but also it is probably active for 5 years before the detection. Like Duqu, its main target is espionage and its victims were diplomatic, governmental and scientific institutions. Authors conclude this report explaining that, due to the complexity of these threats, there is not any single product that can offer effective protection. Thus they add some technical countermeasures that are enough generic to improve defenses against all of them. However, these countermeasures, in practice, are classical operations to strengthen system security, like patch vulnerable systems and add strong access policies.

Other works instead explain in general how APTs operate, in order to suggest a methodology for dealing with them. For example, Jeun *et al.* [38] explain in detail

behavioral patterns of the attacks in order to design preventive countermeasures. They also state that this prevention should involve not only technical countermeasures, like physically separate Internet network from the internal network or patch vulnerable systems; it should also include managerial countermeasures, such as the improvement of security education of employees.

Other researches, instead, focus on detecting APT activities through *Anomaly Detection*. Marchetti *et al.* [60] propose a novel APT detection approach based on the analysis of network traffics. Classical traffic analyzers are able to detect common types of attacks, like a distributed denial of service. However, they are inadequate to identify APTs because this class of attackers usually mimics normal behavior and compromises a limited number of specific hosts, avoiding a typical automatic malware behavior of spreading infections. Their idea is to focus the analysis on the identification of internal hosts that are possibly involved in data exfiltrations as part of APTs, instead of identifying all the possible compromises hosts. The final output of their framework is an ordered list of hosts that performed suspicious network activities possibly related to APTs. Thus, security analysts can focus only on a small number of hosts, the top-k ones of the list. Unlike the majority of previous anomaly detection approaches, they focus on individual hosts and on comparative statistics to identify the hosts that perform suspicious activities, instead of network-wide statistics. From network flows, they extract only features that are relevant for data exfiltration, like the number of megabytes uploaded by internal hosts to external addresses, the number of flows (typically connections) to external hosts initiated by internal hosts. In this way, they can monitor a very complex network using only a small amount of features. Then they continuously observe the variation of these features over time to detect suspicious activities and rank consequently the hosts.

Similar work is presented by Su *et al.* [89]. They propose a general network security framework for APT detection. The main idea is to apply network data-flow to application data-flow reconstruction. They design this framework as composed of five main components. The first is a *network traffic redirection module*, dedicated to copy the actual network traffic and redirects it to the reconstruction module. The second component is the *user agent*, that has the goal of providing auxiliary information about the host to other components. This information can include some basic operating information such as whether the mouse is moving or the process name of the most top window. The third component is the *reconstruction module*. This component allows the framework to reconstruct the application flow according to the network flow together, also with a heavy reduction of the dependency from the host. The purpose of this module is to restore the data which may contain malicious contents. The fourth component is the *dynamic analysis module*. This one receives the reconstructed data and the auxiliary information from the previous components. With this information, this module can realize a virtual environment with the same real context present during the *packet exchange*. Thus the module can replicate what happened in the real host and find out if there was malicious behavior. In this way, it is also possible to detect unknown attacks, including the one that exploits zero-day vulnerabilities. For example, after the .doc document opened by WinWord.exe (process name of Microsoft Word), if a request to a remote address is launched, the framework should log these suspicious behaviors and send

them to the decision module. Last component is the *decision module*. This module integrates former information and gets a conclusion according to pre-defined criteria.

Marchetti *et al.* [61] propose AUSPEX, a framework for APT activity detection based on network analysis. The novelty of their methodology consists of multi-factor approaches where human specialists are supported by big data analytics techniques. These techniques should be able to detect and prioritize weak signals related to APT activities. The main purpose of AUSPEX is to prioritize hosts that are most likely compromised by an APT. To perform this task, AUSPEX gathers and combines *internal information* from local network probes, and *external information* from public sources such as the web, social networks, and blacklists. More in detail, AUSPEX gathers network logs that collected through SIEM and intrusion detection systems, in order to identify any weak signals possibly corresponding to APT activities. It also contains a simplified *assets list* that maps the members of the organization and their client devices, to link technological and open-source data. It gathers also OSINT information from public open sources, in order to identify and quantify the information that is available to APT attackers. Similarly to other works in this field, AUSPEX adopts a network-centric approach. They choose this one because network traffic can be collected and analyzed more easily than host-based logs. This is especially true in large and dynamic organizations like Critical Infrastructures, comprising heterogeneous hardware and software components.

Friedberg *et al.* [24] present an APT detection system based on an anomaly detection approach. The proposed approach is designed to extend common security mechanisms and especially *packet-level* IDS systems. Similar to some existing mechanisms, the proposed approach is based on the analysis of log files. However, it does not rely on existing knowledge. On the contrary, the approach constructs a model while processing the input. This system models through four information categories. The first is called *search-patterns*, it is based on the observation of random substrings of the processed lines. The second category is called *event classes* and it classifies log-lines using the set of known patterns. *hypothesis* is the third class of information. It describes possible implications between log-lines based on their classification. The last is *rules*. A rule is a proven hypothesis. This proof is given if the implication described by the hypothesis holds during evaluations. The system evaluates rules continuously in order to detect anomalous behaviors in the system. Thus the analysis of the system behavior relies only on the knowledge of the model, without any previous information.

Other researchers concentrate their effort on studying different attacks in order to find common patterns and widely used techniques. Then, they propose different countermeasures to strengthen the boundary of organizations. For example, Virvilis *et al.* [100], continuing their previous study [99], analyze malware developed by Stuxnet, Duqu, Flame, Red October, and APT29. Then they try to identify malware common characteristics and the potential reasons behind detection failure. They find that all of them target 32-bit versions of Windows, strangely none of the malware would execute on 64-bit systems. Duqu and Red October both used malicious Word and Excel documents for infecting their targets, while APT29 exploited AdobeâĂŹs PDF Reader. Stuxnet and Flame infection, instead, probably started through infected removable drives. All malware communicated over ports 80, 443 or 22, probably because traffic to those ports is frequently allowed also in

networks with strict access policies. Following their study of these specific APT-related malware, they design some countermeasures. Most of them are similar to general hardening policies described also in other works, like the ones presented at the beginning of this section. The novel suggestion is to increment the use of *Trusted Computing*. This approach has already known strong limitations[74], but the authors stated that, in a scenario with heavy security requirements, like Critical Infrastructures, the benefits introduced by a trusted computing significantly overcome the shortcomings.

Chent *et al.* [15] present the result of a detailed study of the APT phenomenon. Their main contributions are a taxonomy of phases, mechanisms, and countermeasures, both conventional and not-conventional ones. They propose to organize an APT Attack in six phases. The first phase is *reconnaissance and weaponization*. Here, attackers identify and study the targeted organization, collecting as much as information possible. Second phase is *delivery*. In this phase, attackers deliver their exploits to the victims, through directly or indirectly ways. *initial intrusion* is the third phase. It happens when the attacker obtains his first unauthorized access target systems. Next phase is *command and control*. In this phase, attackers establish a backdoor and start using Command and Control (C2) mechanisms. In this way, they take control of compromised computers and enable further exploitation of the network. Fifth phase is *lateral movement*. Once the communication is established, threat actors move inside the network in order to increase the number of controlled systems and discover and collect valuable data. The last phase of an APT attack is *data Exfiltration*. One of the main goals of APT attacks, in fact, is to steal sensitive data. After analysis of various attacks through their taxonomy, authors come up with a series of best practices and security countermeasures to apply for enforcing security.

Brewer *et al.* [10] propose a similar work. They organize the APT lifecycle in four phases. First there is the *reconnaissance* phase. The attacker needs to find an entry-point to the system. Much of the reconnaissance done against the APT's target is passive. Then there is the *compromise* phase. Once reconnaissance is completed, they now need to gain access to the target perimeter. This task usually is accomplished with the delivery of a custom malware via a spear-phishing campaign. The malware is usually built with a zero-day vulnerability exploit. Then the APT needs to *maintain access*. The most common way to do this is by stealing valid access credentials. Once access has been gained this way, the APT usually installs a Remote Access Trojan (RAT) on several hosts within the infrastructure. Next phase is the *lateral movement* phase. Once the attacker is in and is able to explore the infrastructure, the APT tries to identify where the target resides. Last, there is the *data exfiltration* phase. Here, an APT realizes its goal. It starts to identify, gather and move the target data outside the network. As stated by the authors, throughout this *journey*, APT leaves behind a trail of log data. Correlating these logs with information about the attack phases can help in determining if an attack is taking place. For each phase, it is possible to gather different evidences. For example, if a port scan is run against a public IP address from the same origin host within a small period, some sort of port scanning activity may be taking place. This scan can be part of a reconnaissance phase, thus defenders should automatically add the origin address to a *suspicious IP* watch list. This list can also be useful in the

following phases. If a host tries to connect to one of the suspicious IPs, it can be proved that there are APT activities in the systems, such as a RAT in the third phase. However, a huge disadvantage of this approach is that it requires an in-depth understanding of *normal* activities in the network, including all the information on the relations among the various events.

Lockheed Martin [62], instead, shows a slightly different way of tracking APTs activities. They propose a *Kill Chain* divided into seven phases, proposing also some operations that defenders can apply to mitigate risks of APTs attacks. First phase is *reconnaissance*. Here the attackers are in the planning phase of their operation. They conduct research to understand which targets enable them to meet their objectives. For defenders, detecting reconnaissance as soon as it happens can be very difficult, but its discovery, even after the attack, can reveal useful information about the goals of the adversaries. Second phase is *weaponization*. The adversaries are in the preparation phase, for example, they are developing tailored malware. This can be done automatically, with a toolkit that couples malware and exploits, or manually, depending on target defenses. Though defenders cannot detect weaponization as it happens, they can infer what is happened by analyzing malware artifacts. For example, they can conduct full malware analysis, focusing not only on the dropped payload but also on how it was made. In this way, they could find new campaigns and new payloads only because attackers reused a toolkit. Third phase is *delivery*. In this phase, the adversaries convey the malware to the target. They can fulfill this task through *controlled delivery*, like directly infecting a web server, or *released delivery*, like through malicious email. For defenders, this is the first and most important opportunity to stop the operation. The fourth phase is *exploitation*. Here attackers gain access to target systems, exploiting a vulnerability, that can be software, hardware, or even human, for example having an employee that opens the attachment of the maliciously crafted email. Next step is *installation*. Typically, adversaries install a persistent backdoor in the victim environment to maintain access for an extended period. The defender should use endpoint instrumentation to detect and log installation activity. Also, they can take advantage of malware analysis to study the installation phase and create new mitigations. *Command and control* follows installation. Here malware open a channel to enable attackers to remotely manipulate the victim. This phase is the defenders' last best chance to stop the operation. They should block the C2 channel, if malware does not receive a command, it cannot do any harmful operation. Last phase is *actions on objectives*. Here intruders accomplish the missionâĂŹs goal: collecting user credentials, collecting and exfiltrating data, destroying systems, and so on. Defenders must detect this stage as quickly as possible. In fact, the longer an adversary has access, the greater is the impact. They can detect the attack in various ways, for example by using forensic evidence.

Following the previous trends, Ussath *et al.* [98] develop a Security Investigation Framework (SIF). The main objective of the SIF is to support investigators with automatic and semi-automatic functions for analyzing and tracing multistage attacks. To achieve this, all relevant information obtained from the investigation must be collected within the SIF, in order to be able to understand all the details and steps of an attack. Common information sources are pre-filtered logs, results of forensic investigations and malware analysis reports. Within the SIF, the results of

the different investigation methods should be leveraged to reveal new relations and efficiently automate further steps of the investigation. Furthermore, the framework should allow an investigator to carry out additional analyses based on automatically gathered meta-information, threat intelligence or personal experiences.

Despite proposing very interesting and novel ideas, these works do not address the problem from the point of view of a Critical Infrastructure. Approaches that suggest countermeasures after the attacks can be useful, but they are still insufficient. In fact, a CI must keep working seamlessly, so previous proposals do not fit. Many other approaches, instead, focus on anomalies in the systems or the network, without taking into account the attack vector that is mostly used by APTs for their activities: the malware. In fact, all these methods evaluate data on suspicious software equal or even less important than the others. In our opinion, instead, one of the main cyber defense of CIs should be the one against malware. Thus there is the need for methodologies that are focused on the identification of them.

Chapter 3

Architecture for Semi-Automatic Collaborative Malware Analysis for Critical Infrastructures

A Critical Infrastructure (CI) is a complex system deployed over a (sometimes) large geographical territory that ensures a 24/7 service to citizens. Such system is obviously composed of a very high number of devices. Thus, due to its size and complexity, it can embed a huge number of vulnerabilities that could be exploited by attackers. Many of them are known to the CI, however it is sometimes impossible to patch them due to service interruption or because the removal of a vulnerability could take years (*e.g.*, replacement of millions of hackable smart meters already deployed). Hence, vulnerabilities and successful attacks are probably the most *sensitive information* of a CI. It is obviously harmful if an attacker knew vulnerabilities of the infrastructures. Moreover, knowing that a CI was attacked or that it has some unpatched vulnerabilities can heavily impact its reputation. As a result, fully relying on external *security providers*, such as anti-virus and intrusion detection systems companies, is not a good option for a CI. In this scenario, it must be taken into account that there are hidden policies made directly by *adversarial* states that can take advantage of information about vulnerabilities and suffered attacks. An example is the recent news about the U.S. government that decided to wipe software made by a prominent Russian cybersecurity firm[1] from their computer systems, despite is one of the most famous software companies of the field. Moreover, in some particular scenarios, where there are private companies involved as services providers, sharing too much information about suffered attacks can help competitors. In fact, an adversarial society can use information about the attacks to discover some *hidden information* about the composition or the workflow of the victims and this can even ruin their reputation.

Ideally, in case of an attack event, the targeted CI would need to keep the story as much confidential as possible, share the information with national authorities in

[1] `https://www.nytimes.com/2017/09/13/us/politics/kaspersky-lab-anti-virus-federal-government.html`

charge, carry out all the investigations required to shed light to what happened, and finally decide what information to disclose and how to communicate it to the public. Many countries are providing laws and regulations to manage what and how companies and CIs should share with government organizations. For example, the European Commission has recently released the EU Network and Information Security (NIS) directive[2]. As a consequence, CIs need to establish internal processes to manage their cyber issues, and a key element to achieving that consists of the capability of examining the behavior of malware samples.

CIs would need to keep an always updated knowledge base and awareness on novel malware samples. This can be obtained through a proactive approach and by establishing collaborations with third parties. The advantages of this method include the possibility to be as much prepared as possible against novel malware. An effective *malware analysis framework* should support this kind of approach, but the numbers related to new malware variants found daily are impressive, and their analysis would pose practical scale issues. Given the huge amount of samples to analyze, the analysis process should be automated as much as possible. Where automation is not possible, such as when an unknown form of malware is detected, human intervention by some internal analysts is required. To be even more effective, such manual analysis should be done collaboratively by experts having skills in diverse areas, to widen the coverage of the aspects to investigate and consequently enrich the knowledge base. As explained at the beginning of the chapter, this knowledge should be private to avoid the spread of information on vulnerabilities and weakness, and of industrial secrets. However, it should be also public to allow an easy enhancement through threat information exchanges. In practice, CIs should share more information as possible to allow others to take advantage of their suffered attacks and analysis results, but they would keep hidden some details that can be harmful if known by the wrong entities. Thus, a controlled form of information sharing is necessary with national/international agencies and other CIs and organizations facing similar cyber threats [57, 108].

In this chapter we describe an architecture [49] for a framework for CIs dedicated to malware analysis, which takes into account (i) the need of CIs for high confidentiality of their cybersecurity concerns, (ii) the capability of analyzing very large volumes of malware samples in a mostly automated way, while still enabling analysts to contribute, (iii) the possibility to account for *intra-CI collaboration* by allowing different human operators to add results to the same sample analysis, and (iv) the opportunity to open to *inter-CI collaboration* by sharing selected subsets of developed knowledge with other CIs. Although the architecture we designed is highly tailored to meet the *confidentiality* requirements typical of a CI, its field of application can be broader. Enterprises and corporations could surely benefit from such a solution, for example, adding our proposal in their *Computer Emergency Response Team* (CERT) toolset. They could implement our proposed architecture as the basic framework and integrate all their tools and analysis systems in order to feed a global knowledge base. Moreover, they can be part of the threat information sharing system that can enhance their knowledge base with external information that is already been made compatible.

[2] `https://www.enisa.europa.eu/topics/nis-directive`

3.1 Requirements

This section defines the main general requirements for the malware analysis framework, which guide the design of its architecture in the next section.

[R1] Distinct sources for malware samples

Being able to feed as many samples as possible is fundamental to get proper coverage of the latest threats. The framework shall allow input samples coming from a variety of sources, including known malware datasets, honeypots and malware crawlers.

[R2] Mostly automatic analysis

Expecting large amounts of malware to inspect, the analysis of input samples shall be carried out automatically, anyway allowing for intervention by analysts to enhance the quality and the depth of the examination. Samples deserving more thorough investigation shall be pointed out by the framework, with an indication about why such a deepening is advised and a rank on how much worthy this sample is to be further examined.

[R3] Intra-CI collaboration

Manual malware analysis is a required complement to automatic analysis in order to cope with the expected multitude of variants and unknown situations that are likely to be found. The framework shall support the cooperation among malware analysts of the same CI by allowing more authorized users to manually contribute to the analysis of a single specific sample.

[R4] Continuous development of the knowledge base

Since one of the main goals of the framework is to generate and maintain an updated knowledge base about cyber threats, a natural requirement regards the capability to continuously increment and refine all the models learned from malware analysis.

[R5] Private knowledge base

As explained at the beginning of the chapter, CIs have strong privacy requirements, especially about incidents. In fact, they try to hide details about their internal composition as much as possible, while, at the same time, they must comply with legal obligations. It should be preferable for CIs' security to not publicly disclose too much information about suffered attacks, especially when only partially successful. Opponents like attackers or adversarial companies can infer details about weaknesses or the internal configuration of the infrastructure.

[R6] Inter-CI collaboration

Even though the previous requirement seems to call for CIs to evolve their knowledge base in total isolation, collaboration among distinct organizations through data sharing is known to be beneficial for improving the cybersecurity awareness of all

participating organizations. All the *indicators of compromise* (IOCs) such as hashes and analysis result of received malware, or details on anomaly activities, such as domains and IP addresses, can be shared without any risks. Part of this data instead should be cleaned or anonymized before sharing, in order to not reveal too many details on internal composition, and workflows should be kept secret. For example, sharing all the activities of the malware can reveal information on the composition of the internal network of the victim. Then, this information can be used by an attacker to improve his next operation. Moreover, a CI should desire to share all the information with some trusted organizations, but only a small part of the incident details to others. For these reasons, we should apply both private or public requirements to the knowledge. Thus, the framework shall provide for a highly controlled export of partial views of the knowledge base to external CIs, according to well-defined policies, and in turn enable these CIs to deliver their own data share.

3.2 Architecture

In this section, we describe the architecture of the malware analysis framework. We first show how the sample analysis flow is arranged through a staged view of the architecture (Section 3.2.1). Then a high-level layered view of the architecture is presented (Section 3.2.2), to highlight main building blocks and interactions of the framework within CI's borders and with external CIs. The details about identified layers are then explained in Section 3.2.3.

3.2.1 Staged view and Data flows

We envision the flow of sample analysis as organized in a series of stages, from sample retrieval to information sharing. Figure 3.1 shows such a staged view of the architecture. The first stage is the *Loading stage*, where malware samples are gathered from distinct sources such as known datasets, malware crawlers and samples obtained thanks to honeypots. Once loaded, samples are stacked in an *Input stage* together with a set of metadata related to both the samples themselves and the way they have been retrieved. From the Input stage, malware can then be injected into the *Analysis stage*, which takes in charge all the kinds of automatic analyses to be performed. At a very general level, we can structure the Analysis stage as:

- a set of *analysis tools* that examine in details both the content and the sample behavior to produce comprehensive result datasets;
- a set of *classifiers* in charge of performing advanced elaborations, based on machine learning techniques, on such datasets;
- a set of *clustering Tools* responsible for grouping samples that share similar characteristics, such as behavior or malware family;
- a set of *correlators* that retrieve from trusted external sources fresh information about cyber threats, and try to match them with available data resulting from the analyses performed so far.

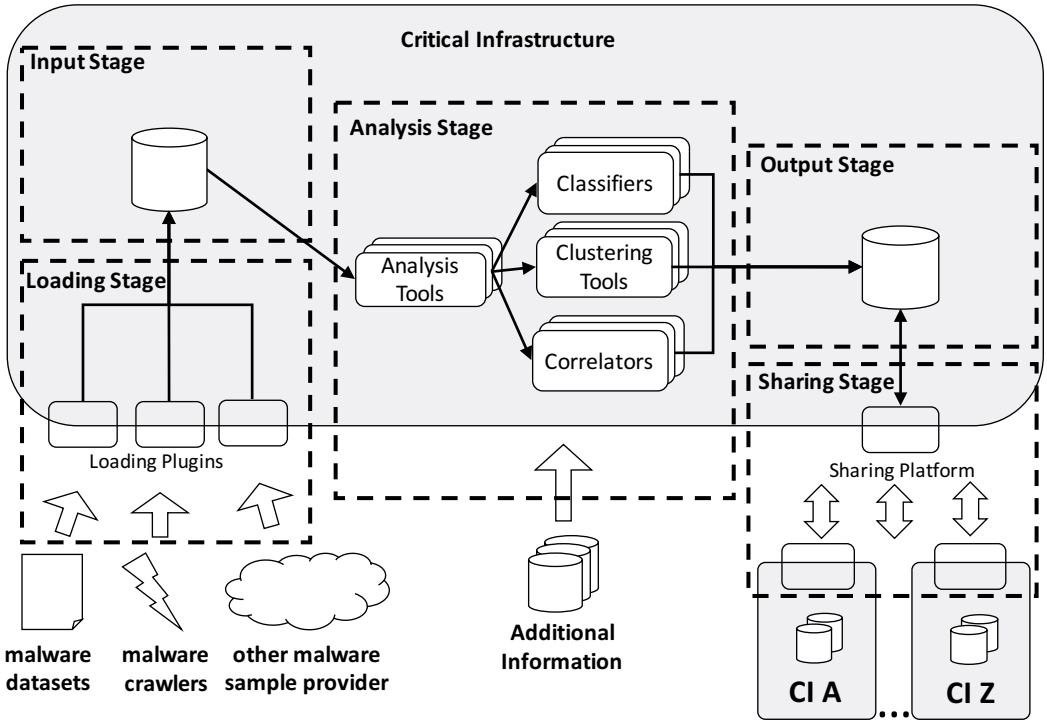


Figure 3.1. Staged view of the architecture of the malware analysis framework.

The results of the Analysis stage are then pushed to the *Output stage*, at disposal of authorized analysts for further analysis. More analysts can contribute to the same sample by providing additional information (intra-CI collaboration). Finally, a subset of the data contained in the Output stage, selected according to properly defined policies, can be moved to the *Sharing stage*. Data in this stage can be retrieved by external CIs for their own analyses, and the same CIs can also provide their own malware related data. In this way, each participating CI can benefit because, overall, the added value resulting from shared information exceeds the value of what every single CI contributed (inter-CI collaboration).

3.2.2 Layered View

Figure 3.2 presents an high-level layered view of the architecture, where all the stages introduced before are mapped to four layers.

Visual Analytics layer

Allows authorized users to monitor and control the running tasks within the framework employing several graphical user interfaces. Users with different roles can access different information and specific actions. At this stage of the design, we identify three main roles: *executives*, *security administrators*, and *malware analysts*. There is a generic partitioning that can be applied in many scenarios, however, CIs can clearly modify them to better fit their internal division, for example, they can

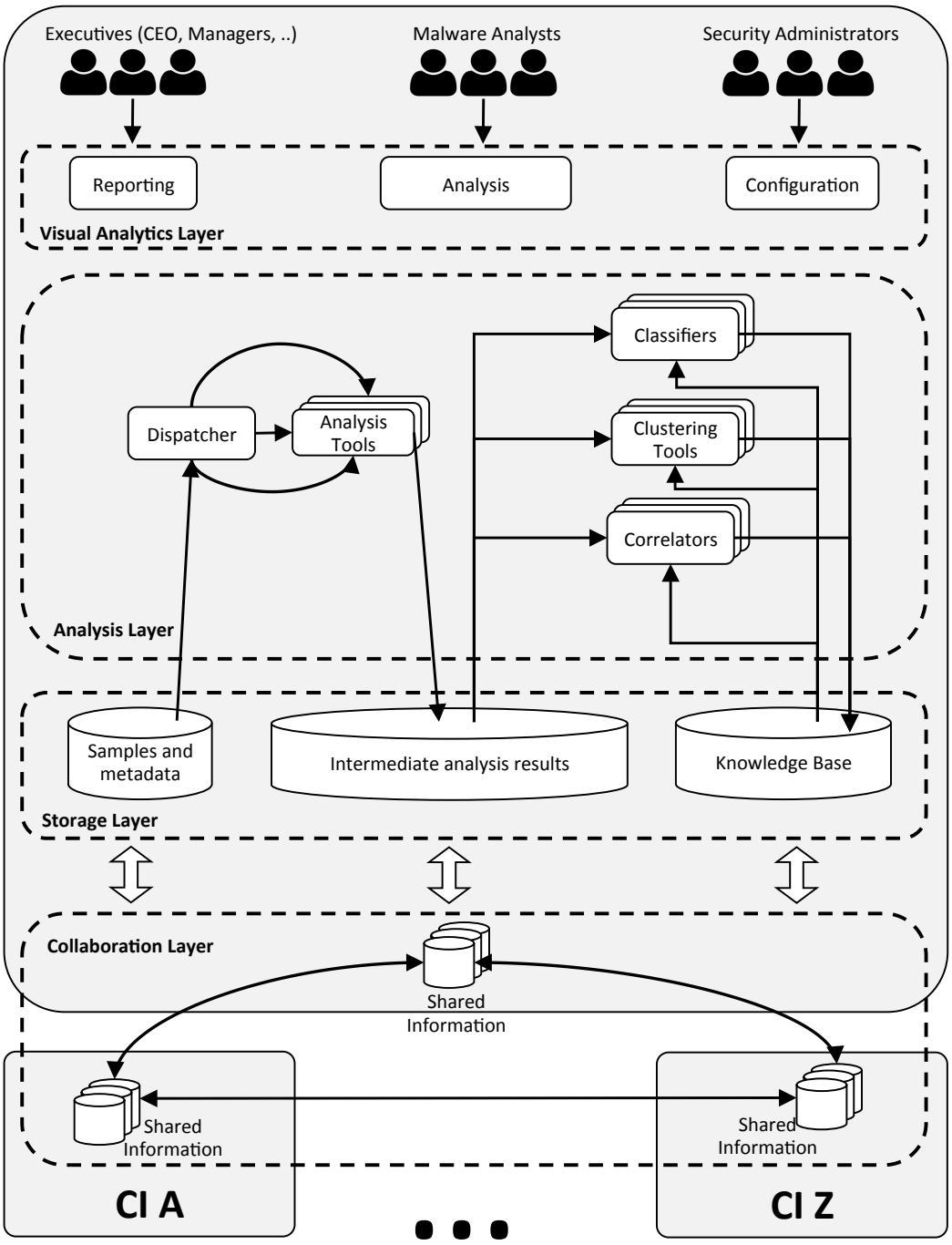


Figure 3.2. High-level layered view of the architecture of the malware analysis framework, where intra-CI (persons with different roles within the CI) and inter-CI (information sharing among different CIs) interactions are highlighted.

have a unique figure that acts as both security administrator and malware analyst. The following lists illustrate some of the *privileges* granted to each role:

Malware analyst

- Access to analysis results (on-going and completed)
- Manually operate many analysis tools
- Enhance information on samples (*e.g.*, adding manual analysis data)

Security Administrator

- Access to all the configurations of the architecture
- Access to system details views
- Modify right access of other roles
- Modify sharing policy

Executive

- Access to global reporting views
- Access to specialized reporting systems

Analysis layer

Contains all the components involved in the automatic elaboration of malware samples performed in the Analysis stage. It includes an initial *Dispatcher* in charge of sending samples to the right *Tools for Analysis*, depending for example on the file type or its source. The outputs of these tools are then provided to *classifiers*, *clustering tools*, and *correlators* by using the storage layer. The sequence of elaboration steps that each sample has to pass through is not static, it is instead defined according to the initial dispatching and possibly to some intermediate analysis results. After each elaboration step, the correspondent results are stored in the storage layer.

Storage layer

Includes all the partitions used for storing data: *(i)* one for the Input stage (*Samples and metadata*), which collects new samples to analyze and related metadata, *(ii)* one for maintaining intermediate data resulting from the Analysis stage (*Intermediate analysis results*), and *(iii)* one for the Output stage (*Knowledge Base*).

Collaboration layer

A view of data contained in the storage layer is extracted and used for information sharing with other external CIs (*inter-CI collaboration*). Regardless of the internal architecture used by other CI to produce their own knowledge base, we envision this layer as a cross border tier gluing together different CIs for the sake of controlled sharing of cyber threat knowledge.

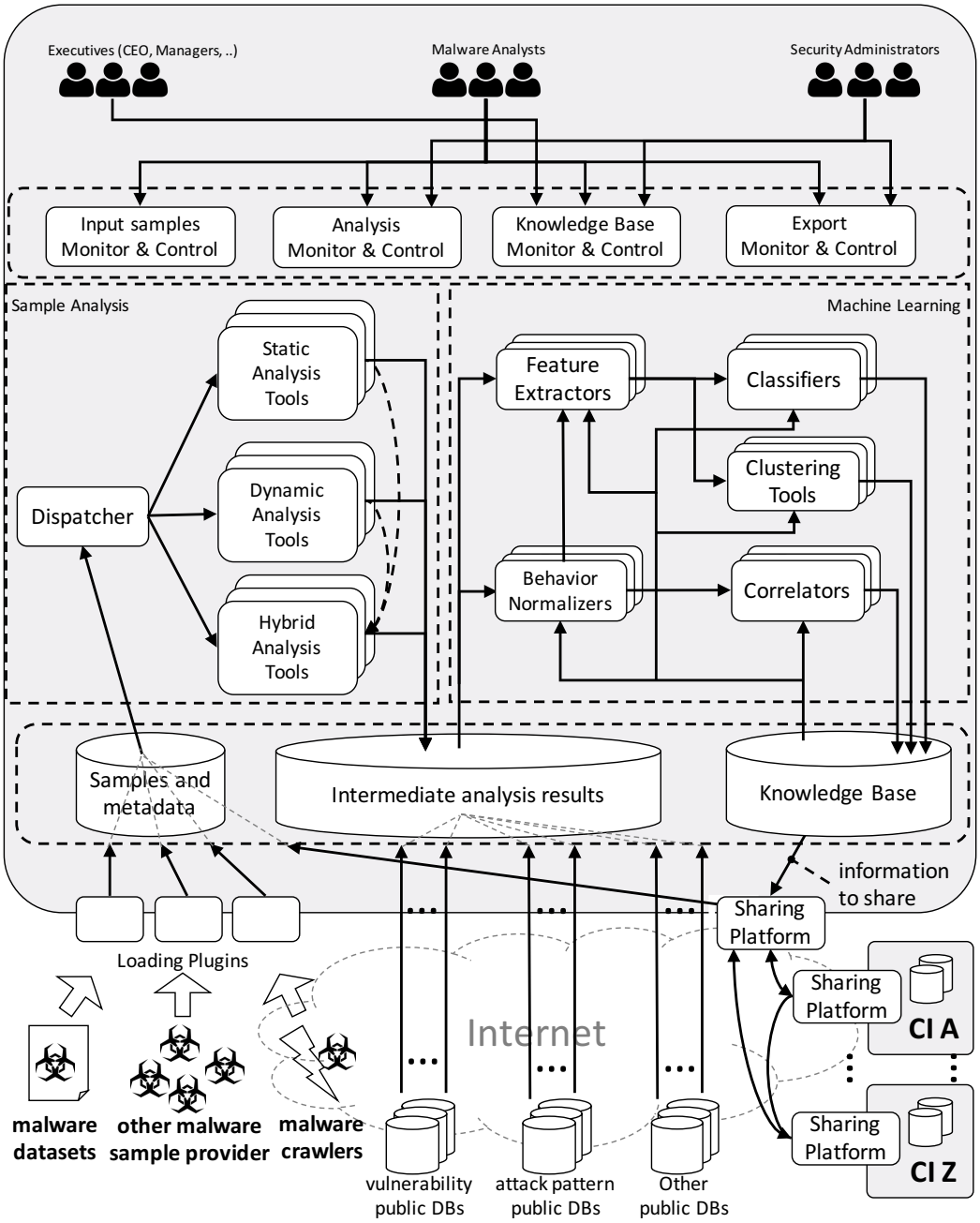


Figure 3.3. Detailed view of the architecture, where more particulars are presented related to each layer and the way it interacts with other layers, CI's authorized users, and the outside.

3.2.3 Layers Details

Each layer is described in more details, according to what is shown in Figure 3.3. Proposed tools for the implementation of the different layers are presented in Section 3.4.

Visual Analytics layer

The three functionalities of this layer (see Section 3.2.2 and Figure 3.2) are implemented through four *monitor & control* (M&C) components.

Input Samples monitor & control Allows analysts to load new samples employing the *loading plugins*, including the possibility to manually inject new samples. It also provides access to sample details contained in the Sample and metadata partition. Moreover, such layer implements part of the Analysis functionality of the visual analytics layer. It corresponds to *Loading and Input stages*, it provides to users an overview of them with the possibility of manual intervention, for example, an analyst may wish to submit a sample that might be immediately analyzed.

Analysis monitor & control Corresponds to the analysis stage, enables users to interact with components in the analysis layer and, hence, contribute to the analysis process. It offers to users various information about current analysis and the possibility of manually using the different tools. This is designed to allow users to execute the *not automatic* part of the analysis. In fact, with these interfaces, analysts can directly perform single tool analysis or also help the automatic workflow. For example, if a malware analyst finds the analysis of a sample stuck at a certain phase, he can access a manual disassembler, inspect the file and find why other tools failed. Then it can add some *special settings* to the analysis environment, like a particular configuration of the virtual machine used by the sandbox software, in order to allow the fulfillment of the entire analysis process. Moreover, it permits to analysts to import external unstructured reports in order to add supplementary information, for example, he may want to add a new report produced by a cybersecurity company that contains a list of IPs of a particular C&C. Then, with this information, an analyst can search for other samples that try to connect to the same destination and inspect them to find new correlations.

Knowledge base monitor & control Provides an interface for accessing results computed by the analysis stage. This layer is included in the output stage and supports authorized users by presenting them a *reporting interface*, which contains accurate statistical information.

Export monitor & control As part of the sharing stage, this layer allows security administrators and malware analysts to, respectively, define and implement policies for exporting and sharing partial views of the knowledge base to external CIs. Policy implementations and definitions can be specified through proper *configuration interfaces*.

Analysis layer

Consists of all the components involved in sample analyses and comprises the analysis Stage. This layer is further organized in two sub-layers: the *sample analysis*, with tools employing static, dynamic, hybrid analyses, and *machine learning*, which contains components able to leverage machine learning techniques to extract new useful sample-related information. Such information are then collected and stored in the knowledge base partition. The components of these two sub-layers are, then, described in the following.

Dispatcher Automatically retrieves not analyzed samples from the samples and metadata partition and dispatches them to the proper analysis tool, according to sample characteristics (i.e., a 32-bit portable executable will be forwarded to a dynamic analysis tool able to execute it).

Static, dynamic, and hybrid tools Receive samples from the dispatcher and perform different types of analyses. While dynamic tools require execution of the input samples, static tools only look at their content. Hybrid tools run analyses based on multiple information, such as the behavior of the samples and their disassembled code and, in addition, can leverage available metadata. These tools can be commercial products, open-source software or directly developed by the CI itself. Results are stored into the intermediate results partition or further used as input to hybrid analysis tools.

Feature extractors and behavior normalizers As described in the previous paragraphs, the results of the analysis tools are stored into the intermediate results partition, along with updated cybersecurity information coming from trusted external sources. Features extractors draw values of specific features from the results produced by the analysis tools. The sets of features to extract depend on the downstream classifiers and clustering tools to feed. Behavior normalizers are meant to prepare the input for the correlators by converting to a common format all the data related to sample execution. These data come from both dynamic and static analysis tools and trusted external sources. In general, extractors and normalizers can be also fed with models and profiles stored in the knowledge base.

Classifiers and clustering tools Classifiers assign samples to predefined learned classes, also defined in the knowledge base, on the base of features values. Clustering tools group samples according to their similarity, which enables to isolate specific families of malware and link unknown samples to such families. Similarities can be found in their behavior or because they share observable characteristics with specific families of malware. Each classification and clustering tool leverages a different machine learning technique. Results obtained by these tools are saved into the knowledge base and made available for further investigations, including a *rank* regarding whether samples are noteworthy and deserve special attention by analysts. Such a *rank* depends on the classification confidence level (the lower the confidence, the higher the attention it requires) and on the extent a sample is marked as an outlier by clustering algorithms. Samples classified with a low confidence level

and/or detected as outliers by clustering algorithms might represent noteworthy samples deserving the attention of malware analysts. Since the knowledge base is continuously updated, classifiers need either to be retrained periodically or to be designed so as to enable online learning. In case of retraining, the time required for each training represents a key aspect and plays a fundamental role in the choice of the employed classification tools.

Correlators They take as input the result of the behavior normalizer and perform their analysis in order to correlate different information. These data are obtained from the analysis tools and various trusted external DBs and the correlation consists of find any affinity between them. The resulting degree of similarities is a valuable result for understanding the nature of a sample. On the other hand, the lack of correlation between information provided by external sources and analyzed samples triggers the need for further analysis by an analyst. Depending on the detected correlation degree, a *rank* is associated with each sample to quantify how much worth it is to be analyzed in more depth.

For example, the sample may have a behavior that is similar to a common attack pattern of a group of malware. Likewise, other components, all results generated by these ones are stored in the knowledge base. While there exist several commercial and open-source software that can be used for the tools of the sample analysis sub-layer, this is not true for the machine learning sub-layer, also because of the greater complexity in the structure of inputs to be provided. Thus such software is more likely to be developed ad hoc, possibly by leveraging existing Machine Learning algorithms implementations.

Storage layer

As mentioned before, three distinct storage partitions have been identified.

Samples and metadata partition Stores new samples coming from the Loading plugins. Since samples are collected from distinct sources, they could have different representation formats. Loading plugins handle the format normalization of samples and their related metadata. The following list contains some examples of useful metadata.

- **hashes**: it contains the hashes of the sample and it is used as a unique key for the DB;
- **sources**: it contains all the places where the sample came from, it is a list because the same sample can be obtained from different sources;
- **status**: it obviously represents the current status of the malware in the analysis layer; possible values are *not analyzed*, *running* and *analyzed*;
- **toolchain version**: it contains the value of the version of the architecture at the moment of the analysis; it is a piece of useful information because with it is possible to easily recognize if current reports are old. It is empty if the status is *not analyzed*;

- **shared version**: it contains the version of the toolchain at the moment of the loading in the collaborative layer; it is empty if the sample is not currently shared

Intermediate analysis results partition Each elaboration step in the Analysis stage produces an intermediate result that is saved into this partition. This latter stores also additional information fetched from the Internet, such as updated publicly known vulnerabilities and attack patterns. In this way, further analyses may leverage intermediate results, obtained in previous elaboration steps, and publicly available security information.

Knowledge base When the analysis of a sample completes, all the related results and outputs are consolidated in this partition. A partial view of such data is periodically exported to an information-sharing subsystem to enable inter-CI collaboration.

Collaboration layer

As discussed in Section 3.1, the proposed framework aims to share, under well-defined policies, partial views of the knowledge base to support inter-CI collaboration. This layer is composed of all the collaborative environment participants' sharing platforms.

Sharing platform Contains the subset of the knowledge base that a CI wants to share with other Critical Infrastructures. The platform should enforce the adoption of policies preventing sensitive information disclosures. It is clear that sharing details of malware behaviors can help others CIs, like the simple example of the famous *kill switch* of the first WannaCry version[3]. However, disclosing detailed information on *tailored* malware can have dangerous consequences, for example showing to many adversaries some unknown weaknesses of its own system before having the possibility of fixing them. For these reasons, we design our architecture in order to have a separate platform that is dedicated to data sharing.

Instead of developing custom applications, storage and collaboration layers can be realized by using either commercial products and open-source software. Furthermore, they are more likely to provide interoperability guarantees that would allow meeting inter-CI collaboration requirements more easily.

3.3 Requirements Satisfaction

Requirement **R1** is satisfied by the plugin-based design of the *Loading stage*, which allows to include additional sample sources over time by developing the related plugins.

The need for automation reported in requirement **R2** is satisfied by the design of the *Analysis stage*, where all the elaboration steps, from sample retrieval to results

[3] `https://www.malwaretech.com/2017/05/how-to-accidentally-stop-a-global-cyber-attacks.html`

storage, are executed in the pipeline without human intervention. For what concerns the other aspects highlighted by requirement **R2**, *classifiers*, *clustering tools* and *correlators* provide a rank to signal samples that require further investigation, and the *Analysis M&C* component in the *Visual Analytics layer* enables malware analysts to manually contribute to the analysis (intra-CI collaboration).

Such component also enables to meet requirement **R3**, since it allows more analysts to collaborate among each other by contributing analysis results for the same malware.

As more samples are analyzed, collected results lead to a continuous enrichment of the knowledge base. Also, the models used in the analysis layer for classification and clustering are kept updated by employing periodic retraining and online learning techniques. This accounts for requirement **R4**.

The need for keeping the knowledge base as much confidential as possible (**R5**) is satisfied by providing full control on all the interfaces with the outside. Loading plugins can be designed and implemented by embedding anonymization techniques to hide the identity of who is collecting samples. The retrieval of additional public data from the Internet, such as known vulnerabilities, does not pose any further issue. What is shared with other CIs can be highly configured by security administrators using the *export* component in the *visual analytics layer*.

Finally, the need for inter-CI collaboration (**R6**) is fulfilled by the *collaboration layer*.

3.4 Technologies

We developed and deployed a working prototype of this architecture in a real environment. We deeply analyze different tools, finding interesting products with both open and commercial licenses, especially for the monitoring part. For the prototype, we prefer to use open software to be free to easily adapt it to our necessities, in the future we may decide to rely also on commercial components.

Visual Analytics layer This layer provides multiple interfaces for monitoring and controlling the architecture to users of different *level*, so we need a flexible way to represents data that can be usable both for simple overview and deep control depending on the current user. For this reason, we decide to implement a custom web interface that takes advantage of APIs that will be provided by products used in the toolchain.

Analysis layer Since we need to coordinate a huge number of different tools for sample analysis and machine learning analysis we decided to use a *Work-flow Manager*, in this way we can consider each tool as a step of a work-flow and thus we can easily concatenate/parallelize step, reuse step in different work-flows, and change the flow of the processes basing on output of single steps. To perform this job we choose *FireWorks*[4]. It is an open-source work-flow manager that provides flexible python APIs and various facilities for easy management; for example, it is possible to detect failed jobs and rerun them, remotely track the status of output

[4] `https://materialsproject.github.io/fireworks/`

files during execution and deploy for running high-throughput computations at large computing centers.

For the initial prototype, we developed a simple analysis layer with tools for Static Analysis and Dynamic Analysis. We opted for *PEFrame*[5] and *Cuckoo Sandbox*[6] for, respectively, static and dynamic analysis. The first is an open-source tool that performs static analysis of *Portable Executable* malware and generic suspicious file. The Portable Executable format contains the information necessary for the Windows Os loader to manage the wrapped executable code. It consists of the MS-DOS stub, the PE file header, and the sections, and can provide an enormous amount of features containing relevant information for malware analysis. Cuckoo Sandbox, instead, is the most famous open-source sandbox; it is a tool for execute malware in an isolated environment and register all the activities of the binary, like API called or its general behavior or also dump its network traffic.

The machine learning section was not present at the beginning of the prototype, but in the following chapters, we will show different works designed and developed to be part of it.

Storage layer Our purpose is to store information from different tools and sources, so we expect to collect data structured in different ways or even without a proper structure. For this reason, we chose to use a NoSQL database. Considered also the need to account for several expected changes during the prototyping of the framework regarding the way information should be organized, we also need high flexibility in the way data schemas have to be defined. We opted for MongoDB because it is document-oriented and allows dynamic schemas through the use of a JSON-like format.

Collaboration layer For the collaboration layer, we need something that allows complete control on what information can be shared and that can be used by different organizations without problems of data structures, for this reason, we decide to adopt *Malware Information Sharing Platform* (MISP)[7]. MISP is an open-source software solution for collecting, storing, distributing and sharing information about malware analysis, it has the advantages of being a widely used platform (it is used by organizations like NATO and Belgian Defense), so it can offer complete documentation and many useful features requested by the community.

[5] `https://github.com/guelfoweb/peframe`
[6] `https://www.cuckoosandbox.org/`
[7] `http://www.misp-project.org/`

Chapter 4

Proposed Malware Datasets

One issue highlighted in our works is the lack of publicly available datasets. Many research areas include some commonly recognized *ground truths* and *experimental datasets* that researchers can use as testbeds for their publications. For example for *Image Recognition* or *Text Analysis* several datasets are available for comparisons. In the Cyber-Security field, due to the rapid evolution and diffusion of malware, is it difficult to obtain a reliable and labeled ground truth.

At the beginning of my Ph.D. studies, we focused our attention on the problem of identifying malware families. Therefore, we looked for a dataset of samples labeled with their family name. To the best of our knowledge, in the case of malware families, there is only a work proposing a reference ground truth [97], where a manually labeled dataset with 85 samples is described. Dell SecureWorks Counter Threat Unit Threat Intelligence analyzes this dataset [34], founding that it is composed only of command-line transfer and traffic proxy tools. So it cannot be used as ground truth because it contains only a special class of malware.

We have hence decided to build a labeled dataset ourselves. Common approaches used in literature to produce a malware family ground truth consists of either labeling samples manually, or using labels provided by a specific anti-virus, or combining together labels of different anti-viruses. After some studies and trials, we come out with MalFamGT, a *Mal*ware *Fam*ilies *G*round *T*ruth[1], containing a set of labeled samples. We build it, collecting various malware from the wild, to obtain a sharable dataset that can be used for tests and comparisons. As explained before, to reduce the effort of security experts there is the need of some automatic or semi-automatic solution, thus a way to test it (or a part of it, if it is something similar to the architecture we proposed in Chapter 3) can be helpful to a complex structure like Critical Infrastructures. We will describe it in details in Section 4.1.

Later, in the Ph.D. course, we focused our research on the detection and identification of Advanced Persistent Threats, one of the worse adversaries for Critical Infrastructures. As far as we know, before our proposal, no public dataset presents structured and complete information about APTs. The cause of this lack is mainly related to the fact that many publicly available reports are made by Security Companies with different naming systems and text structures, making it impossible to

[1]The list with all the malware hashes can be found `http://users.diag.uniroma1.it/aniello/malware_dataset/malware_dataset_md5_list.txt`

easily merge the information, with no standard presentation. To solve this problem we make available dAPTaset, a dataset containing various structured information on APT, with also the methodology to build and maintain updated it using data from different public sources. Section 4.2 will describe in detail the structure of the database and the procedures to fill and update it.

4.1 MalFamGT: Malware Families Ground Truth

As stated at the beginning of the Chapter, common approaches to produce a malware family ground truth consists of samples labeled:

- manually
- using labels by specific anti-virus
- combining together labels from different anti-viruses

Some works use datasets where samples are manually labeled [66]. Manual labeling is a really time-consuming activity that requires highly skilled human resources, so commonly the size of these datasets is quite small. It thus also results difficult to update an existing dataset given the effort it requires and having at disposal a ground truth with novel samples is fundamental in the field of malware analysis. Other works use labels given by a single anti-virus vendor [37, 102], which presents two issues. First, achievable accuracy is strictly constrained by the quality of the labels provided by that vendor. Second, it is not possible to include malware that have not been labeled by that vendor. Combining labels provided by different anti-viruses can address both the issues, but brings an additional problem due to the well-known inconsistencies among different vendors [4, 65, 29]. Indeed, anti-viruses can disagree both on the maliciousness of a sample and the assigned label, in fact, each product can give a different name for the same malware or can use its own label format or also can assign malware to different groups. Another problem is that the labels of the same vendor may change over time. Exist some naming conventions as MAEC[2], but they are not widely used. For this reason, the researchers try to resolve the inconsistencies by translating names used across various vendors. However, sometimes such translation is impossible like when different families have the same name in different anti-virus products. Several studies exist that describe this issue and propose solutions. AVclass [85] is an automatic labeling tool that, given the labels provided by distinct anti-viruses, outputs the most frequent family name for each sample. Bayer *et al.* [6] scan over 14000 malware with six different anti-viruses, then select only those samples for which the majority of the anti-viruses reported the same malware family. This required them to define a mapping between the different labels that are used by different anti-virus products. The resulting dataset has 2658 samples, which is much smaller than the initial one. We notice that the most usual and accepted approach to generate a ground truth for malware families consists of using the labels of diverse anti-viruses, and choosing the most frequent label in case of disagreements. In [52], P. Li *et al.*conjectured that one factor contributing to the strong positive results obtained by Bayer *et al.*might be that these

[2] `http://maec.mitre.org/`

Table 4.1. Clustering quality of Malheur and AVCLASS.

Tool	SC	CHI
Malheur	0.556	1562.94
AVCLASS	-0.212	22.17

2,658 instances are simply easy to classify. In particular, they tried to execute the same clustering algorithm used in [6] on another dataset, for which four anti-virus tools consistently labeled each member, obtaining poor results. They noticed that the original dataset, where the algorithm performed well, is dominated by two large families, while the second dataset is more evenly distributed among many families.

Anyway, also using labels generated by different anti-viruses has issues, indeed in [65] the authors argue they are incomplete, inconsistent, and incorrect. Hence, to provide a fairer comparison on malware family identification accuracy we propose and use two distinct ground truths. One of them combines labels from the anti-viruses available on VirusTotal [3], the second one use the group labels proposed by Malheur[4], a tool already introduced in Section 2.3.

To show the goodness of these two ground truths, we also evaluate them by using two introspective metrics that do not depend on any external reference source: Silhouette coefficient (SC) [81] and Calinski-Harabasz index (CHI) [45]. For both metrics, a higher value indicates better quality. For SC the best value is 1 and the worst is -1. Results are reported in Table 4.1, where it emerges that the ground truth based on Malheur has better scores. Rather than proving that one ground truth is better than the other, this assessment challenges today's common approach of building ground truth solely based on anti-viruses labels and pushes to investigate and evaluate alternative methods.

4.1.1 MalFamGT Statistics

Our reference dataset is formed by a total of 5601 samples collected from the wild. We executed for some months *Maltrieve*[5], an open-source tool that gets malware directly from some feeders. Maltrieve collects information and sometimes the binaries of malware discovered by these security websites, without regards for any characteristics. From all this data, we selected all the Windows malware that can be analyzed in a sandbox without any external file or configuration in order to easily analyze them. We rely on Maltrieve to collect new malware to focus our attention on active software which is still being used, although some binaries were released years before. We considered other sources, but most of them, such as VirusTotal, do not allow us to freely download binaries. Other sources, such as *Malshare*[6], provide blocks of malware but, as stated at the beginning of this section, we prefer to focus on malware that are active at the time of the collection.

AVclass identifies 1321 families, and 887 of them are singletons, i.e. families with exactly one sample. Likewise Rieck *et al.* [80], we discard families with less than

[3] `https://www.virustotal.com`
[4] `https://github.com/rieck/malheur`
[5] `https://github.com/krmaxwell/maltrieve`
[6] `https://malshare.com`

10 samples. The resulting dataset contains 43 families and 3910 samples. Table 4.2 reports the distribution of samples over main families.

We evaluate the anti-virus-based ground truth quality by using the metrics defined in [31], the results are reported in Table 4.3. All the metrics have a value between 0 and 1. The *equiponderance* metric measures how much balanced are the contributions of each anti-virus. When it is close to zero, the ground truth is dominated by extreme cases, thus higher values are better. The *exclusivity* metric is the proportion of samples that were included only due to one anti-virus engine; lower values are better. The *recognition* metric represents the average number of positive detections. The *synchronicity* metric is the average similarity between pairs of anti-virus engines, so higher values are better. The *uniformity* metric shows how much balanced are the families. The value is close to zero when each sample is assigned a unique label by each anti-virus, it is one when the same label is assigned to every sample by all the anti-viruses. Once the distribution of labels has been extracted from the dataset, we can also measure how often labels are reused by anti-viruses. This property is measured by the *genericity* metric, which is the ratio between the number of distinct labels and the number of positive detections. The *divergence* metric measures the average proportion of distinct labels per sample, i.e., it represents to what extent anti-viruses provide for each sample a label that is inconsistent with the other anti-viruses labels. Lower values are better. The *consensuality* metric indicates to what extent we can rely on the most frequently assigned label for each sample. High consensuality values mean that used anti-viruses agree on most samples. Finally, the *resemblance* metric is an estimation of the global resemblance among the labels assigned to each sample.

4.2 dAPTaset

As introduced at the beginning of this Chapter, during the second part of my Ph.D. studies, we focused on the analysis of Advanced Persistent Threat. We have found the lack of a common database for comparison of academic researches and, to solve this problem, we have designed and developed dAPTaset [47], a database that collects data related to APTs through a methodology that semi-automatically gathers information from different public sources, merges them and produces an exhaustive dataset. As detailed in Chapter 1 differently from common threats, Advanced Persistent Threats usually attack a specific target and, as shown in several reports, Critical Infrastructures are in their top targets, for example, *APT30*[7] targeted government structures of some Asian countries like China and Vietnam or the famous *Stuxnet*[8] that targeted the Iranian nuclear facilities. Typically, an APT carries on cyber campaigns with the goal of exfiltrating sensitive and secret information from the victims or compromising the target infrastructure. To reach their goals, APTs follow a specific attack path [16, 32] involving several steps, such as social intelligence to identify easy victims inside the target, malicious files delivering, data exfiltration, etc. Hence, several cybersecurity companies are carrying on several analyses on APT activities and releasing public reports that can be leveraged to

[7] https://securelist.com/the-naikon-apt/69953/

[8] https://www.langner.com/wp-content/uploads/2017/03/to-kill-a-centrifuge.pdf

Table 4.2. Distribution of samples over main families.

Family Name	Samples
allaple	43
amonetize	19
beebone	10
bifrose	10
cosmu	67
crossrider	10
delf	17
domaiq	48
downloadguide	59
expiro	23
fareit	24
firseria	26
flystudio	292
gamarue	29
gator	22
hotbar	32
ibryte	15
installcore	25
installerex	47
kykymber	15
linkury	15
loadmoney	24
locky	11
mira	36
multiplug	71
nsismod	15
onlinegames	24
opencandy	13
parite	22
phishbank	1298
ramnit	817
sality	15
scar	10
soft32downloader	14
somoto	61
sytro	283
temonde	17
virut	34
vobfus	118
wapomi	10
weiduan	11
yantai	119
zbot	39

Table 4.3. Evaluation of anti-virus-based ground truth.

Metric	Value
Equiponderance	0.750
Uniformity	0.008
Exclusivity	0.004
Genericity	0.945
Recognition	0.683
Divergence	0.852
Synchronicity	0.860
Consensuality	0.085
Resemblance	0.538

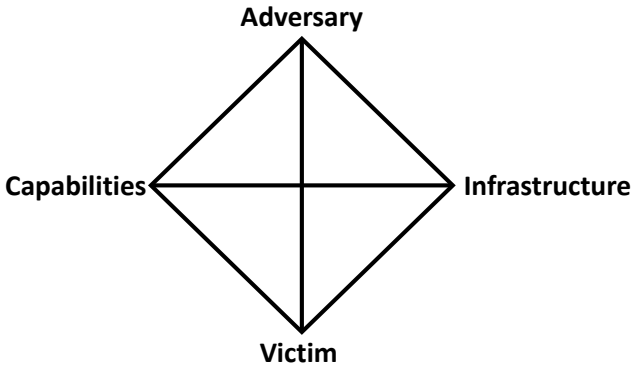


Figure 4.1. Diamond Model of Intrusion.

improve the security of the critical infrastructure.

Stimulated by the importance of the topic, also academic research teams are performing several studies to provide innovative solutions that can be applied for cyber defense [46, 71]. Unfortunately, despite the great number of released public reports and sources, there is not a comprehensive and updated public dataset that can be used to design, test and compare the different proposed methodologies. Thus, gathering enough information for fully understanding a particular APT is a long and complex process, requiring also a huge manual effort to retrieve the information from different sources that cannot be easily merged because they use different naming schemas or data structures. For this reason, we decide to design and share dAPTasetwith the research community.

In the designing of this database, we decided to use the *diamond model* [13], shown in Figure 4.1. It is typically used to describe intrusions and in our opinion, it can be an interesting representation of APT attacks. In its simplest form, the diamond model describes that an *adversary* deploys a *capability* over some *infrastructure* against a *victim*. In this model, adversaries are APT groups, capabilities are malware, exploits, tools, and techniques used, infrastructures are the physical and logical communication structures (IP addresses, domain names, etc.) and the victims are the targets (people, assets, financial organizations, Critical Infrastructures, etc.).

Many open-source platforms present various information about APTs, but do

not fully cover multiple edges of the model, or can be easily used for research purpose.

MITRE ATT&CK[9] is a knowledge base created and maintained by MITRE containing information on APTs, tactics, techniques, and names of the adopted software. On the other side, it does not provide elements like malware, IP addresses, or domains.

APTNotes[10] is a GitHub repository containing publicly-available reports and web articles related to APTs. Each report covers one or more of the diamond edges, but it lacks any standard data structure, so it is very difficult to take a comprehensive look among various documents. Moreover, one of the main problems when using these sources is the great inhomogeneities among the naming systems of different cybersecurity vendors. Hence, reports related to the same group can use different names. Unfortunately, despite the great number of released public reports and sources, there is not a comprehensive and updated public dataset that can be used to design, test and compare the different proposed methodologies. Thus, gathering information for fully understanding a particular APT is a long and complex process. It requires a huge manual effort to retrieve the information from different sources and merge them together.

In the following sections, we will explain details on the structure of the dataset and the methodology used to build it.

4.2.1 Dataset Structure

In this section, we present the structure of the database we have created to store APT information. In designing the database we have taken into account the following arguments: i) all the sources we use to build our dataset are considered as reports (also web pages) that can describe relationships among APTs and campaigns, APTs and different Indicators of Compromise (IOCs), such as samples or IP addresses, or even refer to other reports; ii) information about reports can be useful to verify the source and check if it has been updated from the last analysis and should be reprocessed; iii) more names are attributed to the same APT, hence one of them (the one chosen by MITRE ATT&CK) is used to identify the APT while the others must be saved as aliases to ease the research; iv) we are not interested in the campaigns, but their names can be useful to associate a report to an APT, when the APT name is not explicit, hence we consider campaign names as keywords.

Given those guidelines, dAPTaset is organized as in Figure 4.2 and is composed by the following objects[11]:

- **APT** is the fundamental concept and represents the attacker group. The table has only one field (apt_name) containing the common name of an APT, generally used for the group. To avoid the missing of string variants, APT names are stored with lower case letters and without any space or punctuations. We also store the original name as alias in KEYWORDS table.

[9] https://attack.mitre.org/

[10] https://github.com/aptnotes/

[11] A detailed description of the database is provided in our GitHub repository: https://github.com/GiuseppeLaurenza/dAPTaset

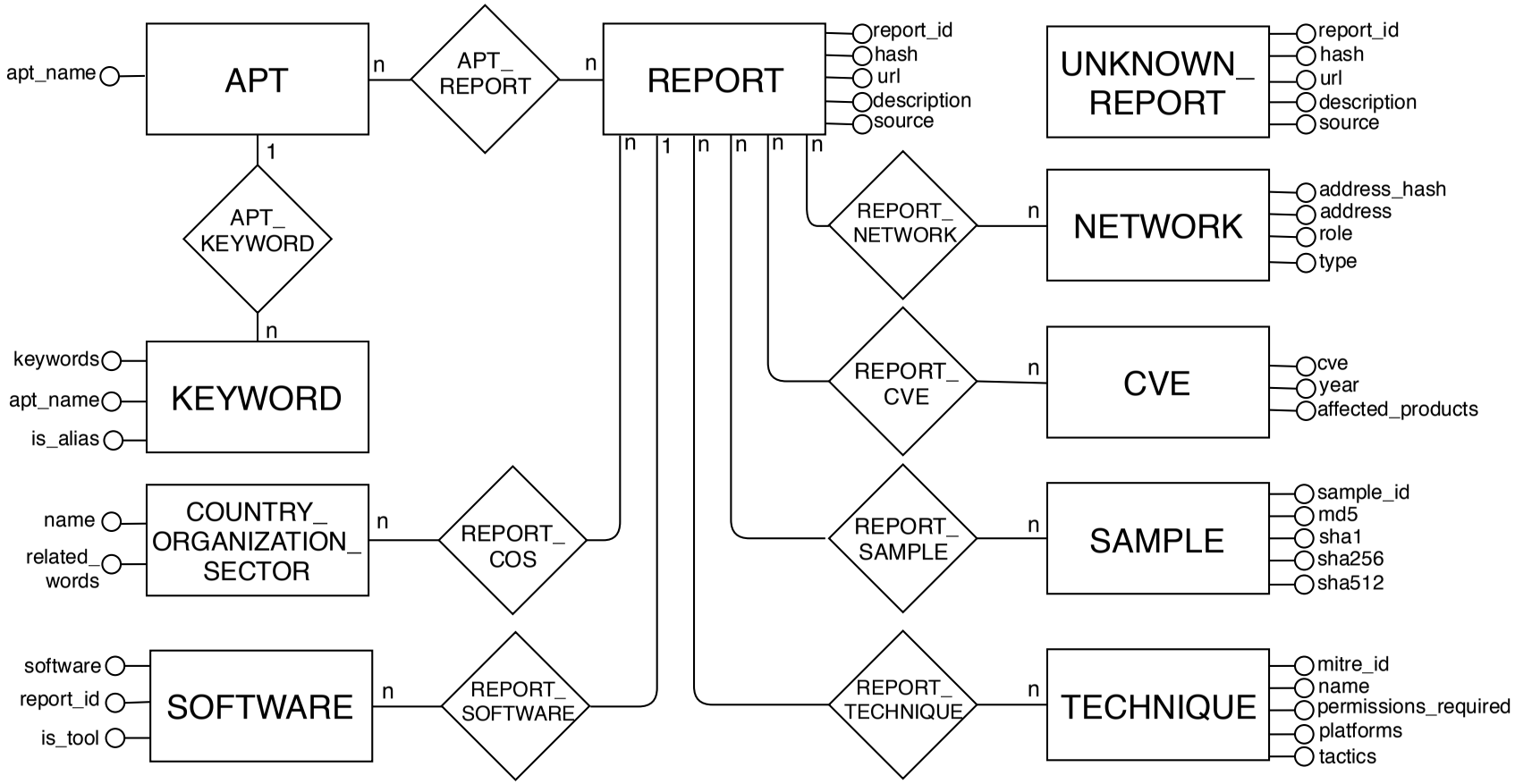


Figure 4.2. ER Diagram (we have omitted two attributes in each entity that respectively contain the timestamp of the creation of the record and the one of its last modification).

- **KEYWORD** contains all the words related to APTs, such as alias, campaigns, etc. These strings, together with the common name, are used to simplify APT recognition when analyzing a report. We also include a flag to distinguish between aliases and other keywords, where aliases are other names used to refer to the main APT group, while keywords can be any word (or also a small phrase) related to the APT. Such keywords can be used to associate a report to an APT, when the APT name is not provided in the report.
- **REPORT** refers to all the collected and analyzed documents that contain information about APTs: pdf files or web pages. For each report, we store the hash of the document to easily verify whether two documents are the same but have different names, or to check whether a document has been updated and deserves to be analyzed again. We also include a simple description about the document and two fields to store the source, one representing the origin of the report and one the url from which the document can be downloaded, if the document comes from the web. Such information can be also useful for users interested to retrieve the original reports.
- **SAMPLE** is related to the APT malware files. The table contains the various hashes of the malware used by APTs to perform their activities. Hashes are obtained by the *parsing* of the documents contained in the REPORT table. From reports, we usually find only one of the possible hashes per file: MD5, SHA1, SHA256, etc. Other hashes are subsequently searched from other sources.
- **NETWORK** table contains all the IOCs related to networking, such as IP addresses, URLs or email addresses. Elements in this table are obtained by *parsing* the documents contained in the REPORTS table. Elements could be not necessarily malicious, hence we use a *role* field to describe their relationship with attacks, set through semi-automatic validation.
- **CVE** contains a list of *Common Vulnerabilities and Exposures* exploited by APTs during their activities. In addition to the identifier, this table also reports other information that can help analysts, such as the *affected_products* field that contains a list of software or devices affected by the vulnerability.
- **TECHNIQUE** presents information about how APTs perform particular actions, such as obtaining high privileges, performing lateral movements or communicating with the command and control.
- **COS** table contains all Countries, Organizations or Sectors that are related to APTs activities, such as the suspected victim sectors or state sponsors. For each entity, we also include a list of possible alias or related keywords.

The database also contains two support tables (**UNKNOWN_REPORT** and **SOFTWARE**) to store data that need further investigation prior to becoming useful and be inserted into the real dataset. Many-to-many relationships between tables are implemented through new tables and sometimes contains some additional information. For example, the **APT_COS** table has a field describing if the APT is targeting or is sponsored by a COS.

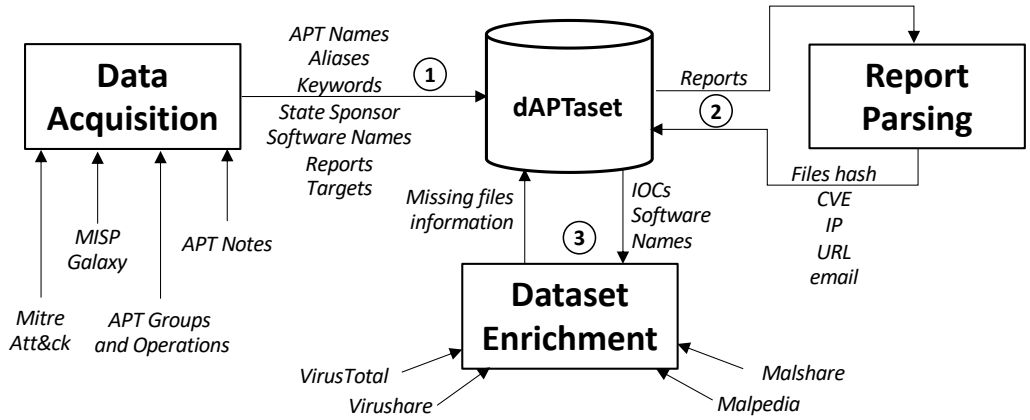


Figure 4.3. Schema of the mechanisms for the data acquisition.

4.2.2 Data Collection

In this section, we provide an overview on how we collect data, detailing the origin and the processing. Figure 4.3 briefly describes the mechanisms.

Data Acquisition

The following are the main sources that we analyze to extract information and build the skeleton of our dataset. From them, we principally acquire the main names of APTs, their aliases, the suspected victims of their attacks, and the address of most of the public documents.

MITRE ATT&CK

The first source considered is *MITRE ATT&CK*, accessed through its GitHub repository. From there, we extract the following information: *name*, *aliases*, *description*, *malware names*, *tool names* (MITRE does not provide the hash of the capabilities), *techniques*, *tactics* and the list of *external references*. For each group we create an entry in the APT table, then we insert its *aliases* into keywords. We also store the MITRE group webpage in a record of the REPORT table, compute the hash of the group data, set url as the address of the corresponding webpage on the ATT&CK website, copy its description in the table, and set *"mitre"* as source. For each external reference, we download the report and create a new record in the REPORT table. Similarly, we extrapolate techniques to populate the relative table, and malware and tools names to create records into the SOFTWARE table.

APT Groups and Operations

Another useful resource is the *APT Groups and Operations* spreadsheet[12] created by Florian Roth. The spreadsheet is subdivided in tabs, each one related to a specific country/region to whom APTs have been attributed. In addition, there are two other tabs, one for other areas and one for groups with unknown location. Each row of the tabs represents a particular APT. These sheets have different columns,

[12] `https://apt.threattracking.com`

but they can be roughly grouped in: (common) *APT name*, *aliases*, *operations*, *malware*, *toolsets*, *targets*, *tactics*, *comments* and *external references*. Periodically we fetch this document and for each APT row we check if one of the names or aliases is present in the APT table. If there is a match, then we add the other ones and the operation names as entries in KEYWORD, the first ones as aliases and the latter as normal ones. Then we create a record corresponding to the APT row in REPORT, using the hash of $Nation\_CommonName$ as hash (where $Nation$ is the tab name and $CommonName$ is the name attributed to the APT). We leave description and url fields empty and we set *"groups_and_operations_sheet"* as source. To have a common representation of the data in the *targets* column, we take advantage of the deep learning *Question Answering System* of DeepPavlov[13], an open-source library for deep learning end-to-end dialog systems and chatbots. We used it to find which are the victim entities according to the text and then fill the COUNTRY_ORGANIZATION_SECTOR table. Names or abbreviations of nations are checked against data from RestCountries[14]. Last, we download the *external references* and we handle them as additional reports to be processed, in the same way, described for MITRE ATT&CK.

APTNotes

Periodically, we download all the new documents from the repository, and we search inside them all the APT names and keywords we know, intending to link each report to a known APT. If we are able to attribute the report to an APT, we add it to the REPORT table, otherwise, we insert a record in UNKNOWN_REPORT and we do not process the report.

MISP Galaxy Cluster

MISP[15] is a free and open-source sharing platform for helping information sharing of threat intelligence. Usually, MISP users create different groups and share information only inside them. Even if it is possible to be part of different groups, the absence of a *general official* group does not allow a simple global sharing. The MISP project includes multiple sub-projects to support the operational requirements of analysts and to improve the overall quality of information shared. One of them is MISP Galaxy[16], that is a simple collection of clusters that can be parsed to add information to our dataset. Being the focus of our work on APTs, we analyze the *Threat Actor Cluster*. From it, we extract aliases of APTs to enrich KEYWORD table, suspected target and victims to be added to COUNTRY_ORGANIZATION_SECTOR table. Similarly to what we do for APT Groups and Operations, we check RestCountries to validate names and abbreviations of countries.

Report Parsing

As previously introduced, each recognized document is parsed to identify information that can enrich the database tables. We use IOCExtract from Microsoft

[13] `https://deeppavlov.ai`
[14] `https://restcountries.eu`
[15] `https://www.misp-project.org`
[16] `https://www.misp-project.org/galaxy.html`

Threat Intelligence Security Tools[17] to extract IOCs, file hashes, IP addresses, and network URLs. Moreover, we use regular expressions to identify emails and CVEs. Finally, we look for the presence of KEYWORD entries into the documents to find any possible relationship with APTs. For this purpose, we first check into title and metadata of the documents. If there is no match, we then look for keywords into the entire text. For any match, a relationship is created between the report and the related group.

Dataset Enrichment

dAPTaset is finally enriched through other cybersecurity websites: VirusTotal[18], VirusShare[19], Malshare[20] and Malpedia[21].

Hashes Unification

SAMPLES table contains various hashes collected from different documents, thus more records may refer to the same capability. This happens, for example, when we retrieve the md5 of a file from a report and a different hash from another one. In addition to the undesired presence of more records, this can hide relationships between apt and report, because such samples can be linked to distinct reports. Using VirusTotal as a search engine, for each file we collect all the possible hashes in order to fill the empty fields. When there are hash repetitions, records are merged and existing relationships are linked to the new record.

Hashes Enrichment

dAPTaset also has a temporary SOFTWARE table. We search for malware names in all the previously listed websites in order to collect samples of these software. These new hashes must be verified before to link them to APT, so we take advantage of the Antivirus Vendor labels provided by VirusTotal to look for keywords and use them as proof of correctness. Each time a new sample is correctly validated, we add it to the SAMPLE table.

Network Validation

NETWORK table contains network-related IOCs retrieved according to Section 4.2.2. To validate such elements, we check again VirusTotal. In the case of a match, we report a brief result of the response in the role column.

CVE Validation

Similarly to NETWORK table, also the CVE table needs validation and enrichment. For this purpose, we rely on the online repository CVEList[22]. For each *new* vulnerability extracted from a report, we check whether it is valid and then collect other information to enhance the data stored in the corresponding table.

[17] https://github.com/microsoft/msticpy
[18] http://virustotal.com
[19] http://virusshare.com
[20] https://malshare.com
[21] https://malpedia.caad.fkie.fraunhofer.de
[22] https://github.com/CVEProject/cvelist

dAPTaset Statistics

Using the methodology described, we have built dAPTaset. Currently, we found 88 different group names, described through 821 reports that allowed us to find 21841 binary hashes with 8927 unique files, 175 different techniques, 2620 network related IOCs and 169 CVEs related to APTs activities. dAPTaset can be downloaded from our GitHub repository[23].

4.3 Considerations

It is easy to imagine that dAPTaset can be an important resource for both Critical Infrastructures and researchers. In fact, it is the first complete database about APTs, making easier to obtain complete information about each of the main aspect of a group. As explained in previous chapters, one of the most important ways to raise CIs' security is through malware, thus we focus our experiments on the dAPTaset binaries. Obviously both for defense or research purposes, it is possible to take advantage of all the other information provided by dAPTaset. Moreover, dAPTaset is continuously semi-automatically updated and can be easily expanded adding more external sources, thus it can be suitable for a wide set of purposes.

Instead, MalFamGT is our *reference dataset* for our tests related to malware analysis. We use it as a testbed for various experiments and make it public to allow others to perform their tests and compare their results with ours. But it can be easily freely replaced with larger datasets. For example, one can leverage providers like Malshare to collect more malware and execute a wider set of tests. However, we observe that MalFamGT is enough for many experiments, like comparing different learning algorithms, as shown in Chapter 5 and Chapter 6. In fact, as reported in Section 4.1.1, MalFamGT contains more than forty families of different sizes, but all of them are fairly represented. Thus we consider it adequate for many purposes and hence it can be helpful for security experts for evaluations or training of models. In the following chapters, it will be more clear that, despite the size, it is possible to see differences in performance among the various algorithms, differences that can only increase with a larger set of samples.

[23] `https://github.com/GiuseppeLaurenza/dAPTaset`

Chapter 5

Malware Family Detection and Identification Through Clustering

As stated in Chapter 1, Critical Infrastructures must maintain their information about malware updated. Due to their nature as a preferred target, CIs are usually the victims of most of the novel malware. For example, the previously described *Stuxnet* uses malware to exploit four zero-day vulnerabilities to attack a nuclear power plant.

Thus, CIs need to known details about detected malware as fast as possible in order to obtain information about the infection, organize system recoveries and improve defenses. But the number of detected malware is so huge that it is not possible to analyze in depth all of them with the limited human resources a CI has. A typical approach to reduce such a workload is to reduce the number of malware that requires manual analysis. One way is to identify the family a malware belongs to, thus malware experts can analyze only a small representative subset of each group. Moreover, detection of new families can be important, so that analysts can focus on the new families, avoiding to inspect already studied malware.

In this chapter, we present our approach to performing malware family detection and identification through the use of clustering algorithms. In our approach, each cluster models a family. Clustering allows us to perform family detection and identification in a single step. In fact, clustering divides malware without any previous knowledge, thus it can easily detect new families among others. A very positive *side effect* of the division is obviously the malware identification, having performed both the operations in a single phase.

In particular, we propose the use of *Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH)* as our algorithm, demonstrating how it is an optimal choice for this class of problem in both quality and time performance [77].

5.1 Balanced Iterative Reducing and Clustering Using Hierarchies (BIRCH)

BIRCH is a hierarchical clustering algorithm proposed in 1996 by Zhang *et al.* [107]. It can cluster very large datasets in Euclidean space with the ability to incrementally and dynamically cluster incoming multi-dimensional data points. It is built around the concept of *Clustering Feature [CF]* and *CF Tree*. The first is a tuple summarizing the information about a cluster: it contains the number of data points and the linear and square sums of them. The latter is a height-balanced tree with two parameters: the branching factor B and threshold T. T is fundamental in the *training part*, it represents the maximum *radius* for clusters, where the radius of a cluster is the average distance of member points from the centroid. If the radius of a cluster becomes larger than such threshold, then a new cluster is created.

5.2 Features

We run available malware on the Cuckoo Sandbox, a dynamic analysis tool introduced in Section 3.4, obtaining as output one report per sample. Each report contains a textual description of the static and dynamic characteristics of malware. Static characteristics include readable strings inside the binary, while dynamic characteristics comprise operations on the file-system, on the registry and network activities. We process all the information contained in the report, in order to understand as much as possible from the analysis. Each feature vector has 241 numeric features. String features are converted to numbers with a hash function based on the index and the ASCII value of their characters. In detail, each character in the string is converted into a number, i.e. the associated ASCII code, and multiplied by its position. The sum of all these results is the final hash. In this way is possible to have a number that represents the string. We chose this function because similar strings tend to have similar hashes, i.e. the distance between two strings proportionally increase relating to the number of different characters. A large part of the features is counts (*e.g.*,written files count, read registry keys count, etc.) and existence checks (*e.g.*,for particular function calls). They can be grouped into different categories such as static features, file-system related features, etc. To be more robust to outliers, we scale feature values on the basis of median and interquartile range. To reduce computation time we perform also a selection of relevant features. A feature can be useless if all samples have the same value, hence we remove all the features presenting the same value in more than 80% of samples. A summary of used features is reported in Table 5.1, a full list is presented at Appendix A.

5.3 Experimental Evaluation

In our tests we compare accuracy and performance of BIRCH with respect to other well-known clustering algorithms against **MalFamGT** with both labelling system, as described in Section 4.1.

Table 5.1. Summary of the used features divided by category.

Category	Feature count
Static	58
File-system	53
Registry	36
Process	68
Network	26

5.3.1 Clustering Algorithms

The clustering algorithms we choose for comparison are:

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [22]: a density-based algorithm that can discover clusters of arbitrary shapes and is efficient for large spatial datasets. The algorithm looks for clusters by searching the neighborhood of each data point in the dataset.

Hierarchical algorithms [59]: methods that construct a tree of clusters, also known as a dendrogram. Every cluster node contains child clusters, this approach allows exploring data on different levels of granularity. These algorithms are divided into *agglomerative hierarchical algorithms* and *divisive hierarchical algorithms*. The agglomerative hierarchical clustering is a bottom-up method that starts with every single instance in a single distinct cluster. Then it repeats merging the closest pair of clusters according to some distance measure until all of the data are in one cluster. The divisive hierarchical clustering has an opposite approach, it divides clusters until only singletons of individual points remain. All these algorithms need a *linkage criterion* to determine the distance between sets of observations as a function of the pairwise distances between observations. Most common linkage criteria are *Ward's*, *Single*, *Complete* or *Average*. A weakness is that they do not scale well, because they typically use a distance matrix.

K-Means [58]: a well known center-based algorithm. The center of each cluster, called centroid, is calculated as the mean of all the instances belonging to that cluster. The number of clusters k is assumed to be fixed. The goal is to minimize a certain error value, in this case, it is the Sum of Squared Error (SSE), which measures the total squared Euclidean distance of instances from their centroid. In each iteration, every instance is assigned to its nearest cluster center according to the Euclidean distance between the two. Then the cluster centers are re-computed.

Mini-Batch K-Means [84]: a variant of K-Means algorithm which uses mini-batches to reduce computation time, while still attempting to optimize the same objective function. Mini-batches are subsets of the input data, randomly sampled in each training iteration. These mini-batches drastically reduce the

amount of computation required to converge to a local solution. In contrast to other algorithms that reduce the convergence time of k-means, mini-batch k-means produces results that are generally only slightly worse than the standard algorithm.

Expectation Maximization (EM) [21]: a general-purpose statistical iterative method of maximum likelihood estimation in the presence of incomplete data which can be used for the purpose of clustering. It is simple, stable and robust to noise. This algorithm begins with an initial estimate of the parameter vector and then alternates between two steps iteratively: the expectation step (E-step) and the maximization step (M-step). In the E-step, the conditional expectation of the complete data likelihood given the observed data and the current parameter estimates is computed; in the M-step, the parameters that maximize the expected likelihood from the E-step are determined.

Clustering using representative (CURE) [27]: a scalable and efficient algorithm for large datasets that is more robust to outliers and finds clusters of arbitrary shape and size. In this algorithm, each cluster is represented by a constant number of points that are well scattered in the cluster.

5.3.2 Accuracy Evaluation

We consider the following accuracy metrics

Adjusted Rand Index (ARI): Rand index is defined as the number of pairs of samples that are either in the same group or in different groups in both partitions, divided by the total number of pairs of samples, assuming the generalized hypergeometric distribution as a model of randomness. The adjusted Rand index is a value between -1 and 1, where the best is 1, and its expected value is 0 in case of random clusters. A larger adjusted Rand index means a higher agreement between two partitions.

Adjusted Mutual Information (AMI): Mutual information is a measure of mutual dependence between two variables. The adjusted Mutual Information, similarly to the ARI, assuming a generalized hypergeometric distribution as a model of randomness, subtracts the expectation value of the MI, so that the AMI is 0 when two different distributions are random, and 1 when they are identical.

Fowlkes-Mallows Index (FMI): FMI is defined as the geometric mean of the pairwise precision and recall. It is 1 when two clusters are the same and 0 when they have nothing in common.

5.3.3 Algorithm Tuning

All these algorithms rely on specific parameters that affect their behavior. BIRCH depends on a threshold that defines the maximum radius of clusters. K-Means needs the setup of the number of clusters to generate. The two main parameters of DBSCAN are (i) the maximum distance between any two samples to be considered

in the same neighborhood, and (ii) the minimum amount of data points in a neighborhood to be considered a cluster. The hierarchical algorithms need a threshold to decide when to cut the tree. Expectation-Maximization has several parameters, among which the number of mixture components. Also, the used implementation of CURE needs to set up in advance the number of clusters and also the number of representative points for each cluster. For this algorithm, it is also important a coefficient that defines the level of shrinking of representation points. For the algorithms with multiple parameters, we tuned each parameter one by one trying to maximize the result. We observed that many algorithms require to know the number k of clusters in advance (*e.g.*,K-Means). For this reason, in our case, they are not really useful, but we decided to include them anyway in the comparison to show their results against the other algorithms. Many others cannot scale well with a large amount of data (*e.g.*,Mean-shift, OPTICS, SyncSom and Affinity propagation) requiring a computation time too long, so we did not include them in our report.

For each algorithm, we tune parameters iteratively to maximize accuracy. In particular, to tune algorithm parameters, we choose to maximize the FMI, because it is a concise representation of precision and recall and it is easier to interpret compared to the ARI. The used maximization approach is based on numerical approximation with direct method: the same algorithm is executed a large number of times on the same input data, each time with a different value of the parameters, in order to cover as much as possible a given range of values. For example, for BIRCH, we vary the value of the threshold between 0 and 10 with a step of 0.1. Figure 5.1 shows the FMI of BIRCH for both ground truths by varying the threshold value. Clustering algorithms can generate clusters that depend on which order the samples are analyzed, thus we execute each algorithm 40 times with samples sorted differently and provide mean and standard deviation (SD) of each accuracy metric.

5.3.4 Clustering Quality Evaluation

Figures 5.2 to 5.4 and Table 5.2 report the obtained results. For each metric m, where $m \in \{FMI, AMI, ARI\}$, we use m_M when the ground truth based on Malheur is used, and m_A when the one based on AVclass in employed. These results show that BIRCH mostly achieves an accuracy highest than the other algorithms, except for hierarchical algorithms which generally obtain slightly higher values.

5.3.5 Clustering Time Evaluation

We test all the algorithms on a Virtual Machine with 4 Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz, dual core virtual socket (8 logical cores) and 8GB of RAM. Figure 5.5 shows average and standard deviation of the execution times of each algorithm. Reported results clearly indicate that BIRCH is faster than all the other algorithms, except for the Mini-Batch K-means, which however turns out to be the worse in terms of accuracy. The most accurate algorithms (i.e., hierarchical algorithms) exhibit instead worse performance than BIRCH requiring execution time three-four times longer. This difference in performance, although not so high, be-

Table 5.2. Comparison of several clustering algorithms.

Algorithm	k	ARI_M	AMI_M	FMI_M	ARI_A	AMI_A	FMI_A
BIRCH	NO	0,947±0	0,616±0	0,955±0	0,819±0	0,727±0	0,851±0
DBSCAN	NO	0,911±0	0,515±0	0,927±0	0,865±0	0,606±0	0,891±0
Hierarchical WardâĂŹs linkage	NO	0,917±0	0,54±0	0,932±0	0,828±0	0,686±0	0,859±0
Hierarchical single linkage	NO	0,949±0	0,607±0	0,958±0	0,891±0	0,739±0	0,911±0
Hierarchical complete linkage	NO	0,949±0	0,605±0	0,957±0	0,786±0	0,697±0	0,826±0
Hierarchical average linkage	NO	0,949±0	0,606±0	0,957±0	0,891±0	0,739±0	0,91±0
K-Means	YES	0,644±0,104	0,535±0,008	0,698±0,089	0,738±0,046	0,625±0,025	0,792±0,035
Mini-Batch K-Means	YES	0,468±0,133	0,437±0,035	0,544±0,118	0,517±0,087	0,586±0,03	0,601±0,074
EM	YES	0,879±0,056	0,51±0,018	0,902±0,048	0,879±0,004	0,648±0,006	0,902±0,003
CURE	YES	0,67±0	0,416±0	0,76±0	0,691±0	0,566±0	0,769±0

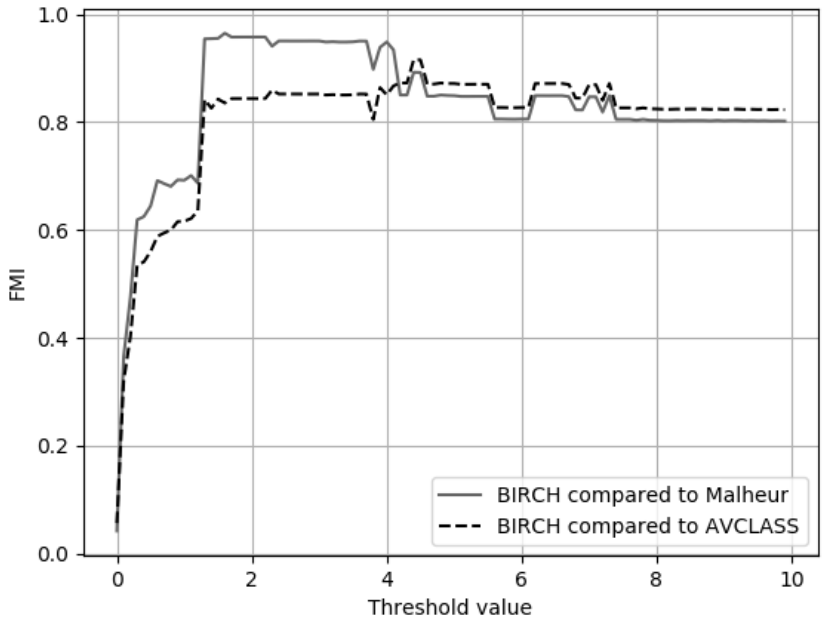


Figure 5.1. FMI variations with a different threshold for BIRCH.

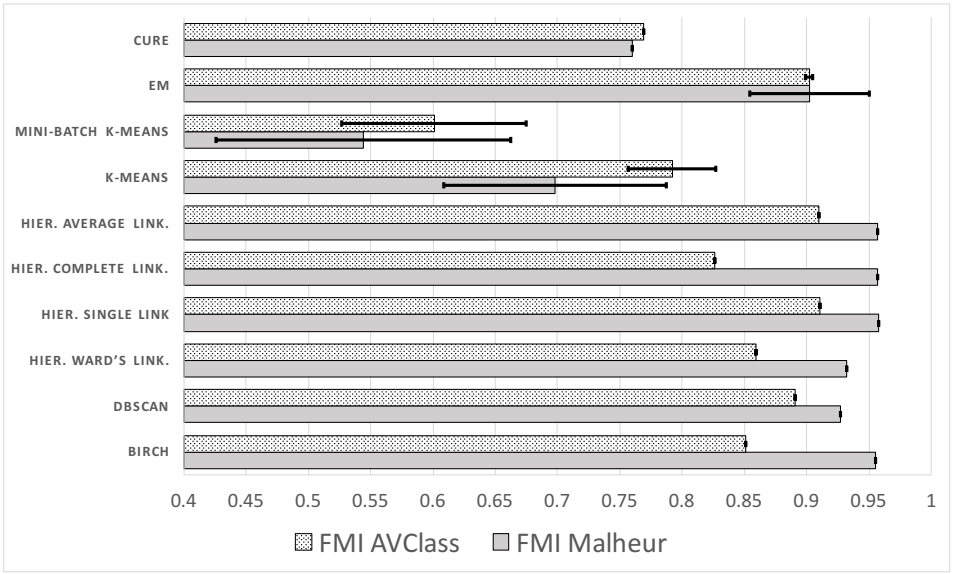


Figure 5.2. FMI achieved by clustering algorithms.

comes strongly relevant in the scenario of Critical Infrastructures, when the analysis of malware should be performed as soon as possible to keep the security level high and design recovery procedures to quickly return to a working state.

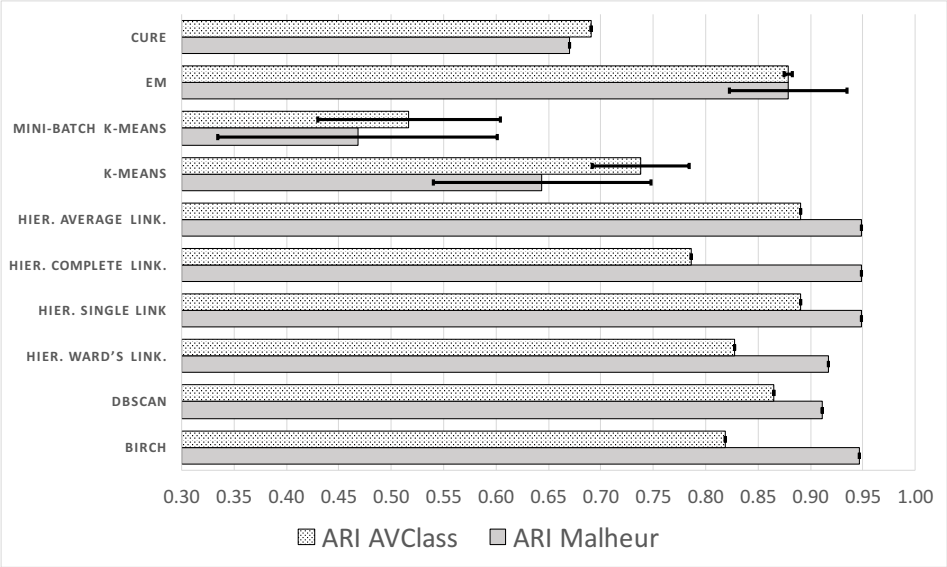


Figure 5.3. ARI achieved by clustering algorithms.

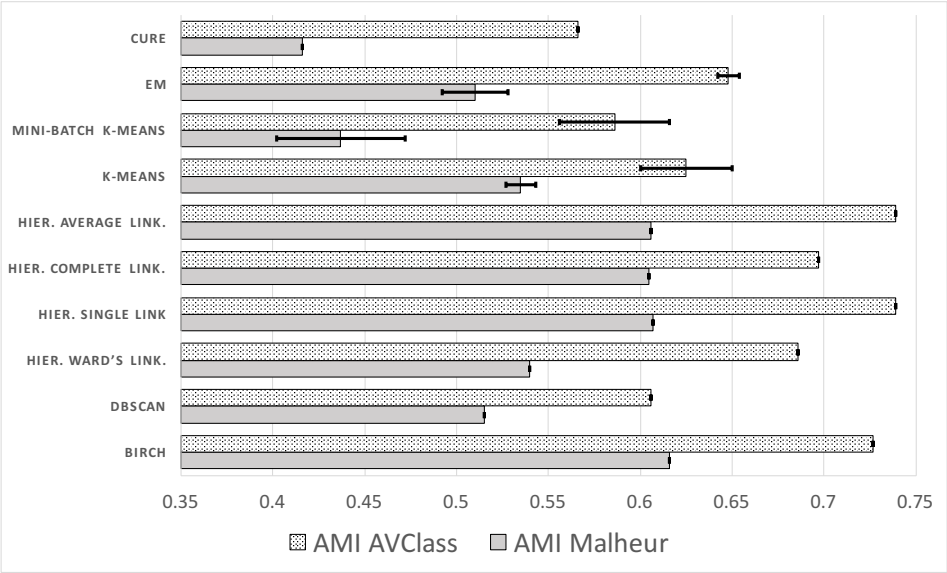


Figure 5.4. AMI achieved by clustering algorithms.

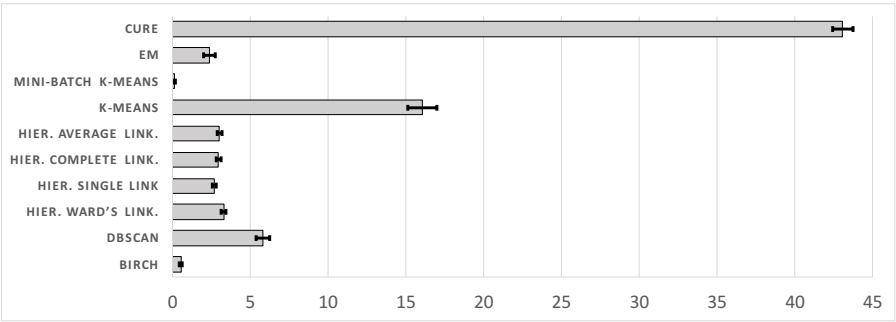


Figure 5.5. Execution times in seconds of clustering algorithms.

Chapter 6

MalFamAware: Automatic Family Identification and Malware Classification Through Online Clustering

In Chapter 5 we demonstrate how clustering malware in families can save security experts' time. In fact, through a fastest analysis and understanding of threats, it is possible to quickly react and recovery, strongly reducing the impact of the attack. As explained before, analysts can focus only on a small representative for each family and also only on new families, heavy cutting the number of samples to analyze.

However, in order to recognize new families, the whole dataset of available malware needs to be clustered from scratch (*re-clustering*) whenever new samples have to be analyzed. This is the main limitation of this approach because the clustering time grows at least linearly with the size of the dataset times the number of clusters, hence the huge number of malware discovered every day makes this approach impractical. In fact, especially in scenarios like CIs defenses, time is the most important resource to save in order to act as soon as possible. As explained in the previous chapter, a deep understanding of malware is required to improve the security and design recovery measures, thus long processes like full re-clustering should be avoided. A confirmation that an online approach is more suitable to save time can be found in the literature, where works such as Litl *et al.* [53] and Liu *et al.* [56] demonstrate how much it is possible to reduce the total clustering time using such class of algorithms in various scenarios.

Such a scalability issue can be mitigated by using an *incremental clustering* approach, like the one applied by Malheur [80]. A classifier is periodically trained with the families identified so far through clustering. This classifier is used to analyze new samples and determine what known families they belong to. Only malware that have not been classified confidently enough to any known family are then clustered to identify new families and incrementally improve the current knowledge. The classifier is then retrained on this updated knowledge. The limitation of this approach lies in the difficulty of choosing the correct batch size or retraining period.

Approach	Malware classification	Family identification
Classifier-only	Classifier trained on available ground truth	No family identification, it needs period retraining
Re-clustering	Classifier trained on available ground truth	Ground truth updated by periodic clustering of the whole dataset
Incremental clustering	Classifier trained on available ground truth	Ground truth updated by periodic clustering of malware whose family is still unsure
Online clustering	*Done simultaneously*	

In fact, too frequent retraining likely brings again scalability problems, like training a classifier on an ever-growing ground truth becomes more expensive as time goes by. On the other hand, waiting too long for retraining prevents to steadily identify and react to new malware families.

To solve this problem, we have proposed MalFamAware, a novel approach to *incremental automatic family identification and malware classification*, based on *online clustering*, where samples are efficiently assigned to families as soon as they are fed to the clustering algorithm, and the families (i.e. the clusters) are updated accordingly. This *update* only modifies the number and the composition of the families without altering the features space, which remains fixed and consistent among the various executions. The advantage over incremental clustering lies in cutting the cost of periodically re-training a classifier. One of the contributions of this work is the idea of using online clustering to implement family identification and malware classification simultaneously. Table **??** shows an overall comparison between existing approaches for malware classification and family identification.

MalFamAware uses BIRCH as the online clustering algorithm. As demonstrated in Chapter 5, BIRCH is one of the best clustering algorithms for family identification and it can be used in an online fashion.

Contrary to our previous chapter, this one focuses on both *family identification* and *malware family classification*, presenting a solution to perform them in the same phase. Indeed, MalFamAware classifies new incoming malware into the correct family, if already known, or creates a new one if no suitable existing family is found. We show that MalFamAware outperforms approaches based on re-clustering or incremental clustering, both in terms of accuracy and execution time. In particular, MalFamAware requires three orders of magnitude less time than other clustering-based approaches to analyze samples, while providing slightly better accuracy.

We also compare the accuracy of MalFamAware against that achieved by using different standard classifiers trained in two distinct ways: only once at the beginning or periodically on all the previous samples. Results show that the accuracy of MalFamAware is comparable to that of other classifiers when samples belong to known families, and it is larger for samples of new families.

6.1 MalFamAware Approach

As introduced at the beginning of the chapter, MalFamAware implements online clustering through BIRCH. Rather than for performance, we choose it because it can undo what done in the previous steps to correct the clustering according to the newly coming samples [28]. When a new sample is fed to BIRCH, it gets assigned to either an existing family or to a novel family created on purpose for that sample. Hence, family identification and assignment of samples to families are carried out together, which makes BIRCH extremely efficient.

As explained in Section 5.3.3, the way BIRCH clusters vectors is based on a threshold representing the maximum *radius* for clusters: if a cluster happens to have a radius greater than such threshold then a new cluster is created. This threshold is set once and cannot be changed unless re-clustering the whole dataset. In our approach, we include a bootstrap phase where we use a ground truth to learn the value of this threshold. In the specific, we choose the threshold which maximizes the accuracy of BIRCH clustering with respect to the ground truth and keeps such a threshold fixed to analyze all the blocks after the bootstrap phase. In Section 6.2 we also consider, for comparison purposes, a version of BIRCH in total re-clustering mode, where all the available samples are reclustered whenever a new batch is provided. All these total re-clusterings are carried out using the same threshold learned in the bootstrap phase. We refer to this version as *BIRCH offline*.

6.2 Experimental Evaluations

We perform all our tests on a machine with Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz, using 4 cores and 8GB of memory. The operating system is an Ubuntu 16.04.2 with kernel GNU/Linux 4.4.0. We employ public and well-known libraries[1][2] for the machine learning algorithm implementations.

We first describe the dataset used in the evaluation (see Section 6.2.1), then present the experiments on accuracy (see Section 6.2.2), where we compare our solution based on BIRCH online with (i) BIRCH offline, (ii) Malheur with total re-clustering and incremental clustering, and (iii) several classifiers. We also deepen the differences in accuracy when samples to analyze are more likely to belong to unknown families. Finally, we report the results on performance comparison (see Section 6.2.3).

6.2.1 Reference Dataset

We tested our approach on 4.1, using the AVclass labels as ground truth. To simulate as properly as possible the actual temporal distribution of samples in batches, we sort the dataset by the timestamps reported in their header. Although some samples have clearly forged timestamps (*e.g.*,in the future), in the large majority of cases timestamps seem to be realistic. In the absence of more authoritative sources, we

[1] `http://scikit-learn.org/`
[2] `https://www.scipy.org/`

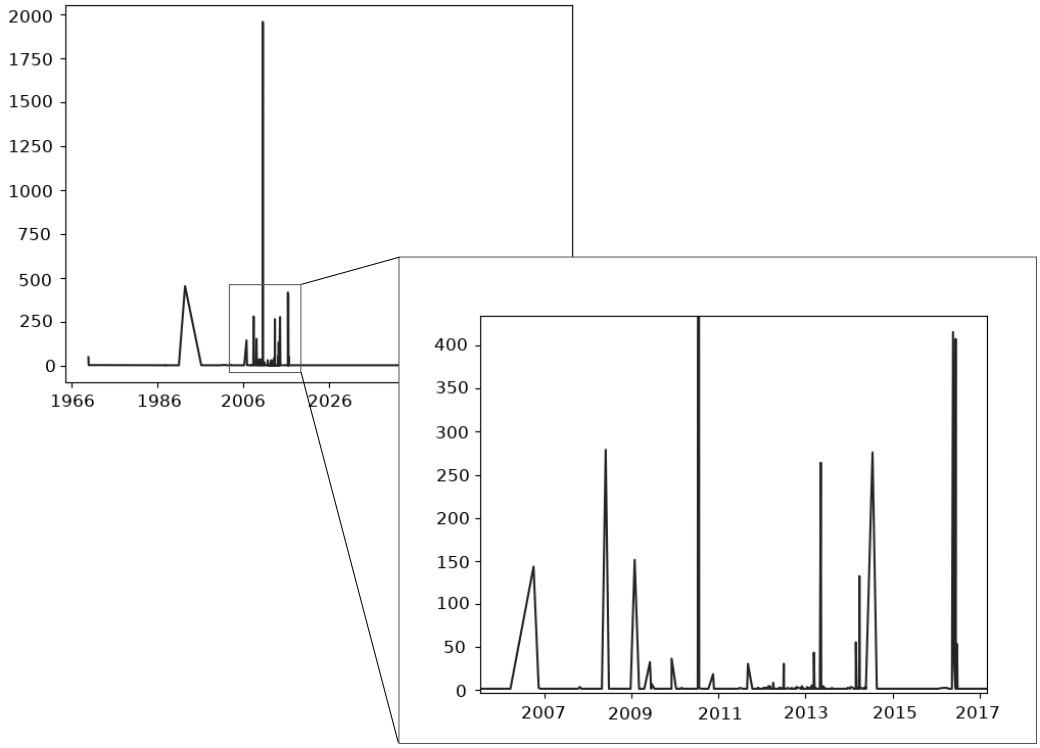


Figure 6.1. Temporal distribution of samples.

choose to rely on those timestamps. The distribution of samples over time is shown in Figure 6.1.

We include in our reference dataset only those samples between 2006 and 2016. The experiments are aimed at comparing different approaches to incremental malware family identification, thus we sorted samples by timestamps and then we divide them into ten blocks, as in the classical cross-folder approach. Each resulting block contains approximately 400 samples. Figure 6.2 reports for each block the percentage of samples belonging to unknown families, i.e., those families for which no sample is present in any of the previous blocks. This information is useful to specifically evaluate the accuracy of an approach for samples of families that are unknown so far.

6.2.2 Accuracy Evaluation

Evaluating the accuracy of an algorithm for malware family identification boils down to measure to what extent a clustering is similar to the available ground truth. A problem to address is that the labels of our ground truth, produced by AVclass, are different from those generated by evaluated clustering algorithms, which assign anonymous labels such as "cluster1". We have to choose comparison metrics that are independent of the naming space of labels, thus we decide to use the precision, recall, and accuracy (i.e., the F1 score) as defined in [88]. Given the set of labels $\mathcal{L}^{GT} = \{l_i^{gt}\}$ of the classes of the ground truth, with $i = 1 \dots N^{GT}$, let n_i^{gt} be the number of samples labelled with l_i^{gt}. Given the set of labels $\mathcal{L}^{C} = \{l_i^{c}\}$ produced

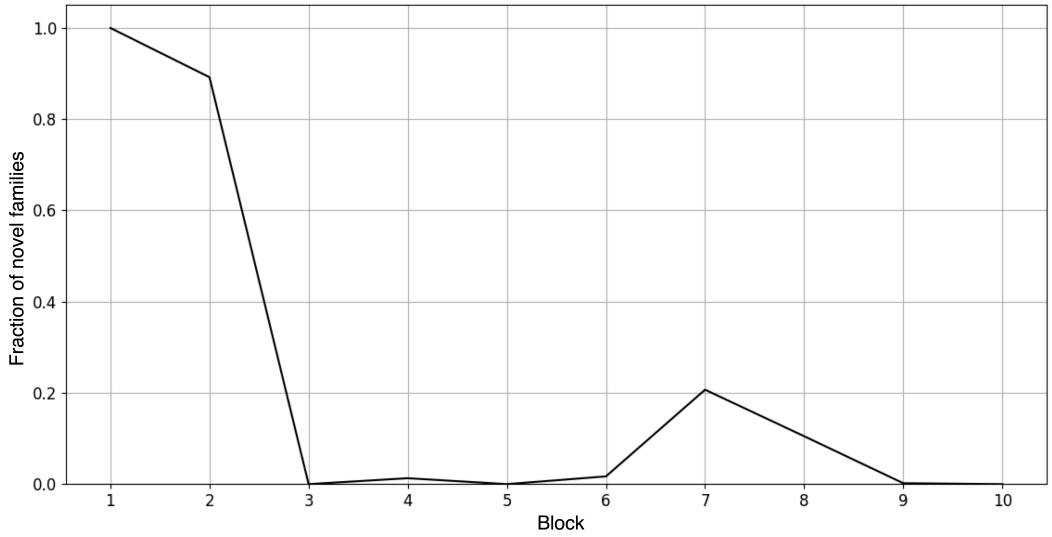


Figure 6.2. Fraction of samples of unknown families in each block.

by the clustering algorithm, with $i = 1 \dots N^C$, let n_i^c be the number of samples labelled with l_i^c. Let $n_{i,j}$ be the number of samples of class l_i^{gt} (according to the ground truth) that have been assigned to cluster l_j^c by the clustering algorithm. We first define precision and recall for a class l_i^{gt} of the ground truth with respect to a cluster l_j^c identified by the clustering algorithm:

$$Precision(l_i^{gt}, l_j^c) = n_{i,j}/n_j^c$$
$$Recall(l_i^{gt}, l_j^c) = n_{i,j}/n_i^{gt}$$

Given the total number n of samples that have been clustered, we then define the overall precision, recall and F1 score (i.e., the accuracy):

$$Precision = \sum_{l_i^{gt} \in \mathcal{L}^{GT}} \frac{n_i^{gt}}{n} \max_{l_j^c \in \mathcal{L}^C} \left\{ Precision(l_i^{gt}, l_j^c) \right\}$$

$$Recall = \sum_{l_i^{gt} \in \mathcal{L}^{GT}} \frac{n_i^{gt}}{n} \max_{l_j^c \in \mathcal{L}^C} \left\{ Recall(l_i^{gt}, l_j^c) \right\}$$

$$F1 = \frac{2 \cdot Recall \cdot Precision}{(Precision + Recall)}$$

We use these metrics also to compare the accuracies obtained with classifiers, which have the same labels of the ground truth because they have been trained on it. To make such comparison as fair as possible, we do not take into account this fact and assume the classifiers output labels in a different namespace.

Comparison with BIRCH offline. We first evaluate whether online clustering can lose in accuracy compared to total re-clustering. At this aim, we compare BIRCH online with BIRCH offline. Figure 6.3 shows that accuracies are comparable for all the blocks, this means that BIRCH online does not lose in accuracy with

respect to its offline version.

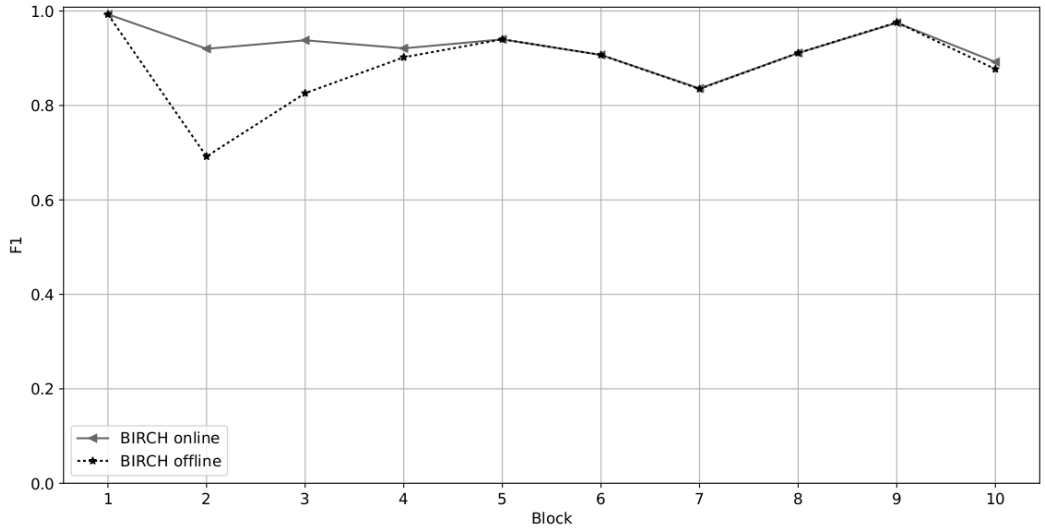


Figure 6.3. Accuracy comparison between BIRCH online and offline.

Comparison with Malheur. Figure 6.4 reports the accuracy comparison between BIRCH online, Malheur and the *incremental* version of Malheur (*Incremental Malheur*). BIRCH proves to outperform the other two algorithms in the large majority of batches. Incremental Malheur performs quite badly for blocks 8 and 9. In both cases, this is due to the presence in the immediately previous block of many malware not belonging to any known family, which are then clustered to derive the new families to add to those identified before. Incremental Malheur does not change the clusters identified previously, thus over time it may end up with an overall clustering that does not result as accurate as BIRCH or Malheur, because they instead allow for updating the clusters found so far.

Comparison with Classifiers. We compare the accuracy of BIRCH online with that achievable by using these standard classifiers: Random Forest, Support Vector Machines (SVM), Gaussian Naive-Bayes and a Multi-layer Perceptron. We trained classifiers in two distinct ways: (i) once at the beginning using the samples in the first block as training set (see Figure 6.5), and (ii) at every batch, where the training set used to analyze the i-th batch includes all the samples of blocks 1 to $i-1$ (see Figure 6.6). In both cases, BIRCH online shows an accuracy comparable with the most accurate classifiers. It is not the evident accuracy degradation for block 7, which is the one having the largest number of samples of still unknown families.

Comparison on samples of unknown families. Intending to highlight the capability of the proposed approach to properly deal with samples belonging to families that are still not known, we present a comparison between BIRCH online and the other algorithms on an ad-hoc metric computed as the product of the accuracy obtained in a certain block and the percentage of malware of still unknown families in that block. We refer to this metric as *weighted accuracy*. In this way, the accu-

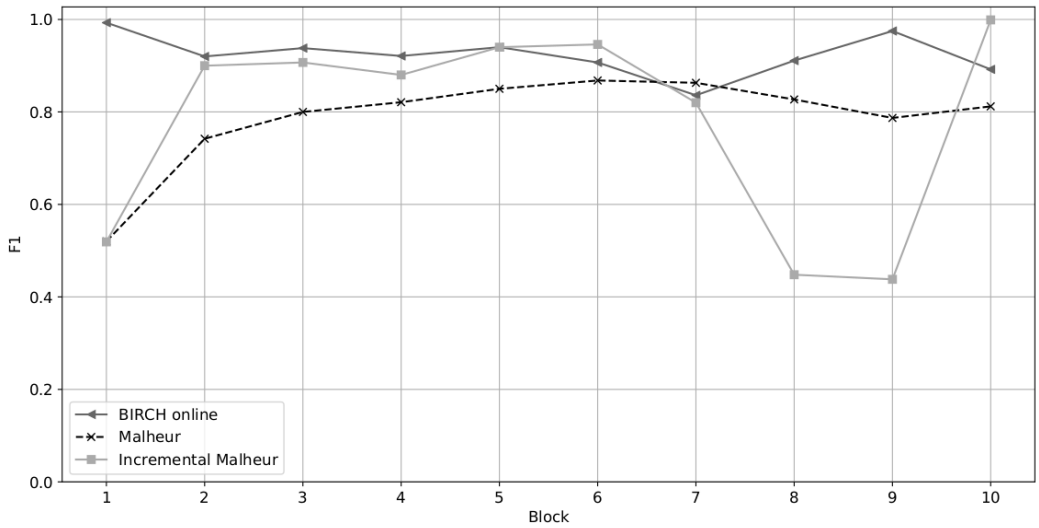


Figure 6.4. Accuracy comparison between BIRCH online, Malheur and Incremental Malheur.

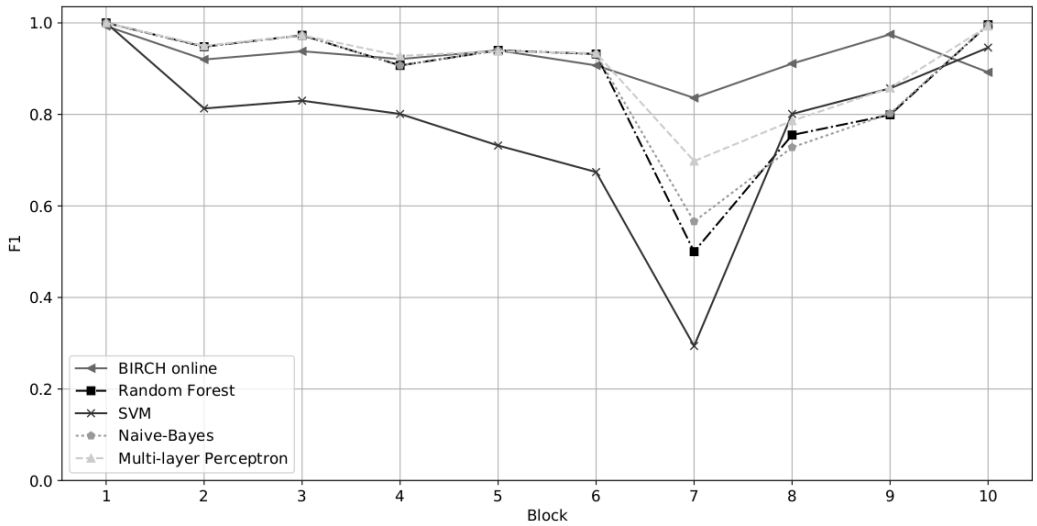


Figure 6.5. Accuracy comparison between BIRCH online and classifiers trained once on the first block.

racy values become zero when the block does not include any sample belonging to families not identified yet. Figures 6.7, 6.8 and 6.9 show that BIRCH online provides accuracy comparable to Malheur and Incremental Malheur, and outperforms classifiers, both when trained once at the beginning and when retrained at each batch.

Finally, we report in Table 6.1 precision, recall and F1 score for the tested approaches, averaged over all the ten blocks and with the value of the standard deviation. It can be observed that BIRCH online provides the highest precision and F1 score.

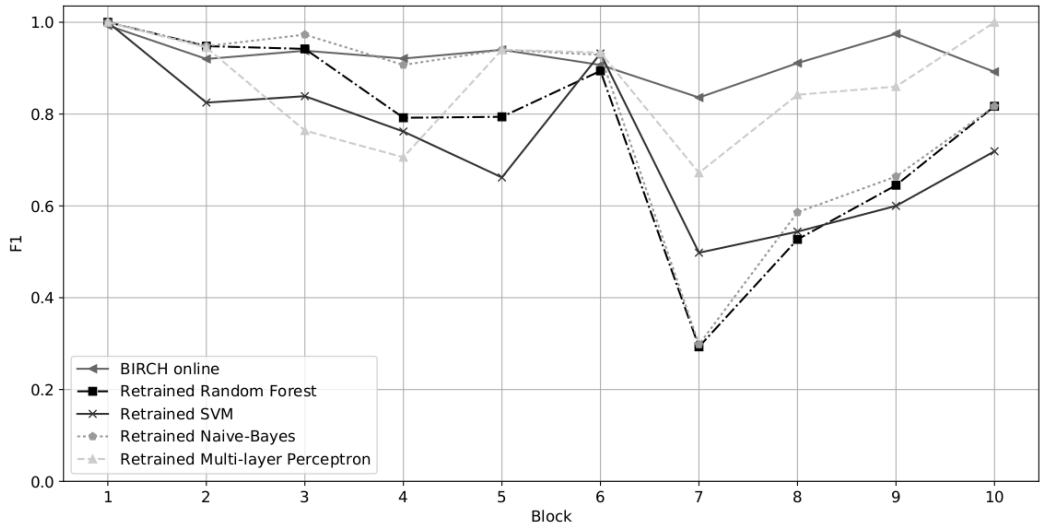


Figure 6.6. Accuracy comparison between BIRCH online and classifiers retrained at every block.

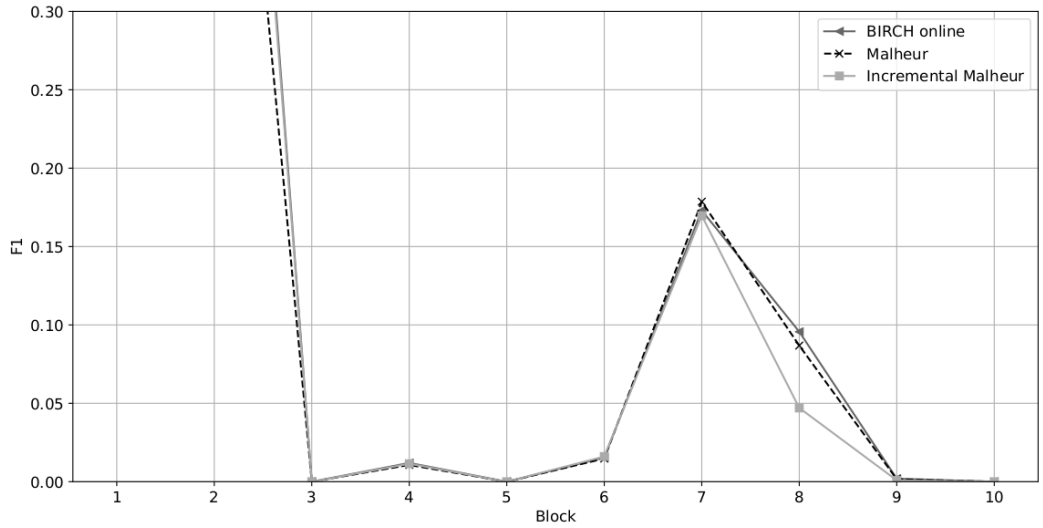


Figure 6.7. Weighted accuracy comparison between BIRCH online, Malheur and Incremental Malheur.

6.2.3 Time Performances

An important quality in the field of malware analysis is speed. In fact, malware need to be processed as fast as possible and this quality is even more important when new samples arrive periodically. The figure 6.10 shown the execution times, on a logarithmic scale, of all the approaches we took into account for each chunk of samples. As expected, re-cluster the dataset after every new chunk of malware requires even more time as the dataset grows. This behavior is exposed by Malheur and offline BIRCH. In particular, Malheur and Incremental Malheur are very slow compared to all the other solutions. Online BIRCH analyzed the incoming samples

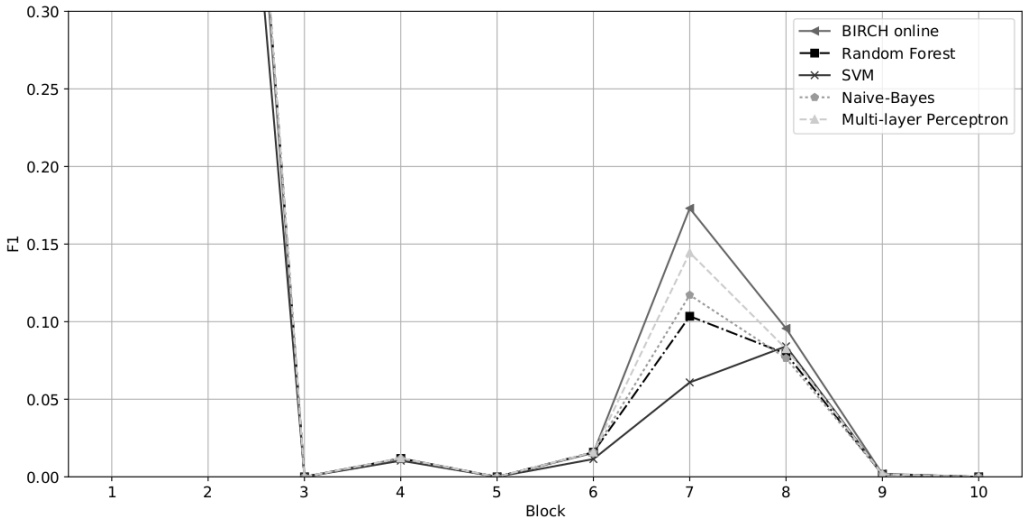


Figure 6.8. Weighted accuracy comparison between BIRCH online and classifiers trained once on the first block.

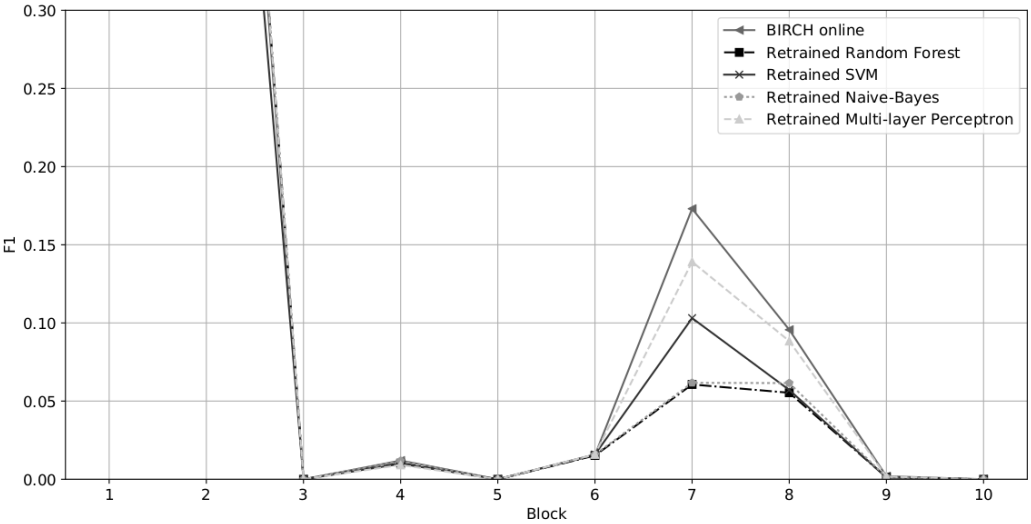


Figure 6.9. Weighted accuracy comparison between BIRCH online and classifiers retrained at every block.

with a speed comparable to the classifiers. For the first chunk, we did not report the time required to fit the model of classification algorithms, because is out of this scope. In fact, we want to highlight how much the online clustering is close to the time required for the classification. We considered the classifiers trained only on the first chunk, instead of when they are periodically re-trained because their execution times depend only by classification.

Table 6.1. Precision, Recall and F1 score for all the tested algorithms.

Algorithm	Precision	Recall	F1
BIRCH online	0.952 ± 0.043	0.902 ± 0.077	0.923 ± 0.041
BIRCH offline	0.952 ± 0.044	0.838 ± 0.130	0.886 ± 0.082
Malheur	0.941 ± 0.018	0.693 ± 0.128	0.789 ± 0.097
Incremental Malheur	0.949 ± 0.043	0.724 ± 0.285	0.780 ± 0.209
Naive-Bayes	0.789 ± 0.181	0.986 ± 0.026	0.866 ± 0.133
Retrained N-B	0.690 ± 0.254	0.997 ± 0.003	0.785 ± 0.214
Random Forest	0.783 ± 0.196	0.995 ± 0.013	0.861 ± 0.148
Retrained R.F.	0.677 ± 0.260	0.995 ± 0.013	0.739 ± 0.204
SVM	0.746 ± 0.217	0.840 ± 0.152	0.750 ± 0.176
Retrained SVM	0.701 ± 0.227	0.821 ± 0.166	0.709 ± 0.137
Multi-layer Perceptron	0.834 ± 0.130	0.977 ± 0.038	0.895 ± 0.091
Retrained M-l P.	0.833 ± 0.131	0.901 ± 0.151	0.851 ± 0.109

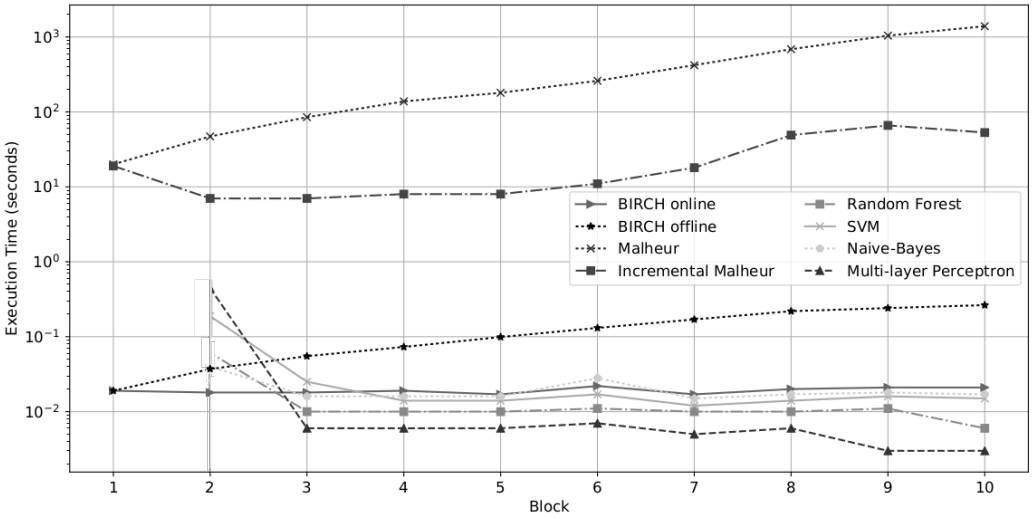


Figure 6.10. Execution time comparison for all the tested approaches.

Chapter 7

A Triage Approach for Identification of Malware Developed by Advanced Persistent Threats

As introduced in Chapter 1, the most dangerous attackers for Critical Infrastructures are Advanced Persistent Threats. For this reason, we propose a module for the architecture presented in Chapter 3 that can help to improve the analysis of APTs attacks observing the tools or malware used.

Similarly to what was explained in the previous chapters, security analysts need means of prioritizing *interesting* malware and the ones developed by APTs need to be analyzed as soon as possible since they represent a concrete danger for the CI. Thus we think that a *Triage Approach* is better to suit these needs. Like the one used in the medical field, the approach we propose focuses on providing a fast and precise result, to avoid that analysts waste their time analyzing not dangerous malware. In other words, we must focus on reducing the number of *False Positive* allowing the possibility that some malware can avoid being detected by this module. We accept this condition because there are other modules and other steps in the architecture that hopefully detect them, but in this one, we want to give an answer as fast as possible in order to dispatch risky malware immediately to the analysts.

7.1 Building Blocks

In this section we shortly describe the basic components of the module. First we show how we collect features from samples (Section 7.1.1), than we explain the classification algorithms (Section 7.1.2) which we relied on, and finally we conclude the section with details on the *features reduction* phase (Section 7.1.5).

7.1.1 Feature Extraction

Due to the time restriction, we decided to rely only on static features obtained from static analysis tools. After a *scouting phase*, we selected PEFrame and radare2[1]. The first is already introduced in Section 3.4, radare2, instead, is a portable *reversing framework* that can perform many useful operations for software exploitation, including disassembling. It allows us to extract data and compute statistics on functions and strings.

Thanks to these tools, we successfully extract more than 4000 features. We then leave classifiers and Linear Discrimination Analysis the handling of this huge number (more details in 7.1.5). These features can be roughly grouped in eight categories, here presented:

- **Optional Header** (30 features): Every file has an optional header that provides information to the loader. This header is optional in the sense that some files (specifically, object files) do not have it. For image files, this header is required. An object file can have an optional header, but generally, this header has no function in an object file except to increase its size. Features are extracted from the optional header of the PE and contain information about the logical layout of the PE file, such as the address of the entry point, the alignment of sections, and the sizes of part of the file in memory.

- **MS-DOS Header** (17 features): The MS-DOS executable-file header is composed by four distinct parts: a collection of header information (such as the signature word, the file size, etc.), a reserved section, a pointer to a Windows header (if one exists), and a stub program. MS-DOS uses the stub program to display a message if Windows has not been loaded when the user attempts to run a program. In this contest, we are interested in features related to the execution of the file, including the number of bytes in the last page of the file, the number of pages, or the starting address of the Relocation Table.

- **File Header** (18 features): The Windows executable-file header contains information that the loader requires for segmented executable files. This includes the linker version number, data specified by the linker, data specified by the resource compiler, tables of segment data, and tables of resource data. Moreover, the features related to this class highlight information about timestamp and the CPU platform which the PE is intended for.

- **String Statistics** (3 features): Binaries contains program messages stored as strings that can be useful to understand their behavior. Classical tools like *String* extract byte sequences that can be readable strings in order to find these messages. Malware authors encode strings in their program to avoid extraction, in fact, even simple schemes can defeat this kind of tools and complicate static and dynamic analysis. Using strings we compute some statistics, such as how many entry-points or relocations are present in the file.

- **Mutex** (7 features): Mutex are objects commonly used to avoid simultaneous access to a resource, like a variable. If different software checks for the same

[1] `https://rada.re/r/`

mutex, they can be linked. Our features are boolean values that map the use of particular mutex identified in the training data.

- **Packer** (64 features) Packers are software that compress binaries, keeping them executable. Similarly to the mutex related features, our features highlight if some particular packers are recognized.

- **Imported API** (3917 features): Each software imports functions from common libraries or external files. The combination of imported functions can show similar behavior. We store this information as a vector with a boolean value for each imported function, using the list of functions present in the training set as a taxonomy.

- **Buckets** (98 features): Similar size in functions and directories can be a proof of similarity in the file structure and thus in the behavior. We observe that, usually, function size values range from 0 to 1822 bytes, while directory size values range from 0 to 2638 kbytes. To track these properties, both of them are subdivided into 49 buckets. Each bucket represents a range and counts the elements whose size is in the range. To choose the different ranges we observe the distribution of sizes and lengths in the training set, trying to form buckets that can better characterize the various classes.

7.1.2 Classification

We first tried to set up the classifier using a class for each known APT, representing the malware collected on the base of APT reports, and an additional class to represent all the samples that have not been created by any known APT. If a sample is assigned to the latter class, then it would be considered not related to any known APT. Otherwise, the classifier would establish the most likely APT which developed malware similar to that sample. The problem of this approach lies in training the classifier on such additional class. Indeed, the overwhelming majority of samples belong to that class, including most of the malicious samples, only a really tiny percentage of malware has been actually created by some APTs. This translates to an excessive heterogeneity among samples of that class, and an extreme imbalance among classes in the training set, which makes this approach infeasible. To solve this problem we give up such additional classes and only use classes representing known APTs for training. To allow the classifier to work we first design a methodology based on *Multi-class classification* and *Classification Confidence* [46]. Moreover, after some studies, we have been able to improve it using, instead, a set of *One-class Classifiers* [48]. We use the Random Forest classifier as the algorithm for the first methodology, instead, we rely on the Isolation Forest classifier for the second one. The remaining part of the section will shortly describe both algorithms.

7.1.3 Random Forest Classifier

A Random Forest [9] is a supervised classification tool that aggregates the results provided by a set of *Decision Trees* through a *Bootstrap aggregated* (a.k.a. *Bagging*) technique.

In general, a *Decision Tree* [26] is a decision support tool that uses a tree-like graph as model of decisions. In machine learning science, *Decision tree learning* concept uses decision trees as a predictive model to go from observations about an object to conclusion about the objects' target value, represented in the leaves of the tree. Although single decision trees can be effective classifiers, increased accuracy often can be achieved by combining the results of a collection of decision trees. As highlighted by the name, a forest is generated by randomly ensembling different decision trees. The ensemble method for Random Forest is focused on *feature bagging* concept, which has the advantage of significantly decreasing the correlation between each decision tree and thus increasing, on average, its predictive accuracy. Feature bagging works by randomly selecting a subset of the feature dimensions at each split in the growth of the individual decision trees. Although this might sound counterproductive, since it is often desired to include as many features as possible, it has the purposes of deliberately avoiding on average very strong predictive features that lead to similar splits in trees, thereby increasing correlation. If a particular feature is strong in predicting the response value, then it will be selected for many trees.

To classify a new object from an input vector, the algorithm uses the input vector into each of the trees composing the forest. Each tree gives a classification "vote" for that class, and the forest outputs the class having the highest number of votes among all the trees in the forest.

7.1.4 Isolation Forest Classifier

In classification problems, there is the possibility to rely on **one-class classification**, where samples used in training phase belong to the same class, and in the classification phase, we use the classifier to distinguish between correct samples, i.e. samples probably related to the class, and outliers, i.e. samples not belonging to the class used for training.

Isolation Forest [55] is a one-class classifier, which data structure is built on the same conceptual principle of Random Forests. In particular, an Isolation Forest is still an ensemble of bootstrapped decision trees, but, instead of profiling normal points, it shows all possible anomalies that are *isolated* with respect to the models just created. As stated by Zhou *et al.* [55], anomalies are data patterns that have different data characteristics from normal instances. Many existing model-based approaches to anomaly detection construct a profile of normal instances, and hence isolate instances that do not conform to the normal profile.

Technically speaking, the Isolation Forest algorithm isolates observations by randomly selecting a feature and then randomly selects a split value between the minimum and maximum values assumed by the selected feature. Hence, the algorithm first constructs the separation by creating isolation trees or random decision trees; then, when they collectively produce shorter path lengths for some particular points, they are highly likely to be anomalies. Based on the average path length derived in the testing phase, it calculates the *anomaly score* for each sample. In the various online tool libraries, it is possible to set a *tolerance* threshold that discriminates the given sample as isolated or not according to the model: the higher it is, the more tolerant the Isolation Forest is with anomalies. The implementation that

we use is based on heights of trees as a metric for decisions, as proposed in [55].

7.1.5 Features Reduction

When dealing with high-dimensional features space, it is actually unfeasible to think that each feature has the same "importance". In particular, both *redundant* and *irrelevant* features exist. We have performed an initial *feature selection* basing it on the importance value given by the Random Forest algorithm to the features described in Section 7.1.1 as a reference, discovering that only 264 of them influence our data. Thus for the first methodology, we pass to the model only the important features.

Instead, for the second approach, we investigate also feature reduction techniques in order to decrease more the time required for the training phase. These techniques can be hence used to reduce the dataset dimensionality in order to facilitate the training process, reduce the variance among features, and avoid *curse of dimensionality*, i.e. with the increasing of the dimensionality, space volume increases so fast that training data become sparse.

In the second methodology, we rely on the **Linear Discriminant Analysis** (LDA) technique, because of its simplicity. LDA is a generalization of Fisher's linear discriminant [64], a method used in statistics, pattern recognition and machine learning to project the feature space in a smaller feature space through a linear combination of features. In practice, given the original dimension M of the features, it is possible to reduce the dimension to a chosen L by projecting into the linear subspace H_L maximizing inter-class variance after projection. After some experiments, we found that an optimal number of features was the number of classes minus one.

7.2 Triage Approach

As introduced in Section 7.1.2, to solve the problem of the excessive heterogeneity of non-APT samples, we first rely on a *Multi-class classification* approach managed through a *Classification Confidence* system. This approach allows us to develop a working module that achieves high results in terms of detection and identification. The main disadvantage of our first implementation is the time required for building the model: compute confidences and threshold of the forest required a high computation cost. Moreover, the *Training Phase* should be fully repeated when it is necessary to update the model. In practice, every time a new APT sample is going to enrich the classification model, there is the need for re-training the whole Random Forest, repeating all the computations. This approach can be a waste of time and resources. To overcome this weakness we decided to rely on One-Class Classification through the Isolation Forest algorithm.

Multi-class Classification

Given $C = \{c_i\}$ be the set of classes, with $N = |C|$ being the number of classes, equal to the number of actually used APTs, we train the classifier on N classes (one for each APT in the dataset). We use random forest [9] as a learning method for the classification, as it turned out to be really effective in several works related

to malware analysis [36, 41, 83], also because of its ensemble approach. Moreover, random forest permits to classify samples by using different types of features (numbers, binary and non-numeric labels). A random forest operates by generating a set of decision trees at training time and using them as classifiers by picking the most frequently chosen class among them. Let $T = \{t_j\}$ be the set of decision trees of the random forest. $N_T = |T|$ is the number of trees. In order to determine whether a sample is related to a known APTs or classified as non-APT, we rely on the confidence score of the classification: if this score exceeds a threshold, then the sample is considered as related to the relative APT, otherwise, it is not.

Classification Confidence Computation The class assigned by a decision tree depends on the leaf node where the decision path terminates. Each leaf node corresponds to a subset of its training samples, which can belong to distinct classes, and the output class of the leaf node is the most frequent one among them. For a decision tree t_j, let $l_j = \{l_{j,k}\}$ be the set of its leaf nodes, with $N_j = |l_j|$. Let $N_{j,k}$ be the number of training samples of leaf node $l_{j,k}$. We define $class_{i,j,k}$ as the number of training samples of $l_{j,k}$ that belong to class c_i. Intuitively, the diversity

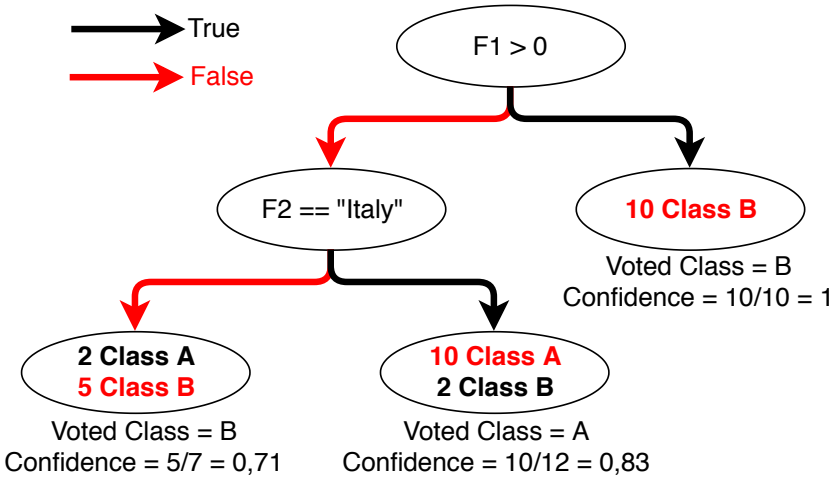


Figure 7.1. An example of how calculate leaves confidences in a Decision Tree of the Random Forest.

of classes among the training samples of a leaf node reflects how much the decision tree is confident about its output, when this output is determined by that leaf node, Figure 7.1 show an example of a decision tree in a Random Forest trained model. Thus, as a confidence score for the single decision tree, we use the percentage of training samples that belong to the same class output by the leaf node. We then assign a confidence score to the classification of the whole random forest by averaging the confidence scores of all its decision trees. In a similar way, we can assign to a classified sample a confidence score for each class, to represent to what extent that sample relates to each class. We assign a confidence vector $confidence_{j,k}$ to each leaf node $l_{j,k}$, where the i-th entry represents the confidence for class c_i, defined as

follows

$$confidence_{j,k}[i] = \frac{class_{i,j,k}}{\sum_{m=1}^{N} class_{m,j,k}} \quad i = 1 \dots N \tag{7.1}$$

For each sample to analyze, we setup the classifier to output a confidence value for each class, which represents the likelihood that the sample resembles malware created by the APT corresponding to that class. Given a sample s to classify with the random forest, we introduce the function $decision_j(s)$ which determines the leaf node of t_j where the decision path for s ends. Let $l_{j,k}$ be such leaf node, then $decision_j(s) = k$. We define the confidence vector $confidence(s)$ assigned to a sample s classified with the random forest as follows, where the i-th entry represents the confidence for class c_i

$$confidence(s)[i] = \frac{1}{N_T} \sum_{j=1}^{N_T} confidence_{j,decision_j(s)}[i] \quad i = 1 \dots N \tag{7.2}$$

Confidence Threshold Computation Malware developed by the same APT can be very different from each other. For example, they may relate to different phases of an attack (*e.g.*,the payload for intruding target system, and the remote access trojan to enable external control), or they may have been used for attacks to distinct victims. Furthermore, we empirically observe that collected malware are distributed really unevenly among known APTs. This implies that confidence scores obtained for distinct classes cannot be fairly compared. Thus, rather than using a unique confidence threshold to discriminate whether a sample can be considered as related to a known APT, we compute a different threshold for each APT.

We first compute the confidence vector for each sample of the training set TS by using *leave-one-out*-CrossValidation: for each training sample $s \in TS$, we use all the other training samples to train the random forest and then classify s to identify the leaf nodes to use to compute $confidence(s)$. Let $class(s)$ be a function that returns i if the class of training sample s is c_i. Let $TS_i = \{s \in TS : class(s) = i\}$ be the subset of the training set containing all and only the samples of class c_i. We then calculate the threshold vector as follows

$$threshold[i] = \frac{\sum_{s \in TS_i} confidence(s)[i]}{|TS_i|} - \Delta \quad i = 1 \dots N \tag{7.3}$$

For each class, rather than directly using the average of its confidence scores as threshold, we decrease it by a *tolerance band* Δ in order to avoid having too many false negatives. During the actual triage in production, a test sample s is classified by the random forest and assigned a confidence vector $confidence(s)$, which is compared to the threshold vector to check whether the following condition holds

$$\exists i \quad confidence(s)[i] > threshold[i] \quad i = 1 \dots N \tag{7.4}$$

In the positive case, s is considered related to known APTs and dispatched accordingly, together with its confidence vector which may guide the subsequent analyses, as it suggests to what extent s resembles malware developed by each of the APTs used for training the random forest.

7.2.1 One-class Classification

We decided to leverage on an Isolation Forest for every single APT class. In such a way we create a model that identifies all those elements belonging to the related APT while considering the other samples as anomalies. Whenever a new sample is identified and associated with a known APT, it is going to enforce the knowledge base and it is sufficient to re-train the related APT classifier, while the others are kept unchanged.

Isolation Forest implementation requires two main arguments to tune in the fitting phase: *contamination* expresses how much the classifier has to be tolerant in creating the model for detecting outliers; *number of estimators* is an attribute totally related to the tree-nature of the Isolation Forest and it defines how many trees it has to use in the ensemble methods. In order to tune the ⟨*contamination*, *number of estimators*⟩ tuple for achieving the best result for each Isolation Forest classifier, we performed ten-CrossValidation over the APT training set. To train our Isolation Forest classifiers, we decided to provide a non-zero value to contamination. This means that some of the training samples can be wrongly considered as outliers, but increases precision in the classification phase. Such a decision is mainly motivated by the following reasons: i) APT reports are a known unreliable source of labeled data [87] and hence we cannot be sure that any identified sample is really related to the APT; ii) malware samples have high feature variability, which can compromise the performance of the classifiers if we do not allow them to discard atypical samples in the training phase; iii) APT can leverage on masking (a non-APT sample is detected) to induce misclassification errors [86]. This is possible if an APT is going to use many different malware samples, not really necessary for its activity, only with the goal of confusing analysts and ML tools during training. This would result in many false alarms in the classification test. Introducing contamination does not completely solve the problem, but makes the system more robust. On the other side, APTs usually perform swamping (an APT sample is classified as an outlier) to hide their presence behind the samples. However our Isolation Forests are trained only on samples belonging to APTs according to public reports, hence swamping is not affecting the training. We only miss not recognized samples and similar samples during classifications.

In the choice of the correct parameters, we mainly rely on i) *Precision*, due to the malware triage nature of the proposed module. A high number of false positives is undesirable since it would increase the workload of human analysts; ii) *Accuracy*, expressing how much the APT classifier is accurate in determining the right APT class of an APT sample. The choice of the contamination-number of estimators parameters for the Isolation Forest is hence totally based on the precision-accuracy trade-off. As explained in the previous part of the Chapter, since the triage nature of the module, we have been more oriented to tune this diatribe toward precision measure. In fact, in our view, in a triage application, stating with high precision that an APT sample belongs to a given APT with high certainty, even if some APT samples are not identified, is more important than recognizing all the APT samples, but also raising false alarms related to non-APT sample that are assigned to some class.

Table 7.1. APTs elements per classes.

Class	Count
APT28	68
APT29	205
APT30	101
Carbanak	105
Desert Falcon	45
Hurricane Panda	315
Lazarus Group	58
Mirage	54
Patchwork	559
Sandworm	44
Shiqiang	31
Transparent Tribe	267
Violin Panda	23
Volatile Cedar	35
Winnti Group	176

7.2.2 Result Validation

The triage system can provide analysts hints on which malware deserve deep inspections because probably developed by a particular APT. Samples associated with some APT are immediately dispatched to human analysts, which have the tasks of validating the classifier results and performing in-depth sample analysis. We remind that our triage prefers precision to accuracy. Hence false negatives can occur. In this case, APT samples are not recognized. However this does not mean that they are excluded by further analysis, but they are not prioritized. Indeed, a high classification score can indicate that a sample deserves further inspection, even if not classified in an APT class. Scores can be a base of a further prioritization, malware with the higher score has more probabilities to be related to an APT and thus it can deserve more attention than others. The output of the triage is the label predicted by the frameworks, but, as *collateral* results, it gives also a score for each of the possible labels. These scores can be very helpful for human analysts because by knowing in advance which group is using the sample, analysts can easily retrieve information about the APT activities. Moreover, these scores can show interesting relationships that analysts or other tools can successively investigate. For example, they can prove that an APT is trying to imitate another one or even that previously separated groups are part of a single APT.

7.3 Experimental Evaluation

In this section, we compare both our approaches. We tested both of them using dAPTaset (see Section 4.2) as the reference dataset. As shown in Table 7.1 we come up with 2086 samples related to 15 distinct APTs. We have trained a Random Forest classifier on the 15 APT classes in Table 7.1 and then created a new dataset

by selecting only the classes that are recognized with Precision and Recall over 95%. This approach gives us two sets of samples, one with malware belonging to 15 APT classes and the other one with malware of 6 APT classes, the latter having higher performances. These classes are: APT28, APT30, Carbanak, Hurricane Panda, Patchwork, Transparent Tribe, so each test will require an execution for the smaller dataset and an execution for the complete one. The same 2 sets have been used to train and test the Isolation Forest classifiers. We have computed the thresholds for the Random Forest: we take the average *confidence* for each class and we decrease them by a factor Δ that we set as 5%, 10% and 15%.

Due to the huge difference in size between the positive and negative classes, we use the weighted version of *Precision* and *Recall*. *F1-Score* is computed with these values instead of the classical ones. The weighted formulas are the following:

$$WeightedPrecision = \frac{1}{\sum_{l \in L} TP_l} \sum_{l \in L} TP_l * Precision_l$$

$$WeightedRecall = \frac{1}{\sum_{l \in L} TP_l} \sum_{l \in L} TP_l * Recall_l$$

where

- L is the set of labels;
- TP_l is the set of samples that have the true label l;
- $Precision_l | Recall_l$ compute the precision or recall for samples that have the label l.

In Section 7.3.1, we evaluate the performances of the two solutions in discriminating whether a sample *belongs* or *not* to some APTs and their performances in the identification of the correct APT class, showing the achieved results for each classifier. Finally, in Section 7.3.2, we compare the execution time required by both our works. A draft of the code with the used dataset can be found on GitHub[2].

7.3.1 APT-Triage

In order to generate a sound and uniform test, we rely on *Stratified 10-Fold Validation*, which splits data into train/test sets, guaranteeing the same percentage of samples for each class. The training set is used to train an Isolation Forest for each APT class in the dataset, thus creating 6 and 15 Isolation Forests for the two identified scenarios. We built a group of classifiers for each scenario because we related the number of features produced by the LDA algorithm to the number of classes. In this way each scenario has its own number of features for the generation of the forest, thus requesting two separate training phases. For the first approach, we consider a sample as *non-APT* if all the scores given by the module are lower than the relative thresholds. The second approach recognizes a sample to belong to some APT if at least one Isolation Forest recognizes it in its class. Otherwise, a malware is classified as *non-APT*, if no classifier recognizes it. We performed a

[2] `https://github.com/GiuseppeLaurenza/I_F_Identifier`

preliminary test of Random Forest solution with a set of more than 800 binaries not belonging to Advanced Persistent Threats. Then we validated the second solution based on Isolation Forests on a larger set of non-APT samples containing 9000 binaries. While APT samples are used in both training and classification in our ten-CrossValidation (hence each sample is used only once in classification), non APT-samples have been used only to evaluate all the 10 trained models.

Table 7.2. APT-Triage Confusion Matrix with 6 APTs.

One Class Classifier		*Predicted*	
		APT	non-APT
Real	APT	1173	242
	non-APT	69	90141
Random Forest ($\Delta = 5\%$)		*Predicted*	
		APT	non-APT
Real	APT	1227	164
	non-APT	12	8618
Random Forest ($\Delta = 10\%$)		*Predicted*	
		APT	non-APT
Real	APT	1261	130
	non-APT	25	8605
Random Forest ($\Delta = 15\%$)		*Predicted*	
		APT	non-APT
Real	APT	1272	119
	non-APT	100	8530

Table 7.3. APT-Triage quality measures with 6 APTs.

	Accuracy	**F1**	**Precision**	**Recall**
One Class Classifier	0.9966	0.9966	0.9965	0.9966
Random Forest ($\Delta = 5\%$)	0.9824	0.9825	0.9826	0.9824
Random Forest ($\Delta = 10\%$)	0.9845	0.9845	0.9845	0.9845
Random Forest ($\Delta = 15\%$)	0.9781	0.9781	0.9780	0.9781

Tables 7.2 and 7.3 show details about the result of the first experiment in the 6 APTs scenario, while Tables 7.4 and 7.5 in the 15 APTs one. Results obtained by the second approach are slightly better than the ones achieved by the first protocol, in fact, we correctly detect most of the malware developed by APTs, achieving a precision of over 99%. Even if the second solution detects a bit less APT malware than the Random Forest module, it strongly reduced the number of false-positive cases, cutting more than half of them in percentage. As described in previous sections, a triage approach must focus on lowering the false detection rate in order to not waste analysts' effort. Hence, this small loss in accuracy can be well accepted if it implies a heavy reduction of *false alarms* and, thus, of time loss. However, the results of these tests show improvements in all the measures, confirming the goodness of the second solution.

Table 7.4. APT-Triage Confusion Matrix with 15 APTs.

One Class Classifier		*Predicted*	
		APT	non-APT
Real	APT	1756	330
	non-APT	685	89525
Random Forest ($\Delta = 5\%$)		*Predicted*	
		APT	non-APT
Real	APT	1759	327
	non-APT	45	8585
Random Forest ($\Delta = 10\%$)		*Predicted*	
		APT	non-APT
Real	APT	1803	283
	non-APT	60	8570
Random Forest ($\Delta = 15\%$)		*Predicted*	
		APT	non-APT
Real	APT	1831	255
	non-APT	77	8553

Table 7.5. APT-Triage quality measures with 15 APTs.

	Accuracy	**F1**	**Precision**	**Recall**
One Class Classifier	0.9890	0.9895	0.9901	0.9890
Random Forest ($\Delta = 5\%$)	0.9653	0.9654	0.9656	0.9653
Random Forest ($\Delta = 10\%$)	0.9680	0.9680	0.9680	0.9680
Random Forest ($\Delta = 15\%$)	0.9690	0.9689	0.9688	0.9690

Table 7.6. APT-Identification Confusion Matrix with 6 APTs.

One Class Classifier		*Predicted*	
		Correct APT	Other APTs
Real	Correct APT	1171	2
	Other APTs	2	5865
Random Forest ($\Delta = 5\%$)		*Predicted*	
		Correct APT	Other APTs
Real	Correct APT	1224	3
	Other APTs	3	6132
Random Forest ($\Delta = 10\%$)		*Predicted*	
		Correct APT	Other APTs
Real	Correct APT	1258	3
	Other APTs	3	6302
Random Forest ($\Delta = 15\%$)		*Predicted*	
		Correct APT	Other APTs
Real	Correct APT	1269	3
	Other APTs	3	6357

Table 7.7. Detailed APT-Identification Confusion Matrix of our Isolation Forest based triage with 6 APTs.

		Predicted					
		APT28	**APT30**	**Carbanak**	**Hurricane Panda**	**Patchwork**	**Transparent Tribe**
Real	**APT28**	44	0	0	0	0	0
	APT30	0	81	0	0	0	0
	Carbanak	0	0	97	0	0	0
	Hurricane Panda	0	0	0	293	0	2
	Patchwork	0	0	0	0	408	0
	Transparent Tribe	0	0	0	0	0	248

Table 7.8. APT-Identification quality measures with 6 APTs.

	Precision	**Recall**	**Accuracy**	**F1**
One Class Classifier	0.9994	0.9994	0.9994	0.9994
Random Forest ($\Delta = 5\%$)	0.9992	0.9992	0.9992	0.9992
Random Forest ($\Delta = 10\%$)	0.9992	0.9992	0.9992	0.9992
Random Forest ($\Delta = 15\%$)	0.9992	0.9992	0.9992	0.9992

Table 7.9. APT-Identification Confusion Matrix with 15 APTs.

One Class Classifier		*Predicted*	
		Correct APT	Other APTs
Real	Correct APT	1731	56
	Other APTs	25	25018
Random Forest ($\Delta = 5\%$)		*Predicted*	
		Correct APT	Other APTs
Real	Correct APT	1753	6
	Other APTs	6	24620
Random Forest ($\Delta = 10\%$)		*Predicted*	
		Correct APT	Other APTs
Real	Correct APT	1797	6
	Other APTs	6	25236
Random Forest ($\Delta = 15\%$)		*Predicted*	
		Correct APT	Other APTs
Real	Correct APT	1822	9
	Other APTs	9	25625

Table 7.10. APT-Identification quality measures with 15 APTs.

	Precision	**Recall**	**Accuracy**	**F1**
One Class Classifier	0.9970	0.9970	0.9970	0.9970
Random Forest ($\Delta = 5\%$)	0.9995	0.9995	0.9995	0.9995
Random Forest ($\Delta = 10\%$)	0.9996	0.9996	0.9996	0.9996
Random Forest ($\Delta = 15\%$)	0.9993	0.9993	0.9993	0.9993

Tables 7.6 and 7.8 show the results of the second experiment in the 6 APTs scenario, while Table 7.7 presents the detailed confusion matrix for this scenario. Tables 7.9 and 7.10 contain results of the second experiment in the 15 APTs scenario.

Table 7.11. Execution Time with 6 APTs in seconds.

	One Class Classifier	Random Forest
Feature Reduction (LDA)	0.1729 ± 0.0004	0,0000 ± 0,0000
Training Phase	1.6948 ± 0.3495	1031.9726 ± 9.1574
Prediction Phase	3.7564 ± 0.2253	7.0076 ± 0.0012

Table 7.12. Execution Time with 15 APTs in seconds.

	One Class Classifier	Random Forest
Feature Reduction (LDA)	0.2129 ± 0.0005	0,0000 ± 0,0000
Training Phase	1.6179 ± 0.0111	1600.7085 ± 25.1686
Prediction Phase	4.7240 ± 0.1566	8.4778 ± 0.,3841

Again, in this experiment Isolation Forest module achieves results comparable with the Random Forest one. The small number of APTs misclassified as other ones in the second experiment is related to the structure of the second module. In fact, using different classifiers, more than one classifier may label the same sample as belonging to the relative APT. As introduced in Section 7.2.2, our framework can provide a *score vector*, thus this kind of error can be probably reduced by evaluating it before making a decision. Indeed, this number is small enough to be negligible, in fact, the average metrics are only less than 0,002% lesser than the ones obtained by the first solution. Moreover, the assignment of a sample to more classes can highlight some relations between APTs that an analyst can deepen.

Summarizing the result of all tests, with achieving high results with both solutions, the second one is comparable to the first but it heavily reduces the number of *False Positive* that is the main objective of the improvement.

7.3.2 Execution Time

Here, we compare the runtime of the training and the evaluation phases for both Random Forests and Isolation Forests. In our tests, we have measured the time of 10 executions and computed the average time.

Tables 7.11 and 7.12 respectively show execution time in seconds with 6 and 15 APTs. Tables also include the training time required by the LDA algorithm to generate the LDA transformation, based on the training set. On the other side, LDA runtime for feature reduction in classification is not provided because negligible. We underline that the configuration of the LDA transformation is performed only at the first training and repeated when we need to re-train all the Isolation Forests (because precision and accuracy significantly decreased after the introduction of several new samples) or when we need to add another APT. In both training and prediction, our solution is faster than the previous one, especially in the training phase. In fact, training of a set of Isolation Forests is orders of magnitude lower than the training of a Random Forest and computing the various thresholds. Moreover, we remind that the training time of the set of classifiers for the second solution is the sum of the time required to build each single Isolation Forest; thus when there is the need to retrain only a single class, for example after collecting new samples developed by a particular APT, the time is only a fraction of that total time. Exactly, our solution requires an average of 0.3 seconds to build an Isolation

Forest related to a particular APT.

7.4 Considerations

Our solution can strongly help security experts to improve Critical Infrastructures security. In fact, it is possible to cut the number of malware requiring deep inspections and, at the same time, quickly detecting the APT-related ones, that can be the most dangerous. Moreover, detecting as soon as possible the malware developed by an APT allows a quick reaction and thus a faster recovery. In addition, identifying also who probably is the author of the attack can give information on how the threat will proceed, thus simplifying recovery procedures to block the attack and enhance the defenses.

However, one weakness of our approach is the one shared by many techniques based on classification algorithm: an adversary can find a way to *trick* the model and evade the detection, in an *adversarial machine learning* scenario[3]. APTs, in fact, usually deeply study the target to identify weakness in its defenses, thus they can also try to study classification models.

There are two main ways to attack our module, one *passive* and one *active*. The first one consists of observing how the module reacts to some modifications of particular characteristics of the analyzed malware. In this way, it is possible to find the most *discriminant* features for a particular binary, thus an attacker can alter the code in order to obtain a malware that can pass undetected. Moreover an APT might want to attack a target not only for damaging it but also pretending to be someone else in order to draw backlash against another *entity*. Instead, the active method to attack our module is to continually try to infect the target that is using it, but using especially *forged* malware with some characteristic that can lead the model to wrongly learn the main features of the APT-related software. Attackers need to deeply study the classification model in order to understand how it works. Then, they can develop malware that, when added in the training set, cause a divergence in the model thus the module misclassifies following suspicious software. Moreover, similar to the previous approach, they can *corrupt* the model in two different ways, they can develop new malware that remain undetected or that will be classified as part of another group.

This kind of attack must be taken into consideration when talking about the security of complex infrastructures, especially in the Critical Infrastructures scenario, when the adversaries could be APTs, ones of the most skilled and dangerous cyber threats. As explained in other chapters of this thesis, we do not consider our approach as the unique defense to use against APTs activity, but only a module of a more complex system, such as the architecture presented in Chapter 3. This module can improve the security of a CI reducing the number of malware requiring manual analysis and, at the same time, highlighting the ones that need quick attention. But, for this reason, it is designed to be fast and precise; thus it should be used in combination with other systems that perform more slower analysis and checks all the malware in order to find if there is something that is trying to evade

[3]To the best of our knowledge, adversarial machine learning to evade detection has not been applied by APTs so far. However, this could be a serious problem to be faced in the near future.

the detection.

Moreover, the results of the following modules in the architecture can be used to improve this one. In fact, experts need to observe why the module misclassified something, thus they can identify adversaries that are attacking the module studying in detail the reasons behind the classification and looking for strange behaviors.

Chapter 8

APTs Identification and Detection Through Multi-Step Approach

In the previous chapter, we presented our proposal to prioritize the suspicious software that must be manually analyzed by security experts. It is a triage module for detecting and identifying malware developed by Advanced Persistent Threats. It is based on a Random Forest Classifier and it achieves very good results in terms of prediction quality This solution has two main weaknesses. The first one is that it requires a full re-training each time we need to add a newly discovered APT or a new sample to a known APT. The second is the high computational time required for the training of the module. For this reason, in the same chapter, we also presented an enhanced version of this module. The idea is to build a model for each APT, using a One-Class classifier. In this way, it is possible to update only a module with new related samples or to add a module when a new group is recognized without a full re-train. The limit of this new solution is related to the *Feature Reduction* phase, in which we have to rely on *Linear Discriminant Analysis*. The problem of using LDA is that each time we need to find the best number of features, in order to perform better results and it is not a trivial task.

The improved version achieves results similar to the first approach, but significantly reduce the training time required to build and update the model. Despite the high precision obtained, both versions cannot detect all the APT-related malware, collecting a small but not negligible number of *False Negative*. We hence aim to further improve the accuracy and the precision of our triage step. In the previous approaches, we perform directly *APTs Identification*. We train one or more classifiers to recognize if a malware was developed by one of the known groups. In other words, we can consider the *APTs Detection* phase as a *side effects* of Identification. If we cannot recognize the Advanced Persistent Threat behind a malicious software, then we consider it is not developed by any APTs.

In our opinion, performing the two analyses in two distinct components, working one after another, can largely improve the quality of the prediction[1]. The Triage module has a focus on lowering *False Positives*, which can cause a raising of *False*

[1]The content of this chapter is novel and will be submitted soon to some security conference.

Negatives. Making two different systems allow us to tune them differently, for example having the Detection system that achieves both low False Positives and False Negatives and the Identification one tuned to obtain quite zero False Positive. In this way, we can detect almost all the APT-related malware that will be analyzed soon, then we can make a second *prioritization*, providing to malware analysts the ones we can identify the author. Moreover, malware detected but not identified, are *highlighted* from normal malware, thus they should be analyzed later. If analysts confirm that they are False Positive of the first phase, they are discarded. Otherwise, these malicious software are probably developed by a known or unknown APT, thus they still deserve special attention. In both cases, we can use the samples to better tune the model.

Taking inspiration from the Medical Triage, we can add a color schema to the combined prediction of the two components. Malware detected as APT-related and subsequently identified as developed by a known APT, receive the *red* color. Similarly to red marked patients, they need to be analyzed by security experts as soon as possible. Instead, malware detected as APT-malware but not identified as developed by any of the known APTs, receive a *yellow* color. It indicates the requirement for observation, in fact, experts should analyze them to understand if they are False Positive cases, part of an unknown APT, or not correctly assigned to an APT. However, due to this doubtful situation, they are less urgent than the red ones. Finally, malware detected as non-APT receive a *green* color, indicating that they do not need any prioritization and malware experts should focus on analyzing other suspicious software.

8.1 Feature Exctraction

In this enhanced version, we both improve the way we analyze malicious binaries and feature set extraction.

As shown in Section 7.1.5, most of our features come from PEFrame analysis. We have reduced the original 4000 features to 264 in the first approach and used LDA to further decrease this number in the second approach. PEFrame was recently updated with many modifications and improvements. The new version shows that it is possible to extract a smaller number of features maintaining the quality of representation. In addition, we also replace radare2 with another tool. radare2 offers several useful operations for software exploitation, including disassembling. However, this tool requires time to perform all these operations, but we are interested only in discovering binary functions and strings to build our features, so we should find a more specific tool that can obtain the same or better results in less time. For the strings part, we rely on the new PEFrame version, which can collect the same data of radare2 during its very fast analysis. The function analysis, instead, is built on Nucleus[2], which implements a novel way to find functions inside binaries[3] that is faster and more effective than radare2 method. Using these tools, we are able to collect 623 features. Table 8.1 show details about the distribution. Some of them are already described in Section 7.1.1, hence, here we report only new or modified features.

[2] `https://bitbucket.org/vusec/nucleus.git`

Table 8.1. Features distribution in categories.

Category	Features Count
Optional Header	22
MS_DOS Header	17
File Header	18
String Statistics	3
Mutex	7
Imported DLL	88
Buckets	50
Packer	159
Anti Debugger	19
Anti Virtual Machines	3
Behavior	46
Breakpoints	192

- **Buckets**: We discover that information on directory sizes are not relevant during classification, so, now, we only compute statistics on function sizes.
- **Packer**: This class of features is used also in previous protocols. However, thanks to the updated version of PEFrame, this time we also track the use of particular libraries or functions that are commonly used by specific packers.
- **Anti Debugger**: Many malicious software, tried to recognize the presence of a running debugger to find if they are currently being analyzed through a debugger. This information is fundamental for malicious softwares. If they suspect they are running in an analysis environment, they can modify their behavior to hide the malicious intent. In order to detect this pattern, we store the presence of functions that can be used for this scope. Different combinations of functions can be a "typical pattern" of a particular group.
- **Anti Virtual Machines**: Similarly to Anti Debugger features, malicious software can use some *tricks* to detect if they are currently running in a virtual environment. We track the use of particular functions used for this scope.
- **Behavior**: As introduced at the beginning of the Section, the new version of PEFrame comes with various improvements. One of them is the addition of a group of YARA rules[3]. Yara is a tool for helping malware researchers to identify and classify malware samples based on textual or binary patterns. PEFrame adopts rules to understand some actions that the malware performs, like whether it spreads files on the disk or performs screenshot. Thus it can create a brief description of its behavior. Pattern of actions can be a distinguishing characteristic, so we track with a boolean vector the presence of these actions.
- **Breakpoints**: We track the presence of particular functions that can cause a breakpoint during the execution of the malware. Like other features, the

[3] `https://github.com/VirusTotal/yara`

particular combinations of functions can be a distinguish pattern. Some functions can be also detected as Mutex, Anti Debugger or Anti Virtual Machine features. We decide to consider them only in the more specific features category.

8.2 Methodology Details

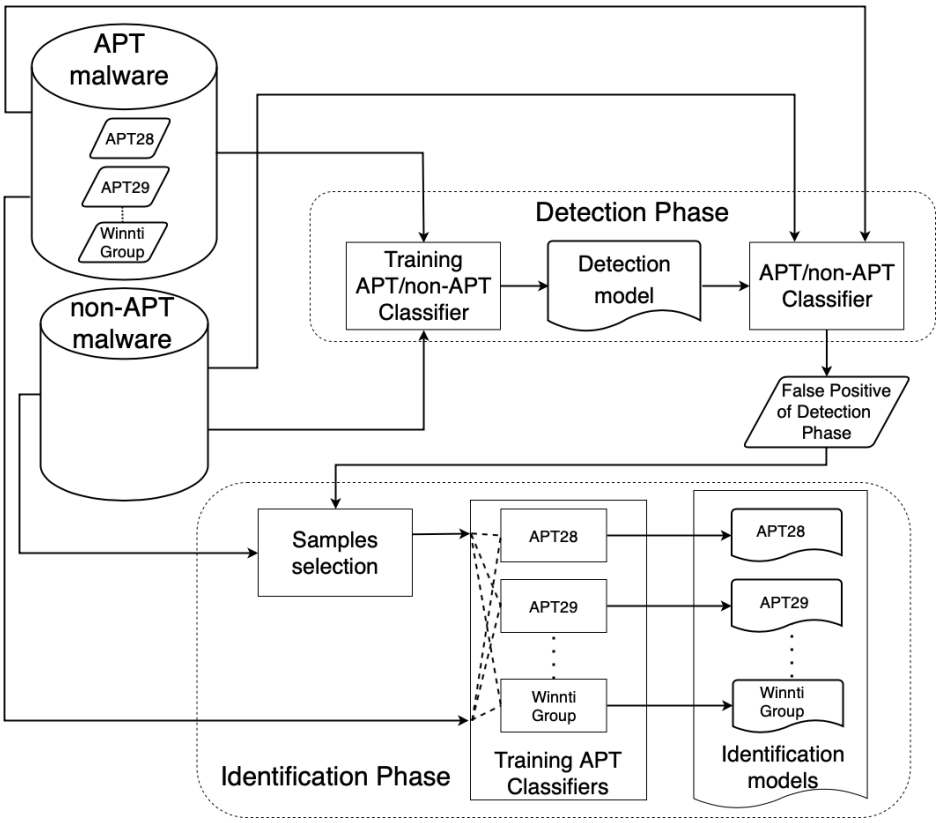
As introduced at the beginning of the Chapter, we design a multi-step approach to separate the Detection Phase from the Identification Phase. Figure 8.1 shows a schema of the proposed methodology.

To perform APT-related malware detection, we rely on a unique binary classifier. This classifier needs to be trained considering all the various APTs as a unique large class (**APT malware**), in opposition to the class that contains all the *normal* malware (**non-APT malware**). Malware identification is made by a set of binary classifiers, each one trained to recognize a specific APT. For each model, we consider the dataset divided as malware belonging to the target APT and malware belonging to other APTs. In the following part of the Section, we explain the Training Stage and the Prediction Stage more in detail.

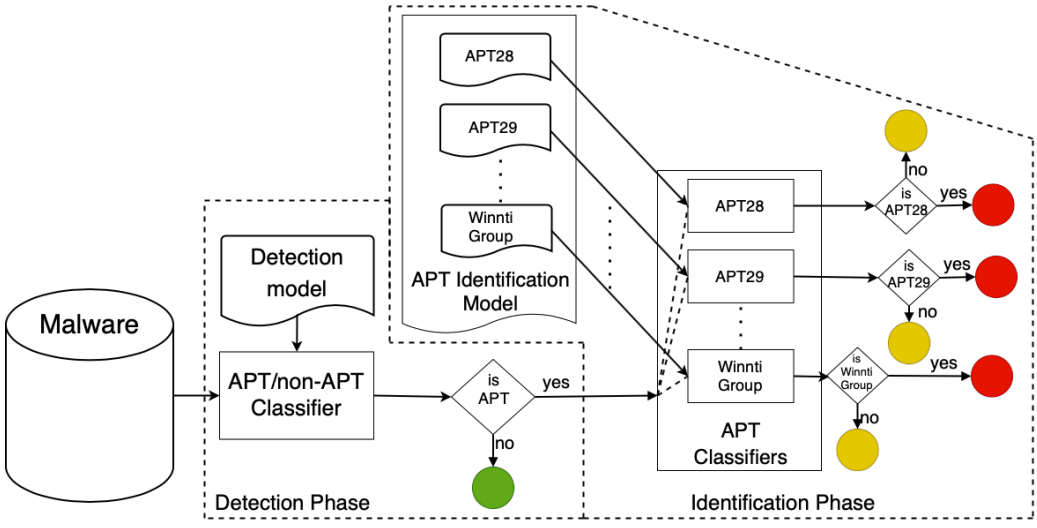
8.2.1 Training Stage

Detection Phase

We trained a single binary classifier to distinguish between APT malware (*Positive* class) and non-APT malware (*Negative* class). We consider all malware developed by an APT as a unique *Positive* class. Classification algorithms usually require some hyperparameters to improve their performances, thus we perform a tuning phase before the real training. We perform a ten-CrossValidation to evaluate each configuration and select the one that provides the highest value of *Precision* and *Accuracy*, but with the lowest as possible number of False Negatives. As we previously discussed, in the Detection phase we can give up a small percentage of precision in exchange to detect most APT malware as possible. Obviously, this drop of precision has to be very small to avoid to lose the real goal of the entire Triage approach. After the tests, we build a model with the selected configuration of hyperparameters. We also analyze all the malware used for the training of the model to observe how they are detected. This analysis on something that we already know, has the scope of improving both Detection and Identification phases. Malware misrecognized by the Detection phase can be undetected APT malware or wrongly detected non-APT ones. For the first case, we want to discover which are these malware and why they are undetected. This could lead to interesting discoveries, for example, many of them could be part of the same APT. Therefore misclassified malware need specialized attention. For the latter case, instead, we want to discover which are the characteristics of these malware that mislead the classifier and use them for the training of the models of Identifications phases. We hence desire that in the identification phase, the models are also able to place in the negative class all the non-APT malware that can be wrongly recognized in the first step.



(a) Training Phase



(b) Prediction Phase

Figure 8.1. Schema of the proposed methodology.

Identification Phase

We build a set of classifiers, each one trained to recognize only a specific APT. Each model is trained considering the malware of the relative APT as the positive

class and malware of other APTs as the *Negative* class. Moreover, we include in the negative class all the false positives of the Detection phase. In this way, these models can learn to distinguish the malware developed by the current APT from the ones developed by the other groups and also from the ones that are similar to APT-related malware but that are normal malware. Our idea is that this *shrewdness* can improve the Identification of known APTs. We tune each classifier with the same method of the previous stage. Each time, we use a three-CrossValidation method to test all the possible combinations of arguments. Then, we trained the model with the selected arguments, in this case, we choose the combination that provides the highest value of *Precision* and *Accuracy*, quite ignoring the number of False Negatives. We take this decision because malware at this phase are all *highlighted* by the system (in this step they can receive only *yellow* and *red* color), therefore we want to avoid any doubt when giving additional information to security experts.

8.2.2 Prediction Stage

Malicious software pass through the first step classifier that tries to find any relationship with APTs, from their features. Each time the model labels an APT-malware, it marks the malicious software to be further analyzed in the Identification phase. Then, malware is analyzed by the various identification classifiers. If almost one of them finds a hit, the malware is sent to security experts as soon as possible. More classifiers may recognize the malware as developed by their APTs. In this case, malware analysts receive malicious software with all the assigned labels. Further analysis can confirm which is the real author and malware is added to the training set of all identification classifiers, one time as positive class and the other times as negative class. As previously explained, only malware that are not been recognized by the Detection phase receive a green mark. If they are labeled as related to specific APTs by classifiers in the identification step, they receive high priority (red mark) for further analysis. If no model of the Identification phase recognizes them, they are not discarded, but they receive a yellow mark, stating that they require attention by experts. However its priority should be lower than the identified ones.

8.3 Experimental Evaluation

We perform our tests with dAPTaset (see Section 4.2) as the reference dataset. In Section 7.3 there are more details about the composition.

For the test, we perform a ten-CrossValidation. Each time we use 9/10 of the malware for training (**Training set**) and the remaining for test (**Test set**). As explained in Section 8.2, we tune each algorithm using only malware of Training set to choose the hyperparameters. Then, we test each classifier with the selected arguments. The remaining part of this Section is divided as following: in Section 8.3.1 we detail the algorithms used in the test, Section 8.3.2 focuses on the tuning of the algorithms, and finally in Section 8.3.3 we present and discuss the results obtained in each phase.

8.3.1 Classification Algorithms

We tested our methodology using Random Forest, Support Vector Machines (SVM)[14] and eXtreme Gradient Boosting (XGBoost)[17] in both phases. We also tested the Isolation Forest for the Identification Phase, as we observed that an anomaly detection algorithm is not suitable for the detection phase. Random Forest and Isolation Forest are already presented in Section 7.1.2. In the following, we provide a small explanation for the other two.

Support Vector Machine (SVM) Classifier

A Support Vector Machine (SVM) is a supervised machine learning algorithm that can be employed for both classification and regression purposes. SVMs are more commonly used in classification problems and as such, this is what we used it for. SVM implements the idea of mapping the input vector x into a high-dimensional feature space Z through some nonlinear mapping, chosen *a priori*. In this space, it is constructed an *Optimal* separating hyperplane.

Intuitively, a good choice is the hyperplane that leaves the maximum margin between classes, where the margin is defined as the sum of the distances of the hyperplane from the closest point of the groups. If the classes are non-separable, it is still possible to look for the hyperplane that minimizes the misclassifications errors.

Support vectors are the data points nearest to the hyperplane, the points of a dataset that, if removed, would alter the position of the dividing hyperplane. Because of this, they can be considered the critical elements of a data set. These data points are the ones that lie at the borders among the classes. Their number is usually small, and it is proportional to the generalization error of the classifier. This algorithm was initially developed for linear classification. It was extended through the *kernel trick* to use different functions to model nonlinear hyperplanes. In practice, these functions are the transformations of the space to better highlight the separation of samples in classes.

eXtreme Gradient Boosting (XGBoost) Classifier

XGBoost is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient boosting framework. In prediction problems involving unstructured data (images, text, etc.) artificial neural networks tend to outperform all other algorithms or frameworks. However, when it comes to small-to-medium structured/tabular data, decision tree-based algorithms can perform very well.

In boosting, each new tree is a fit on a modified version of the original data set. It begins by training a decision tree in which each observation is assigned an equal weight. After evaluating the first tree, the algorithm increases the weights of those observations that are difficult to classify and lower the weights for those that are easy to classify. The second tree is therefore grown on this weighted data. Thus the new model is composed of the combination of two trees. Then it is possible to compute the classification error from this new 2-tree ensemble model and grow a third tree to predict the revised residuals. Subsequent trees can classify observations that are not well classified by the previous trees. Predictions of the final ensemble

model are therefore the weighted sum of the predictions made by the previous tree models.

The great novels of XGBoost respect to other boosting algorithms are that it is sparsity awareness and it employs the distributed weighted *Quantile Sketch algorithm* to effectively find the optimal split points among weighted datasets. Moreover, it comes with a built-in CrossValidation method at each iteration, taking away the need to explicitly program this search and to specify the exact number of boosting iterations required in a single run.

8.3.2 Tuning Algorithms

A model hyperparameter is a characteristic of a model that is external to the model and whose value cannot be estimated from data. The value of the hyperparameter has to be set before the learning process begins. In contrast, a parameter is an internal characteristic of the model and its value can be estimated from data. Therefore, we can tune hyperparameters to improve classification results.

There are different methodologies to perform this search, the most famous are *Exhaustive Grid Search* and *Parameter Randomization*. The first consists of exhaustively generating candidates from a grid of hyperparameter values, the latter instead implements a randomized search over hyperparameters, where each setting is sampled from a distribution over possible hyperparameter values. For this work, we have chosen to use a grid search. We used a three-CrossValidation each time we need to perform a search, dividing the training set and using each time one group as validation test. Then we use the selected hyperparameters to perform the real training of the model, using the entire training set for the operation.

8.3.3 Experimental Results

We first discuss the results obtained in the Detection Phase, then we analyze the result of the Identification Phase. Finally, we show the performance in time of the approach and last we summarize the results.

APT Detection

Table 8.2 shows the confusion matrices of the three classifiers. Table 8.3, instead, contains the weighted quality metrics (see Section 7.3 for details on the formulas).

Looking at the various metrics, it appears that all algorithms perform well, with Random Forest that achieves a better result. However, looking at the confusion matrices, it becomes clear that Random Forest outperforms the other two, especially regarding the number of the *False Negatives*. Due to this huge difference, we report the Identification phase only on the malware detected by Random Forest.

Our detection model based on the Random Forest algorithm also performs better than our previous works, achieving a smaller number of false positives.

Table 8.2. Confusion Matrices for the APT Detection Phase.

XGBoost		*Predicted*	
		APT	non-APT
Real	APT	1074	1012
	non-APT	48	20119

SVM		*Predicted*	
		APT	non-APT
Real	APT	651	1435
	non-APT	0	20167

Random Forest		*Predicted*	
		APT	non-APT
Real	APT	2080	6
	non-APT	0	20167

Table 8.3. Quality metrics of the Detection phase test.

	Accuracy	**F1**	**Precision**	**Recall**
XGBoost	0.9524	0.9525	0.9526	0.9524
SVM	0.9355	0.9377	0.9398	0.9355
Random Forest	0.9997	0.9997	0.9997	0.9997

Table 8.4. Quality metrics of Identification phase test.

	Accuracy	**F1**	**Precision**	**Recall**
Isolation Forest	0.079	0.132	0.841	0.079
Random Forest	0.992	0.992	0.992	0.992
SVM	0.955	0.952	0.955	0.955
XGBoost	0.975	0.972	0.971	0.975

APT Identification

Malware of the Test set detected as APT malware go trough the Identification phase. We evaluate the four classification algorithms by training a classifier for each algorithm and for each APT group.

As shown in Table 8.4, Random Forest is the one that performs better also for this phase. It outperforms Isolation Forest and achieves slightly better results than SVM and XGBoost. Moreover, looking in details the prediction made by all the classifiers, we notice that Random Forest is the one with more true positive among the *best* three, with a very small number of false positives. In addition, it is the one with the smallest number of false negatives. These results confirm that Random Forest is the best choice also for this second phase. Tables 8.5 and 8.6 give details

Table 8.5. Results of Random Forest classifiers.

APT	True Positive	False Negative	False Positive	True Negative
APT28	57	11	0	2012
APT29	182	23	1	1874
APT30	85	16	0	1979
Carbanak	78	27	0	1975
Desert Falcon	30	15	0	2035
Hurricane Panda	291	24	0	1765
Lazarus Group	46	12	0	2022
Mirage	42	12	0	2026
Patchwork	528	31	8	1513
Sandworm	38	6	0	2036
Shiqiang	23	8	0	2049
Transparent Tribe	243	18	5	1814
Violin Panda	16	7	3	2054
Volatile Cedar	32	3	0	2045
Winnti Group	163	13	1	1903

of the results of Random Forest classifiers, showing that it performs very well in the identification of all the APTs and that it is comparable to our other approaches.

Table 8.6. Quality performance of the best classifier for each APT.

APT	Accuracy	F1	Precision	Recall
APT28	0.9947	0.9947	0.9947	0.9947
APT29	0.9885	0.9885	0.9885	0.9885
APT30	0.9923	0.9923	0.9924	0.9923
Carbanak	0.9870	0.9871	0.9872	0.9870
Desert Falcon	0.9928	0.9928	0.9928	0.9928
Hurricane Panda	0.9885	0.9885	0.9886	0.9885
Lazarus Group	0.9942	0.9942	0.9943	0.9942
Mirage	0.9942	0.9942	0.9943	0.9942
Patchwork	0.9812	0.9813	0.9813	0.9812
Sandworm	0.9971	0.9971	0.9971	0.9971
Shiqiang	0.9962	0.9962	0.9962	0.9962
Transparent Tribe	0.9889	0.9889	0.9889	0.9889
Violin Panda	0.9952	0.9950	0.9949	0.9952
Volatile Cedar	0.9986	0.9986	0.9986	0.9986
Winnti Group	0.9933	0.9933	0.9933	0.9933
Mean	0.9922	0.9922	0.9922	0.9922

Time Performance

Tables 8.7 and 8.8 show details about duration of phases and stages. Also in this case, Random Forest outperforms the other algorithms completing the task in some order of magnitude lower than the others.

Table 8.7. Execution time of Detection phase in seconds.

	XGBoost	SVM	Random Forest
Training Stage	20.4710 ± 7.7513	716.4045 ± 46.5924	1.2394 ± 0.1590
Prediction Stage	0.0858 ± 0.0076	15.8967 ± 0.9876	0.2086 ± 0.0326

Table 8.8. Execution time of Identification phase in seconds.

APT	Training Stage	Test Stage
APT28	0.4707 ± 0.0585	0.1277 ± 0.0023
APT29	0.4764 ± 0.0707	0.1439 ± 0.0340
APT30	0.5490 ± 0.0525	0.1405 ± 0.0397
Carbanak	0.4700 ± 0.0569	0.1270 ± 0.0054
Desert Falcon	0.5333 ± 0.0973	0.1310 ± 0.0040
Hurricane Panda	0.4985 ± 0.0618	0.1375 ± 0.0159
Lazarus Group	0.4694 ± 0.0331	0.1307 ± 0.0075
Mirage	0.4578 ± 0.0096	0.1299 ± 0.0070
Patchwork	0.4774 ± 0.0581	0.1310 ± 0.0098
Sandworm	0.4834 ± 0.0403	0.1307 ± 0.0058
Shiqiang	0.4903 ± 0.0624	0.1273 ± 0.0032
Transparent Tribe	0.4589 ± 0.0444	0.1282 ± 0.0086
Violin Panda	0.4900 ± 0.0541	0.1332 ± 0.0180
Volatile Cedar	0.4739 ± 0.0754	0.1280 ± 0.0029
Winnti Group	0.4662 ± 0.0448	0.1371 ± 0.0217
Total Time	7.2651 ± 0.0547	1.9837 ± 0.0124

Results Discussion

To summarize the experiments, the Random Forest classification algorithm outperforms the others demonstrating its good qualities in all the metrics, included runtimes. However, the high number of features can be the cause of the poor results of XGBoost and SVM results. In fact, the Random Forest classification algorithm includes a phase in which it applies some feature selection and features randomization methodologies. During the design of the previous approach, we observed that the use of Isolation Forest in combination with Linear Discriminant Analysis leads to good performances. Therefore it can be interesting in the future to evaluate also the application of algorithms for features selection or features projection, like LDA or Principal Component Analysis, to reduce feature dimensions and thus the impact of the *curse of dimensionality*.

Actually, our new methodology shows exceptional results in both Detection and Identification. This enhanced version strongly improves results obtained in the previous approaches, confirming the correctness of the idea of separating the Detection and Identification phases.

Chapter 9

Conclusions

Critical Infrastructures are important assets for countries and organizations. It should not be surprising that nations are investing huge resources into improving the security of CIs. As computer technologies evolve and grow more pervasive, the cyber-space becomes the preferred attack surface for malicious people. Therefore enhance the cyber-security level of CIs becomes urgent, but it is an extremely demanding task.

The first part of this Thesis (Chapter 1 and Chapter 2) has the objective to provide a comprehensive view of the problems, what has been done until now and what is currently lacking. We have explained the importance of defending Critical Infrastructures and how most of the cyber-attacks against them pass through malware. Moreover, we have demonstrated the dangerousness of Advanced Persistent Threats and why they require special treatment. We have also shown the current state of the art of the various research areas that can have an impact when discussing CIs defenses. This part

In the second part of the Thesis (Chapter 3 and Chapter 4), we have presented the basis of our proposal. We believe that one of the needs of CIs is a semi-automatic architecture that secures the organization against malware infection. Moreover, we have detailed the datasets we use in our works to provide means for future comparisons by other researchers.

The last part of the Thesis (Chapters 5 to 8) focuses on our proposed approaches for supporting malware analysts. We have designed modules for our proposed architecture that can give useful information to security experts, significantly reducing their workload. We have tested all these modules by using our datasets to demonstrate their efficacy in malware defenses.

9.1 Ideas for Future Works

We have several ideas regard improvements of the modules. First of all, we would like to enlarge *MalFamGT* with other families, analyzing them with more advanced dynamic tools that can hopefully be better against anti-VM techniques. Also, dAPTasetcan be improved. We want to extend it with additional information related to Victims and Infrastructures with in-depth details that can help security experts. In addition, we would enrich dAPTaset, by improving the report parsing step through

more advanced text analysis. It would be interesting to perform advanced semantic analysis, or rely on artificial intelligence techniques in order to extract data related to behaviors, target, etc., and also improve the attribution to the original attacking group directly from each report.

We are also interested in adding new classes of features taking inspiration from some recent works. For example, Webster *et al.* [101] demonstrate the importance of the *Rich Header* in malware similarity field and APT-related malware field. This header is one of the *hidden attributes* of the PE32 executable. They find that this hidden header is also common in malware, present in 71% of their random sample set. Leveraging this information, they present proof of concept methods to demonstrate the significant value the Rich Header can provide for triage. Thus we think that features extracted from it can be very helpful for our analysis modules.

Another interesting work that can be adopted in our architecture is *APIScout* [78]. It is a fully automated method to recover API usage information from memory dumps. Authors have developed the concept of ApiVectors that efficiently stores the information extracted by ApiScout. They test their methodology on a large malware dataset with interesting results. We think that it is possible to modify their approach to work also without requiring the execution of the malware and its memory dump. We are investigating a static approach based on the same concept of the APIVector withe the goal of integrating it in our architecture, specifically in the APT-Triage module.

Also, Caliskan *et al.* [12] recently present an interesting idea, based on the use of code stylometry analysis against compiled binaries. Code stylometry is a branch of stylometry with the objective of identifying which one of the known authors writes a particular software. As far as we know, none has so far applied these concepts in the malware analysis field, but we are confident that it can be possible. In fact, discovering which APT developed a particular malware is nothing else than identifying which of the known authors is the one behind the malicious software. Therefore we think that these features can be very helpful in the APT Identification of our architecture. Moreover, we believe that stylometry features can be useful in improving malware family identification. In fact, many variants of the same malware are developed from the same piece of code, thus identifying a common author can be proof of a relationship among different suspicious software.

Moreover, as stated in Section 7.4, our modules have a weakness. In fact, adversaries can try to *trick* them through active or passive techniques. For this reason, we want to improve the robustness of our modules, adding some methodologies to avoid that our machine learning algorithms are influenced by attackers. The first step could be the integration of some specialized libraries, like CleverHans [76]. They can provide strong help in improving the robustness through additional defenses and also allowing the possibility of easily testing our architecture against such threats. We need to try to find the defenses that better suit our architecture and its need. However, we think that it would not be possible to make such improvements in a straight way, due to strong requirements that we think such architecture must have when dealing with Critical Infrastructures security.

9.2 Last Considerations

Improvements for the security of Critical Infrastructures have been presented in this Thesis, highlighting problems and limits of current approaches. We believe that these proposals can be helpful both for security experts, reducing the time needed for their tasks, and also for other researchers that can take inspiration from our methodology and compare their works with our results.

Appendix A

Clustering Features List

This appendix contains the full list of features used by family detection and identification modules described in Chapter 5 and Chapter 6. Table A.1 contains the list of static features, table A.2 contains the list of process features, table A.3 contains features related to filesystem, table A.4 contains the list of features related to networking and table A.5 contains features related to activity on the Windows registry.

Table A.1. List of *Static Features*

Name	Type
File size	Integer
Sections with virtual size / 2 >raw size	
Sections with entropy <1	
Sections with entropy >7	
Other sections count	
Resources count	
Imported libraries count	
Imported API count from Kernel*.dll	
Imported API count from User*.dll	
Imported API count from ADVAPI*.dll	
Imported API count from SHELL*.dll	
Imported API count from COMCTL*.dll	
Imported API count from CRYPT*.dll	
Imported API count from msvcr*.dll	
Imported API count from GDI*.dll	
Imported API count from SHLWAPI*.dll	
Imported API count from WS*.dll	
Imported API count from WININET.dll	

Imported API count from WINHTTP.dll	
Concatenated strings Most frequent string	String
Is 32 bit? Has GUI? Has .rdata section? Has .data section? Has .rsrc section? Has .reloc section? Is present âĂŸLoadLibrary*âĂŹ ? Is present âĂŸGetProcAddressâĂŹ ? Is present âĂŸMessageBox*âĂŹ ? Is present âĂŸShellExecute*âĂŹ ? Is present âĂŸIsDebuggerPresentâĂŹ ? Is present âĂŸVirtualAlloc*âĂŹ ? Is present âĂŸCreateThreadâĂŹ ? Is present âĂŸCreateProcessâĂŹ ? Is present âĂŸOpenProcess*âĂŹ ? Is present âĂŸRaiseExceptionâĂŹ ? Is present âĂŸCreateEvent*âĂŹ ? Is present âĂŸGetSystemInfoâĂŹ ? Is present âĂŸGetComputerName*âĂŹ ? Is present âĂŸSetWindowsHook*âĂŹ ? Is present âĂŸWriteProcessMemoryâĂŹ ? Is present âĂŸGetTickCountâĂŹ ? Is present âĂŸSleepâĂŹ ? Is present âĂŸGetDiskFreeSpace*âĂŹ ? Is present âĂŸSetThreadContextâĂŹ ? Is present âĂŸCreateRemoteThreadâĂŹ ? Is present âĂŸGetVersion*âĂŹ ? Is present âĂŸGetProcessHeapâĂŹ ? Is present âĂŸGetUserNameâĂŹ ? Is present âĂŸExitProcessâĂŹ ? Is present âĂŸ_CorExeMainâĂŹ ? Is present âĂŸWaitForSingleObjectâĂŹ ? Is present âĂŸGetStartupInfo*âĂŹ ? Is present âĂŸGetKeyboard*âĂŹ ? Is present âĂŸSetUnhandledExceptionFilterâĂŹ ?	Binary

Is present âĂŸHttpSendRequestâĂŹ ?	
Is present âĂŸHttpQueryInfoâĂŹ ?	

Table A.2. List of *Process Features*

Name	Type
Children processes count	Integer
Total processes count	
Main process threads count	
Main process children threads count	
Other processes threads count	
Other processes children threads count	
Executed commands count	
Started services count	
Created services count	
Number of calls to âĂŸFlsGetValueâĂŹ	
Number of calls to âĂŸCreateWindowâĂŹ	
Number of calls to âĂŸGetSystemMetricsâĂŹ	
Number of calls to âĂŸNtMapViewOfSectionâĂŹ	
Number of calls to âĂŸRtlRunDecodeUnicodeStringâĂŹ	
Number of calls to âĂŸSystemParametersInfoâĂŹ	
Number of calls to âĂŸOpenServiceâĂŹ	
Number of calls to âĂŸRemoveDirectory*âĂŹ	
Number of calls to âĂŸNtOpenDirectory*âĂŹ	
Number of calls to âĂŸGetAdaptersAddressesâĂŹ	
Number of calls to âĂŸCopyFileâĂŹ	
Number of calls to âĂŸNtSetTimerâĂŹ	
Number of calls to âĂŸProcess32NextâĂŹ	
Number of calls to âĂŸGetCursorPosâĂŹ	
Number of calls to âĂŸCryptHashDataâĂŹ	
Number of calls to âĂŸCryptCreateHashâĂŹ	
Number of calls to âĂŸInternetSetOptionâĂŹ	
Number of calls to âĂŸHttpQueryInfoâĂŹ	
Number of calls to âĂŸHttpAddRequestHeadersâĂŹ	

Number of calls to âĂŸInternetReadFileâĂŹ		
Number of calls to âĂŸHttpSendRequestâĂŹ		
Number of calls to âĂŸGetClipboardâĂŹ		
Number of calls to âĂŸUnhandledExceptionFilterâĂŹ		
Number of calls to âĂŸGetLastErrorâĂŹ		
Number of calls to âĂŸTerminateProcessâĂŹ		
Number of calls to âĂŸGetFileVersionInfoâĂŹ		
Number of calls to âĂŸHGetFolderPathâĂŹ		
Number of calls to âĂŸVirtualProtectâĂŹ		
Number of calls to âĂŸNtWaitForSingleObjectâĂŹ		
Number of calls to âĂŸNtAllocateVirtualMemoryâĂŹ		
Number of calls to âĂŸNtCreateThread*âĂŹ		
Number of calls to âĂŸNtQueryInformationProcessâĂŹ		
Number of calls to âĂŸNtResumeThreadâĂŹ		
Number of calls to âĂŸNtTerminateProcessâĂŹ		
Number of calls to âĂŸNtCreateFileâĂŹ		
Number of calls to âĂŸNtOpenFileâĂŹ		
Number of calls to âĂŸNtQueryInformationFileâĂŹ		
Number of calls to âĂŸLdrLoadDllâĂŹ		
Number of calls to âĂŸNtCreateSectionâĂŹ		
Number of calls to âĂŸNtOpenSectionâĂŹ		
Number of calls to âĂŸNtQueryDirectoryFileâĂŹ		
Number of calls to âĂŸNtProtectVirtualMemoryâĂŹ		
Number of calls to âĂŸNtQueryAttributesFileâĂŹ		
Number of calls to âĂŸDeviceIoControlâĂŹ		
Number of calls to âĂŸNtQuerySystemInformationâĂŹ		
Number of calls to âĂŸRegOpenKey*âĂŹ		
Number of calls to âĂŸRegCreateKey*âĂŹ		
Number of calls to âĂŸRegEnumValue*âĂŹ		
Number of calls to âĂŸRegQueryValue*âĂŹ		

Number of calls to âĂŸRegQueryInfoKey*âĂŹ Number of calls to âĂŸNtOpenKeyâĂŹ Number of calls to âĂŸNtQueryValueKeyâĂŹ Number of calls to âĂŸbindâĂŹ Number of calls to âĂŸconnectâĂŹ Number of calls to âĂŸsendâĂŹ Number of calls to âĂŸrecvâĂŹ	
Main process children names Other processes names Other processes children names	String

Table A.3. List of *Filesystem Features*

Name	Type
Read PIPEs count	Integer
Written PIPEs count	
Read files count from âĂŸFontsâĂŹ folder	
Read files count from âĂŸAssemblyâĂŹ folder	
Read files count from âĂŸChromeâĂŹ folder	
Written files count in âĂŸChromeâĂŹ folder	
Read files count from âĂŸPythonâĂŹ folder	
Read files count from âĂŸSystem32âĂŹ folder	
Written files count in âĂŸSystem32âĂŹ folder	
Read files count from âĂŸTempâĂŹ folder	
Written files count in âĂŸTempâĂŹ folder	
Read files count from âĂŸDeviceâĂŹ folder	
Written files count in âĂŸDeviceâĂŹ folder	
Read files count from âĂŸWindowsMailâĂŹ folder	
Written files count in âĂŸWindowsMailâĂŹ folder	
Read files count from âĂŸCommon FilesâĂŹ folder	
Written files count in âĂŸCommon FilesâĂŹ folder	
Read files count from âĂŸProgram FilesâĂŹ folder	
Written files count in âĂŸProgram FilesâĂŹ folder	
Read files count from âĂŸAppDataâĂŹ folder	
Written files count in âĂŸAppDataâĂŹ folder	
Read files count from âĂŸMicrosoft.NETâĂŹ folder	
Written files count in âĂŸMicrosoft.NETâĂŹ folder	
Read files count from âĂŸMicrosoftâĂŹ folder	
Written files count in âĂŸMicrosoftâĂŹ folder	
Read files count from âĂŸProgramDataâĂŹ folder	
Written files count in âĂŸProgramDataâĂŹ folder	
Read files count from âĂŸDocuments and settingsâĂŹ folder	
Written files count in âĂŸDocuments and settingsâĂŹ folder	
Read files count from âĂŸWindowsâĂŹ folder	
Written files count in âĂŸWindowsâĂŹ folder	
Read files count from âĂŸUsersâĂŹ folder	

Written files count in âĂŸUsersâĂŹ folder	
Read files count from other folders	
Written files count in other folders	
Created EXE count	
Created BAT count	
Created COM count	
Created VBS count	
Created JPG count	
Created SYS count	
Created PIF count	
Created MSI count	
Created MSP count	
Created TMP count	
Created DAT count	
Read EXE count	
Read DLL count	
Read BAT count	
Created files count with multiple extensions	
Deleted files count	
Created mutexes count	

Table A.4. List of *Network Features*

Name	Type
Contacted âĂŸcomâĂŹ domains count	Integer
Contacted âĂŸorgâĂŹ domains count	
Contacted âĂŸnetâĂŹ domains count	
Contacted other domains count	
HTTP GET requests count	
HTTP POST requests count	
Other HTTP requests count	
Contacted hosts count	
Hosts subnet	
udp packets count	
udp source IP subnet	
udp destination IP subnet	
irc packets count	
irc source IP subnet	
irc destination IP subnet	
smtp packets count	
tcp packets count	
tcp source IP subnet	
tcp destination IP subnet	
dns packets count	
icmp packets count	
udp source port average	Real
udp destination port average	
tcp source port average	
tcp destination port average	
User-Agent	String

Table A.5. List of *Registry Features*

Name	Type
Deleted registry keys count	Integer
read HKLM\software\Policies\Microsoft\Windows\CurrentVersion\Internet Settings count	
written HKLM\software\Policies\Microsoft\Windows\CurrentVersion\Internet Settings count	
read HKLM\Software\Microsoft\.NETFramework count	
written HKLM\Software\Microsoft\.NETFramework count	
read HKLM\Software\System count	
written HKLM\Software\System count	
read HKLM\Software\Classes count	
written HKLM\Software\Classes count	
read HKLM\Software\Microsoft\Windows\CurrentVersion count	
written HKLM\Software\Microsoft\Windows\CurrentVersion count	
read HKLM\Software\Microsoft\Windows NT\CurrentVersion count	
written HKLM\Software\Microsoft\Windows NT\CurrentVersion count	
read HKLM\Software\Policies count	
written HKLM\Software\Policies count	
read other HKLM count	
written other HKLM count	
read HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer count	
written HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer count	
read HKCU\Software\Policies\Microsoft\Windows\CurrentVersion\Internet Settings count	
written HKCU\Software\Policies\Microsoft\Windows\CurrentVersion\Internet Settings count	
read HKCU\Software\Classes count	
written HKCU\Software\Classes count	
read HKCU\Software\Microsoft\Windows\CurrentVersion count	
written HKCU\Software\Microsoft\Windows\CurrentVersion count	
read HKCU\Software\Microsoft\Windows NT\CurrentVersion count	
written HKCU\Software\Microsoft\Windows NT\CurrentVersion count	
read HKCU\Software\Policies count	
written HKCU\Software\Policies count	
read other HKEY CURRENT USER count	
written other HKEY CURRENT USER count	
read HKEY CLASSES ROOT count	
written HKEY CLASSES ROOT count	
read HKEY USERS count	
written HKEY USERS count	
Run at startup?	Binary

Bibliography

[1] Abou-Assaleh, T., Cercone, N., Keselj, V., and Sweidan, R. N-gram-based detection of new malicious code. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, vol. 2, pp. 41–42. IEEE (2004).

[2] Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M., and Giacinto, G. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the sixth ACM conference on data and application security and privacy*, pp. 183–194. ACM (2016).

[3] Andriesse, D., Slowinska, A., and Bos, H. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 177–189. IEEE (2017).

[4] Bailey, M., Oberheide, J., Andersen, J., Mao, Z. M., Jahanian, F., and Nazario, J. Automated classification and analysis of internet malware. In *International Workshop on Recent Advances in Intrusion Detection*, pp. 178–197. Springer.

[5] Baldoni, R. and Montanari, L. Italian national cyber security framework. In *Proceedings of the International Conference on Security and Management (SAM)*, p. 168 (2016).

[6] Bayer, U., Comparetti, P. M., Hlauschek, C., Kruegel, C., and Kirda, E. Scalable, behavior-based malware clustering. In *NDSS*, vol. 9, pp. 8–11. Citeseer.

[7] Bayer, U., Kruegel, C., and Kirda, E. *TTAnalyze: A tool for analyzing malware*. na (2006).

[8] Borbely, R. S. On normalized compression distance and large malware. *Journal of Computer Virology and Hacking Techniques*, **12** (2016), 235.

[9] Breiman, L. Random forests. *Machine learning*, **45** (2001), 5.

[10] Brewer, R. Advanced persistent threats: minimising the damage. *Network security*, **2014** (2014), 5.

[11] Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., and Yin, H. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pp. 65–88. Springer (2008).

[12] Caliskan, A., Yamaguchi, F., Dauber, E., Harang, R., Rieck, K., Greenstadt, R., and Narayanan, A. When coding style survives compilation: De-anonymizing programmers from executable binaries.

[13] Caltagirone, S., Pendergast, A., and Betz, C. The diamond model of intrusion analysis. (2013).

[14] Chang, C.-C. and Lin, C.-J. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, **2** (2011), 27:1. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[15] Chen, P., Desmet, L., and Huygens, C. A study on advanced persistent threats. In *IFIP International Conference on Communications and Multimedia Security*, pp. 63–72. Springer (2014).

[16] Chen, P., Desmet, L., and Huygens, C. A study on advanced persistent threats. In *IFIP International Conference on Communications and Multimedia Security*, pp. 63–72. Springer (2014).

[17] Chen, T. and Guestrin, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785–794. ACM (2016).

[18] Christopher, D. M., Prabhakar, R., and Hinrich, S. Introduction to information retrieval. *An Introduction To Information Retrieval*, **151** (2008), 5.

[19] Conteh, N. Y. and Schmick, P. J. Cybersecurity: risks, vulnerabilities and countermeasures to prevent social engineering attacks. *International Journal of Advanced Computer Research*, **6** (2016), 31.

[20] Dahl, G. E., Stokes, J. W., Deng, L., and Yu, D. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 3422–3426. IEEE (2013).

[21] Do, C. B. and Batzoglou, S. What is the expectation maximization algorithm? *Nature biotechnology*, **26** (2008), 897.

[22] Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, vol. 96, pp. 226–231 (1996).

[23] Falliere, N., Murchu, L. O., and Chien, E. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, **5** (2011), 29.

[24] Friedberg, I., Skopik, F., Settanni, G., and Fiedler, R. Combating advanced persistent threats: From network event correlation to incident detection. In *Computer & Security, Elsevier*, pp. 35–57. Elsevier (2015).

[25] Ghanaei, V., Iliopoulos, C. S., and Overill, R. E. A statistical approach for discovering critical malicious patterns in malware families. In *The Seventh International Conferences on Pervasive Patterns and Applications,(Patterns 2015): IARIA* (2015).

[26] Grajski, K. A., Breiman, L., Di Prisco, G. V., and Freeman, W. J. Classification of eeg spatial patterns with a tree-structured methodology: Cart. *IEEE transactions on biomedical engineering*, (1986), 1076.

[27] Guha, S., Rastogi, R., and Shim, K. Cure: an efficient clustering algorithm for large databases. In *ACM Sigmod Record*, vol. 27, pp. 73–84. ACM (1998).

[28] Han, J., Pei, J., and Kamber, M. *Data mining: concepts and techniques*. Elsevier.

[29] Harley, D. The game of the name malware naming, shape shifters and sympathetic magic. In *CEET 3rd Intl. Conf. on Cybercrime Forensics Education & Training, San Diego, CA*.

[30] Hu, X., Shin, K. G., Bhatkar, S., and Griffin, K. Mutantx-s: Scalable malware clustering based on static features. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pp. 187–198 (2013).

[31] Hurier, M., Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. On the lack of consensus in anti-virus decisions: metrics and insights on building ground truths of android malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 142–162. Springer.

[32] Hutchins, E. M., Cloppert, M. J., and Amin, R. M. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, (2011).

[33] Indyk, P. and Motwani, R. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pp. 604–613. ACM (1998).

[34] Intelligence, D. S. C. T. U. T. Analysis of dhs nccic indicators. Available from: `https://www.secureworks.com/research/analysis-of-dhs-nccic-indicators`.

[35] Islam, R., Tian, R., Batten, L., and Versteeg, S. Classification of malware based on string and function feature selection. In *2010 Second Cybercrime and Trustworthy Computing Workshop*, pp. 9–17. IEEE (2010).

[36] Islam, R., Tian, R., Batten, L. M., and Versteeg, S. Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications*, **36** (2013), 646.

[37] Jang, J., Brumley, D., and Venkataraman, S. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 309–320. ACM.

[38] Jeun, I., Lee, Y., and Won, D. A practical study on advanced persistent threats. In *Computer applications for security, control and system engineering*, pp. 144–152. Springer (2012).

[39] Kang, M. G., Poosankam, P., and Yin, H. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pp. 46–53. ACM (2007).

[40] Kawaguchi, N. and Omote, K. Malware function classification using apis in initial behavior. In *2015 10th Asia Joint Conference on Information Security*, pp. 138–144. IEEE (2015).

[41] Khodamoradi, P., Fazlali, M., Mardukhi, F., and Nosrati, M. Heuristic metamorphic malware detection based on statistics of assembly instructions using classification algorithms. In *Computer Architecture and Digital Systems (CADS), 2015 18th CSI International Symposium on*, pp. 1–6. IEEE (2015).

[42] Kinable, J. and Kostakis, O. Malware classification based on call graph clustering. *Journal in computer virology*, **7** (2011), 233.

[43] Kong, D. and Yan, G. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1357–1365. ACM (2013).

[44] Kostakis, O., Kinable, J., Mahmoudi, H., and Mustonen, K. Improved call graph comparison using simulated annealing. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pp. 1516–1523. ACM (2011).

[45] Kozak, M. "a dendrite method for cluster analysis" by caliński and harabasz: A classical work that is far too often incorrectly cited. **41**, 2279. doi:10.1080/03610926.2011.560741.

[46] Laurenza, G., Aniello, L., Lazzeretti, R., and Baldoni, R. Malware triage based on static features and public apt reports. In *International Conference on Cyber Security Cryptography and Machine Learning* (2017).

[47] Laurenza, G. and Lazzeretti, R. daptaset: a comprehensive mapping of apt-related data. In *International Workshop on Security for Financial Critical Infrastructures and Services (FINSEC2019)*.

[48] Laurenza, G., Lazzeretti, R., and Mazzotti, L. Malware triage for early identification of advanced persistent threat activities. *arXiv preprint arXiv:1810.07321*, (2018).

[49] Laurenza, G., Ucci, D., Aniello, L., and Baldoni, R. An architecture for semi-automatic collaborative malware analysis for CIs. In *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*, pp. 137–142. IEEE (2016).

[50] Li, P., Hastie, T. J., and Church, K. W. Margin-constrained random projections and very sparse random projections. In *Proceedings of the Conference on Learning Theory (COLT)*, pp. 635–649. Citeseer (2006).

[51] Li, P., Hastie, T. J., and Church, K. W. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 287–296. ACM (2006).

[52] Li, P., Liu, L., Gao, D., and Reiter, M. K. On challenges in evaluating malware clustering. In *International Workshop on Recent Advances in Intrusion Detection*, pp. 238–255. Springer. Available from: `http://link.springer.com/chapter/10.1007/978-3-642-15512-3_13`.

[53] Lilt, D. and Kubala, F. Online speaker clustering. In *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, pp. I–333. IEEE (2004).

[54] Lipovsky, R. Blackenergy trojan strikes again: Attacks ukrainian electric power industry. `http://www.welivesecurity.com/2016/01/04/blackenergy-trojan-strikes-again-attacks-ukrainian-electric-power-industry/` (2016). Accessed: 2016-03-31.

[55] Liu, F. T., Ting, K. M., and Zhou, Z.-H. Isolation forest. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pp. 413–422. IEEE (2008).

[56] Liu, J., Lee, J. P., Li, L., Luo, Z.-Q., and Wong, K. M. Online clustering algorithms for radar emitter classification. *IEEE transactions on pattern analysis and machine intelligence*, **27** (2005), 1185.

[57] Lodi, G., Aniello, L., Di Luna, G. A., and Baldoni, R. An event-based platform for collaborative threats detection and monitoring. *Information Systems*, **39** (2014), 175.

[58] MacQueen, J. et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, pp. 281–297. Oakland, CA, USA. (1967).

[59] Maimon, O. and Rokach, L. *Data mining and knowledge discovery handbook*, vol. 2. Springer (2005).

[60] Marchetti, M., Pierazzi, F., Colajanni, M., and Guido, A. Analysis of high volumes of network traffic for advanced persistent threat detection. *Computer Networks*, (2016).

[61] Marchetti, M., Pierazzi, F., Guido, A., and Colajanni, M. Countering advanced persistent threats through security intelligence and big data analytics. In *2016 8th International Conference on Cyber Conflict (CyCon)*, pp. 243–261. IEEE (2016).

[62] Martin, L. Cyber kill chain® · lockheed martin.

[63] McAfee. Quarterly report - august 2019. Available from: `https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-aug-2019.pdf`.

[64] Mika, S., Ratsch, G., Weston, J., Scholkopf, B., and Mullers, K.-R. Fisher discriminant analysis with kernels. In *Neural networks for signal processing, signal processing society workshop, IEEE*, pp. 41–48. IEEE (1999).

[65] Mohaisen, A. and Alrawi, O. Av-meter: An evaluation of anti-virus scans and labels. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 112–131. Springer.

[66] Mohaisen, A., Alrawi, O., and Mohaisen, M. Amal: High-fidelity, behavior-based automated malware analysis and classification.

[67] Mohaisen, A., Alrawi, O., and Mohaisen, M. Amal: High-fidelity, behavior-based automated malware analysis and classification. *Computers & Security*, (2015).

[68] Nari, S. and Ghorbani, A. A. Automated malware classification based on network behavior. In *2013 International Conference on Computing, Networking and Communications (ICNC)*, pp. 642–647. IEEE (2013).

[69] Nasi, E. Bypass anti-virus dynamic analysis. *Limitations of the AV model and how to*, (2014).

[70] Nataraj, L., Karthikeyan, S., Jacob, G., and Manjunath, B. S. Malware images: visualization and automatic classification. In *Proceedings of international symposium on visualization for cyber security*. ACM.

[71] Nath, H. V. and Mehtre, B. M. Static malware analysis using machine learning methods. In *SNDS* (2014).

[72] NIST. Cybersecurity framework. Available from: `https://www.nist.gov/cyberframework`.

[73] Oliva, A. and Torralba, A. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision*, **42** (2001), 145.

[74] Oppliger, R. and Rytz, R. Does trusted computing remedy computer security problems? *IEEE Security & Privacy*, **3** (2005), 16.

[75] Pankov, N. Wannacry: What you need to know. Kaspersky (2017). Available from: `https://www.kaspersky.co.in/blog/wannacry-what-you-need-to-know/6933/`.

[76] Papernot, N., Goodfellow, I., Sheatsley, R., Feinman, R., and McDaniel, P. cleverhans v2. 0.0: an adversarial machine learning library. *arXiv preprint arXiv:1610.00768*, **10** (2016).

[77] Pitolli, G., Aniello, L., Laurenza, G., Querzoni, L., and Baldoni, R. Malware family identification with birch clustering. In *2017 International Carnahan Conference on Security Technology (ICCST)*, pp. 1–6. IEEE (2017).

[78] Plohmann, D., Enders, S., and Padilla, E. Apiscout: Robust windows api usage recovery for malware characterization and similarity analysis (2018).

[79] Raff, E. and Nicholas, C. An alternative to ncd for large sequences, lempel-ziv jaccard distance. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1007–1015. ACM (2017).

[80] Rieck, K., Trinius, P., Willems, C., and Holz, T. Automatic analysis of malware behavior using machine learning. **19**, 639.

[81] Rousseeuw, P. J. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. **20**, 53. Available from: `http://www.sciencedirect.com/science/article/pii/0377042787901257`, `doi:10.1016/0377-0427(87)90125-7`.

[82] Ryder, B. G. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, (1979), 216.

[83] Santos, I., Devesa, J., Brezo, F., Nieves, J., and Bringas, P. G. Opem: A static-dynamic approach for machine-learning-based malware detection. In *International Joint Conference CISIS12-ICEUTE12-SOCO 12 Special Sessions*, pp. 271–280. Springer (2013).

[84] Sculley, D. Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*. ACM (2010).

[85] Sebastián, M., Rivera, R., Kotzias, P., and Caballero, J. AVclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 230–253. Springer.

[86] Serfling, R. and Wang, S. General foundations for studying masking and swamping robustness of outlier identifiers. *Statistical Methodology*, **20** (2014), 79.

[87] Shick, D. and OMeara, K. Unique approach to threat analysis mapping: A malware centric methodology for better understanding the adversary landscape. Tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA PITTSBURGH United States (2016).

[88] Steinbach, M., Karypis, G., Kumar, V., et al. A comparison of document clustering techniques. In *KDD workshop on text mining*, vol. 400, pp. 525–526. Boston (2000).

[89] Su, Y., Lib, M., Tang, C., and Shen, R. A framework of apt detection based on dynamic analysis. (2016).

[90] Tan, P.-N. *Introduction to data mining*. Pearson Education India (2018).

[91] Tian, R., Batten, L., Islam, R., and Versteeg, S. An automated classification system based on the strings of trojan and virus families. In *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 23–30. IEEE (2009).

[92] Tian, R., Batten, L. M., and Versteeg, S. Function length as a tool for malware classification. In *2008 3rd International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 69–76. IEEE (2008).

[93] Tian, R., Islam, R., Batten, L., and Versteeg, S. Differentiating malware from cleanware using behavioural analysis. In *2010 5th international conference on malicious and unwanted software*, pp. 23–30. IEEE (2010).

[94] Torralba, A., Murphy, K. P., Freeman, W. T., and Rubin, M. A. Context-based vision system for place and object recognition. (2003).

[95] Ucci, D., Aniello, L., and Baldoni, R. Survey of machine learning techniques for malware analysis. *Computers & Security*, (2018).

[96] Ugarte-Pedrero, X., Graziano, M., and Balzarotti, D. A close look at a daily dataset of malware samples. *ACM Transactions on Privacy and Security (TOPS)*, **22** (2019), 6.

[97] Upchurch, J. and Zhou, X. Variant: a malware similarity testing framework. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 31–39. IEEE.

[98] Ussath, M., Cheng, F., and Meinel, C. Concept for a security investigation framework. In *New Technologies, Mobility and Security (NTMS), 2015 7th International Conference on*, pp. 1–5. IEEE (2015).

[99] Virvilis, N. and Gritzalis, D. The big four-what we did wrong in advanced persistent threat detection? In *2013 International Conference on Availability, Reliability and Security*, pp. 248–254. IEEE (2013).

[100] Virvilis, N., Gritzalis, D., and Apostolopoulos, T. Trusted computing vs. advanced persistent threats: Can a defender win this game? In *Ubiquitous Intelligence and Computing, 2013 IEEE 10th International Conference on and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC)*, pp. 396–403. IEEE (2013).

[101] Webster, G. D., Kolosnjaji, B., von Pentz, C., Kirsch, J., Hanif, Z. D., Zarras, A., and Eckert, C. Finding the needle: A study of the pe32 rich header and respective malware triage. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 119–138. Springer (2017).

[102] Wicherski, G. pehash: A novel approach to fast malware clustering. *LEET*, **9** (2009), 8.

[103] Ye, Y., Li, T., Adjeroh, D., and Iyengar, S. S. A survey on malware detection using data mining techniques. *ACM Comput. Surv.*, **50** (2017), 41:1.

[104] Yin, H., Liang, Z., and Song, D. Hookfinder: Identifying and understanding malware hooking behaviors. In *NDSS*, pp. 16–23 (2008).

[105] Yin, H. and Song, D. *Automatic Malware Analysis: An Emulator Based Approach*. Springer Science & Business Media (2012).

[106] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 116–127. ACM (2007).

[107] Zhang, T., Ramakrishnan, R., and Livny, M. BIRCH: an efficient data clustering method for very large databases. In *ACM Sigmod Record*, vol. 25, pp. 103–114. ACM.

[108] Zhao, W. and White, G. A collaborative information sharing framework for community cyber security. In *Homeland Security (HST), 2012 IEEE Conference on Technologies for*, pp. 457–462. IEEE (2012).