Eindhoven University of Technology

MASTER

# Passive asset discovery and operating system fingerprinting in industrial control system networks

Mavrakis, C.

*Award date:*
2015

Link to publication

**Disclaimer**
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

**General rights**
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

# Passive Asset Discovery and Operating System Fingerprinting in Industrial Control System Networks

*Master Thesis*

Chris Mavrakis

Supervisors:
prof.dr. S. Etalle
dr. T. Özçelebi
dr. E. Costante

Eindhoven, October 2015

# Abstract

Maintaining situational awareness in networks of industrial control systems is challenging due to the sheer number of devices involved, complex connections between subnetworks and the delicate nature of industrial processes. While current solutions for automatic discovery of devices and their operating system are lacking, plant operators need to have accurate information about the systems to be able to manage them effectively and detect, prevent and mitigate security and safety incidents. In this work, we present two tools that enable automatic and completely passive discovery of hosts and their operating systems (OSes) and we prove that using machine learning to perform passive OS detection yields promising performance gains. The results of the tests we performed on real-world ICS network data bolster the effectiveness of our approach and propose the direction that future work can follow towards further improvement.

# Acknowledgements (ACK)

This thesis describes the result of my final project towards the completion of the Information Security Technology (IST) master's program at Eindhoven University of Technology.

I would like to thank my supervisor, prof. Sandro Etalle, whose lectures were so interesting that immediately intrigued and motivated me to look into critical infrastructure from the first semester of the aforementioned program. I thank him for honoring me by accepting me as a master's student and bringing me in contact with interesting problems and brilliant people. I also want to thank dr. Tanir Özçelebi for joining the assessment committee and devoting his time to judge this work.

My graduation tutor, dr. Elisa Costante, deserves her own paragraph, along with my sincerest gratitude for all the guidance, feedback and help she provided in the course of this project. Her critical thinking, attention to detail, academic experience and tireless inner drive enabled me to complete this thesis and make the most out of it.

This project was carried out in collaboration with SecurityMatters B.V. which evolved into a welcoming big famiglia. I want to thank Damiano Bolzoni and Emmanuele Zambon who believed in me at first sight and gave me all the guidance I needed to explore the -until then unknown to me- ICS setting. Special thanks to Chris and Maarten for patiently answering all my questions about ICS networks and providing me with all the server power I needed to run experiments. I thank the rest of the crew, too, who welcomed me and made the days at the company so enjoyable: Christina, Daniel, Daniele, Gijs, Giorgio, Marieke, Massimo and Michele thank you so much!

Finally, I thank my family, who tirelessly supported me throughout the project, remotely or by visiting. Really, I don't know many other parents who are so devoted to their children. Last but not least, my love, Despina, thank you for the endless support and encouragement that you never seemed to run out of, no matter how bleak the situation seemed. I truly appreciate it.

Chris M., Eindhoven, October 2015.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

As industrial control system (ICS) networks consist of numerous devices, interconnected in complex ways, managing and securing them effectively is becoming challenging. Increasingly, internet connectivity is added into the mix to enable remote observation and control, but at the same time the attack surface is increased substantially. Consider the network of an electric power company that consists of over 300 devices, organized in 5 subnetworks, spread out in 5 different locations. The administrators and operators of the plant need to know what devices normally live in their network and in what ways they communicate with each other, to be able to quickly identify and tackle safety or security issues, but documentation about the networks is scarce.

One way to gain the desired situational awareness is to manually (using software tools) examine the network, and record all findings in inventory lists, but this approach is flawed in three ways. First, given the device count and complex subnetwork interconnections, it takes a tremendous amount of effort for an administrator to meticulously enumerate all items and find out e.g. what operating system they are running. Second, actively probing devices to discover used IP[1] addresses, open ports and operating systems can be detrimental to ICS components, which are often legacy devices, fine-tuned to handle exactly the connections needed to perform their duties and can even freeze or reset if pinged[2]. Third, the exact moment that the enumeration is completed, the information can be considered time-stamped and stale, since the real situation of the network may soon change. This change, which may be a rogue device just installed on the network, will not be registered in the operator's inventory until the next (costly) enumeration happens, which may be months or years away.

At the same time, monitoring of the network is important because relying on the process of timely vulnerability patching to ensure security cannot be an effective strategy. The lifetime of devices is much longer than that in typical information technology (IT) networks. ICS devices are designed to operate for 15 or more years before needing replacement and taking parts of the system offline to apply updates is done in pre-scheduled maintenance windows that may be months apart. At the same time, the priority of the three components of the information security triad, confidentiality, integrity and availability (CIA) is inverted in industrial environments. Confidentiality comes third, while availability is usually the most important attribute that must be preserved. This hierarchy makes sense, e.g. if one thinks about a power distribution plant or a nuclear reactor's cooling circuit, but it also makes patch management more involved.

**The problem.** This situation creates the need for tools that can detect attributes of systems, along with their positioning and organization in networks in a way that does not obstruct normal operation. The method must be reasonably effective and fast, so that it can be run frequently or

[1] IP refers to the widely used Internet Protocol https://en.wikipedia.org/wiki/Internet_Protocol.

[2] The act of *pinging* refers to the practice of sending an internet control message protocol (ICMP) *echo request* packet to a remote network host, with the expectation that if it is alive and connected it will issue an ICMP *echo reply* towards the requesting host.

continuously, to maintain an updated image of the network and even provide feedback to intrusion detection systems (IDS) that may be in place, aiding detection or interpretation of security events.

**Contributions.** Current approaches have come a long way towards fulfilling the previously-mentioned requirements, but as we will discover in Chapter 2 there is no security silver bullet yet. While we do not claim to provide one ourselves, we design our own approach which revolves around employing classic machine learning techniques to improve passive OS fingerprinting performance and is discussed in Chapter 3. In the same chapter, we set out to measure the performance of the two prominent tools in passive OS fingerprinting, PRADS and p0f v3, which has not been done before in a formal manner. Along with the two well-known tools, we evaluate an implementation of our approach, by testing it on the same real-world datasets and comparing the results head to head.

In regard to the arrangement and connections of networked devices, we refer to the concept as network topology, and we examine methods to discover it passively in Chapter 4. First we measure how much various routing protocols are used in several ICS networks, and demonstrate which are the best candidates to provide topology data. Next, we propose an algorithm that performs topology discovery functions and demonstrate its effectiveness. The algorithm can be run in a continuous way, only outputting data that are considered final, unlikely to change, so that the operator is confident in the readings. We close the chapter with useful take-aways that include conclusions about the minimum time for which a network must be observed before a reliable topology map can be drawn, and what pieces of information should be considered final, after how much time.

In Chapter 5 we wrap the work up with a summary of the findings of individual sections, and their implications in ICS network security. Furthermore, we provide suggestions of exciting new directions that future work can take, to achieve further performance improvements.

**Conclusion teaser.** The results in both fronts (OS fingerprinting and topology discovery) are encouraging. Especially in the former, we show that our approach builds upon the state of the art and improves the desired attributes, while reducing the effects of their limitations. Regarding discovering topologies, our tests bring forth interesting remarks on the time that is required to passively complete a network map of reasonable detail.

# Chapter 2

# Related work

Identification of devices that comprise a computer network is both an interesting and technically challenging task. In networks that include hundreds or more devices, organized in tens of sub-networks, maintaining situational awareness by manual inspection becomes infeasible. As such, the operators of these networks can find great benefit from using tools that automatically complete most of the recognition task, and generate accurate and up-to-date information about the network. We can distinguish two main activities that are needed for such information acquisition:

- **Device Fingerprinting.** In this activity, device-specific information, independent from the network where the device is placed, is acquired. Device-specific attributes gathered in this phase may include the *hostname*, the *operating system (OS)*, the *firmware* version (in the case of embedded devices), the *services* that are running on the device, *protocols* and *ports* in use, *MAC address* and *vendor* of the network interface.

- **Topology Discovery.** The term *network topology* refers to the configuration of a network in terms of how nodes (i.e. hosts or routers) are inter-connected and how a network is divided in segments (i.e. subnets). To gather information about network topology, one should collect data such as *IP address* of each device in the network, their *geographical location*, the set of *subnet* or *local area network (LAN)* and the mapping between a device and the subnet in which it resides. During topology discovery, subnets should be fully defined, i.e. either the complete range of IPs included in a subnet or the first IP address and the subnet mask should be known.

Device fingerprinting and topology discovery caught the interest of the research community, as shown by the presence of numerous scientific papers (from 1999 [31] until recently [2]) and application tools (such as PRADS and the infamous p0f). However, the state-of-the-art mainly proposes solutions for standard IT networks, while few efforts have been done with a specific focus on ICS networks. In this latter case, standard device fingerprinting and topology discovery techniques cannot be applied due to environment-specific constraints: for instance, techniques based on traffic injection (e.g. port scanning) cannot be adopted because of the limited resources of the nodes in the network.

In this chapter, we will explore relevant studies and tools and examine their applicability to ICS networks. The chapter is separated in two sections: Section 2.1, that discusses existing solutions for the device fingerprinting, and Section 2.2 that deals with topology discovery. Most of the methods and tools described throughout the chapter will be assumed to run on a measurement device, referred to as a *sensor* from here on, that is comprised of a computer system that is connected to a computer network and is able to communicate with the rest of the devices.

## 2.1 Device Fingerprinting

In this section, we will examine existing techniques and tools that enable the recognition of applications, operating system or firmware that are running on network connected devices. Due to the focus of the current work, emphasis is given to OS detection and passive acquisition of information. However, approaches for application detection or active acquisition are also briefly discussed.

### 2.1.1 Operating System Detection

Operating system detection can be performed by leveraging various sources of data to predict what OS or firmware version is running on a device. The traditional approach is to use fields of transmission control protocol (TCP) packet headers and compare them to a database of known OSes, a process known as *OS fingerprinting*. Recently, techniques that are based on other protocols have also been developed and they will be discussed in the following.

A main distinctive characteristic between existing OS fingerprinting techniques is related to the way data is gathered, so that we can have *active* or *passive* fingerprinting techniques. Active methods require a two-way interaction between the sensor device (or software) and the network that is being observed. These interactions typically include queries to services running on hosts and subsequent analysis of the reply or lack thereof. Passive methods, on the other hand, solely rely on the observation of communications of the target network or host, without ever needing to transmit any packets on the wire. The network sensor can sometimes be connected to an ethernet network using a cable or plug that has the transmitting wires disconnected, as long as it receives packets that are destined for the hosts or networks it needs to identify. Passive approaches should operate normally even when a network trace file is provided to them, instead of live traffic. Given that no interaction is required, such tools should be able to successfully complete their task just by observing traffic.

#### Active OS Detection

In [41], Veysset et al. analyze the pattern of packet retransmissions caused by network congestion and delays to deduce the OS of a host. Although congestion is a prerequisite, it does not need to be caused intentionally, but can be simulated instead. This is achieved by manipulating the 3-way handshake that is done when a TCP connection is first opened. The second part of the handshake, the SYN+ACK request, must be responded to with an acknowledgement (ACK), and by blocking that response, congestion is simulated. This interaction, though, renders the technique effectively active. Combining multiple packets and analyzing them together has also been proposed, mainly for stimulus-response pairs such as TCP SYN and SYN+ACK. Using this method, signatures can be extracted more efficiently for protocols like ARP, IP, ICMP, UDP and TCP [27]. DNS queries can be captured and analyzed to determine that a mobile device is running the Android OS [24], but this conclusion is probably of limited practical value.

In [20] Kollmann argues that most networked systems in IT networks run services that transmit a plethora of information on the network, which can be used to passively fingerprint them. The claims for the effectiveness of the method are encouraging, as the level of detail and accuracy of the information achieved is unmatched, and successful implementation would provide impressive network maps. In this document, emphasis is given in the DHCP protocol, the implementation of which heavily depends on the version of the OS that a host is running. Using signatures for some of the DHCP fields, such as Option 55, can lead to easy and accurate OS identification on IT networks. Unfortunately, DHCP is not commonly used in ICS networks, so this innovative and highly effective technique cannot be leveraged in these environments.

In an earlier report [19], Kollmann enumerates most of the usable fingerprinting methods available in 2005. Among many valid observations, the ones more relevant to our purposes include the following:

- *Time to live* (TTL) can be a very useful metric. Its initial value is predictable with high accuracy, while at the same time different values are used by different OSes.

- *Server message block* (SMB) and *Microsoft Windows browser protocol* frequently include the exact version of used OS and service pack level.
- Combining multiple techniques to have multiple sources of information available will probably give the best results.

The author of the above reports has incorporated his findings and techniques in a closed-source tool called Satori[1]. Further documentation about how the program operates is scarce and the main supported environment is Microsoft Windows systems. A linux version exists but only implements a small subset of the features.

One of the most established methods for OS identification is TCP fingerprinting [23] which is based on the observation that different TCP implementations construct packets in different ways. As developers add features, fix bugs or improve performance in subsequent releases of implementations, differences can be observed between different versions of an OS, or even a TCP implementation. By exploiting these differences in the packets, one can determine which implementation is used, and consequently estimate which OS is running on the host. Even when the implementation's code is not known, particularities like the rate of connection re-attempts can be observed and recorded in the form of fingerprints, which can later be used for identification [31].

This approach is widely used in tools such as NMAP[23] which initiate TCP connections with hosts, frequently using uncommon combinations of flags and options, and analyze the responses to draw assumptions on the OS of the host that replied. The principle can also be used in a passive manner, where no special connections are initiated but packets of normal TCP traffic are analyzed [37]. Using normal packets instead of crafted ones has negative implications in the performance of the approach, and the options to be used for fingerprinting need to be chosen using different criteria [22] [37] [38].

SinFP is a tool that contains both active and passive OS fingerprinting based on three packets [4]. The packets can be collected easily when in active mode, but doing the same passively can be problematic. The matching engine is different from other solutions, as it uses the intersection of the domains generated by elements of the three packets being checked against the signature database in file. Xprobe2++ is yet another active fingerprinting tool that uses various layers [42]. The signature matching method used is unique, as each element of the signature (e.g. the TTL) is considered independently and a match is seeked [3]. A score is calculated for each possible OS and each possible element, and in the end all the scores for each OS are added together. The OS with the highest score is considered to be the dominant prediction.

#### Passive OS Detection

Research has also been done on approaches that do not require the transmission of any packets on the network, enabling a network sensor to determine the OS of other hosts in a passive fashion. This knowledge may help intrusion detection systems (IDS) solve ambiguities when possible malicious activity is detected [10] [38].

This technique is used by tools that are well known outside academia. Ettercap[2], a tool for man-in-the-middle attacks, also includes active as well as a primitive passive fingerprinter with its own database [30]. P0f[3], on the other hand, can be considered one of the most advanced passive OS fingerprinting tools. Since its release in 2000 there have been three major versions, and the latest one (p0fv3) is the most sophisticated. The tools emphasizes on TCP fingerprinting, although v3 also employs Layer 7 recognition. In order to increase p0f's performance, Barnes et al. have designed k-p0f. The main difference with the original tool is that high layer processing, such as HTTP parsing, is dropped and a kernel module is used to increase efficiency and enable the system to process more packets per time interval [6]. A different approach, to improve the accuracy of p0f, correlates FIN+ACK to the corresponding SYN packets, and analyzes them with respect to each other. While this may yield better results in networks where connections are short-lived, it

[1] http://chatteronthewire.org
[2] https://github.com/Ettercap/ettercap/blob/master/README
[3] http://lcamtuf.coredump.cx/p0f3

may be impractical when the ending packet of a connection is transmitted days or months after the connection is first opened [2].

Due to p0f's supremacy, other tools or toolkits have reused its engine or signatures instead of starting an implementation from scratch. One such approach is PRADS (Passive real-time asset detection system[4]) which combines p0fv2, earlier system PADS (Passive asset detection system) and new techniques, to passively obtain information about network hosts. More details about how PRADS and p0f work can be found in Section 3.1.

Most of the above approaches require fingerprints of known systems to be known beforehand, and generating them manually is a tedious and inefficient task. Especially in areas like ICS networks, where not enough work has been done, automatic fingerprint extraction would be useful. By passively analyzing Layer 7 features such as the grammar of messages, and given labeled data set, the algorithm in [1] can create signatures for the firmware version of embedded devices. These signatures can then be used for identification of malicious devices on a network. This method may be labeled as fingerprinting but in essence it is more closely related to what is refereed as white-listing, while the criteria used for filtering are dynamically generated.

As far as we know, there have been very few attempts to use machine learning to improve passive OS fingerprinting. In [7], Beverly has used two techniques to build a naive Bayesian classifier, which can later be used for matching OS fingerprints. The comparison is done between using p0f's signature database and captured HTTP UserAgent data to train the classifier, and the latter proves to be more effective. The classifier's matching accuracy is not evaluated, due to lack of a suitable data set. An earlier work by Lippmann et al. [22] has studied the way common passive OS fingerprinting software worked and how machine learning could be employed to improve matching. The results suggest that *binary tree classifiers* are the most effective, a notion that we will re-use in Chapter 3.

**Active versus Passive Fingerprinting.** In the following, we provide a list of the main difference between active and passive fingerprinting

- Active methods usually result in faster and more accurate recognition of hosts and networks, because specially crafted messages can be used to trigger responses that contain a lot of data. Also, the rate at which packets are exchanged, and thus the OS detection speed can be increased by the sensor, while within the target device's and network's abilities.
- In contrast, passive approaches are slower to produce results and their conclusions may be less accurate or detailed because they have to work with whatever data is provided to them, at whatever rate it is "naturally" present in the network.
- When stealthiness is required, passive information gathering is the go-to solution. If partial stealthiness is acceptable, an active approach with reduced transmitting rates can be used.
- When operating in sensitive environments, where integrity in general and uptime in particular are required, actively probing devices may prove too dangerous. Either fully passive or fine-tuned active tools must be used to ensure safe operation.
- Passive methods can be run off-line on recorded network data, whereas most of their active counterparts will fail.

### 2.1.2 Service/Application Detection

Instead of the operating system, some techniques have been proposed for the identification of the services and applications used by hosts on a network. For instance, observing that port 22 is accepting connections on a host indicates that an ssh service may be running, which in turn means that ssh vulnerabilities might available for exploitation.

[4] https://gamelinux.github.io/prads

Statistical analysis of IP packet size, inter-arrival time and order can be used to classify application-specific traffic [9]. Using IP flow fingerprinting, the technique can determine what protocol is running in layer 7. This method's main advantage is better performance than the traditional approach of deep packet inspection (DPI), which involves meticulously parsing higher layers of packets and requires the use of a significant amount of system resources.

In [28] it is denoted that using default port numbers to identify which application is running on a host is considered unreliable. This is because random ports may be caused by applications using custom port configurations or traffic encapsulating techniques, such as VLAN functionality over HTTP. Instead, it is proposed that information from higher layers should be used. The approach is further developed in [15], in which packet contents from layers above the transport layer (L4), is used to create signatures, which are then fed to a classifier to determine which application is running. An interesting property is that the approach seems resilient to encryption [28].

Techniques that exploit alternative sources of information have occasionally been proposed, aiming to recognize specific hardware at the remote end of a connection. Typical examples include the analysis of the particularities observed in analog signal of ethernet devices [11] or microscopic deviations in network device hardware known as clock skews [18]. The applicability of these methods is limited because either direct access to the hardware is required or extensive training of the algorithm must be done on specific devices before its deployment.

### 2.1.3 Towards Fingerprinting in ICS

While most of the approaches described above are designed for wide use in IT networks and over the Internet, there exist some approaches that are specialized for ICS networks. In [21], Leverett uses banner grabbing to gather information on publicly facing ICSes, searches for relevant exploits in an online archive of exploits and shellcode, known as ExploitDB, and presents the information along with location data to the user. Furthermore, a complete system called Sophia has been deployed in two power utilities and the results documented. Sophia monitors the ports used by hosts and the relevant network flows, and allows the user to whitelist legitimate instances. Although the approach is dubbed passive fingerprinting, it must be noted that the term *fingerprinting* concerns the whole system and not specific hosts [35]. A slightly different approach, called *flow fingerprinting*, is described in a deliverable of the Crisalis project[8]. Instead of identifying specific software, the focus is put on the role of each device inside the ICS network. The system works by fingerprinting a known network and is then able to recognize devices in similar setups.

All the examined techniques have contributed to the advancement of OS detection, but each one is aiming at solving a slightly different problem or in a different environment. An approach that is completely passive and still offers reasonable performance when used in ICS networks could not be located. For these reasons, in Chapter 3 we propose a solution that focuses on solving the problem of OS detection for ICS networks. Note that, some of the techniques we mentioned above (namely PRADS and p0fv3) will be used as benchmarks to discuss and evaluate the performance of our solution.

## 2.2 Topology Discovery

The term *topology discovery* refers to the process of identifying a network structure, its connections with other networks and the details of its subnets. Information about hosts is also included, such as their physical location, their IP address and the subnet and LAN they belong to. In this section, we survey the existing works that aim at solving the topology discovery problem. The methods can be characterized as active and passive in the same way that OS detection methods have been classified in Section 2.1.1.

When terms such as L2 or L3 are used, they refer to the respective layer of the open systems interconnection model (OSI Model) as defined in [33]. OSI includes layers from physical (L1) up to application (L7). The layers most relevant to our purposes are L2 and L3, the data link and network layer respectively. MAC addresses and network switches reside in L2, while IP addresses

and routers in L3. Subnetworks or subnets are L3 divisions of IP networks, in which hosts have IP addresses with the most significant part, called the network prefix, being the same. The classless inter-domain routing (CIDR) notation is used to describe the IP addresses range of a subnet. With this notation, the first IP address of a subnet is followed by the symbol '/' and the length of the prefix in bits. For example, a subnet can be described with the notation 192.168.0.0/24, meaning that the subnet can contain 256 IP addresses that can range from 192.168.0.0 to 192.168.0.255. Note that only 254 of the possible IP addresses can be used for hosts, as the first and last are reserved for special purposes (i.e. .0 is the network address and .255 is the broadcast address). Instead of the CIDR notation, the same subnet can be defined by the first IP address, 192.168.0.0, and the subnet mask 255.255.255.0.

Network topology discovery has been discussed in numerous research works. Some focus on the discovery of routers and the relationships between them, while others emphasize on discovering as many of the network hosts as possible, regardless of whether they are routers, personal devices or printers. Active and passive techniques seem to be used interchangeably, and in most papers the authors assume that the networks under investigation are connected to the Internet. As far as we know, none of the existing works has a precise focus on a passive approach for topology discovery and no solutions exist that are specifically crafted for ICS networks.

*Alias resolution* is the act of determining which IP addresses in a given subnet (that is remote in relation to the observation point) map to one router that is local. For example, consider two adjacent subnets with IPs such as 192.168.0.0/24 and 192.168.1.0/24, being connected together through a router. The router will have at least two network interfaces; the first of which can have the IP address 192.168.0.1 and the second 192.168.1.1. If both subnets have network sensors installed, and these sensors aggregate the observed data, there will be packets from both 192.168.0.1 and 192.168.1.1 in the dataset, as if the two addresses belong to different devices. Successful alias resolution can correlate the two IPs into a single device with two network interfaces on two different subnets, which is the real situation. After examining alias resolution methods it is evident that many rely on the use of traceroute-like techniques [12], [14]. *Traceroute* is a tool that aims to detect the intermediate hosts between the computer it is running on and a remote system. To obtain this information, traceroute sends packets towards the destination with various time to live (TTL) values in the IP layer. The TTL counter of each packet is decreased every time the packet is being forwarded by a router, and when it reaches zero the packet is dropped. By using different (decreasing) TTL values and observing which packets are getting dropped, traceroute can list each intermediate host up until the destination. Projects that use similar methods, such as traceNET [39] and exploreNET [40], have been shown to be quite effective in network mapping. Although proven, these techniques may not always be applicable in networks in which active probing is prohibited. In these cases, passive collection of data will be used, but as traceroute data may not be present, reliable operation can not be guaranteed.

Alternative methods eliminate the need for traceroute-like probing by making use of the internet group management protocol (IGMP) [26]. It has been suggested that some IGMP messages can be used to infer L2 topology, but they are only transmitted as answers to specific queries. These transactions are not present under normal conditions in networks, but have to be generated, rendering the technique effectively active [25]. Additionally, further limitations apply, such as the fact that only networks with two or more routers can be detected. The symmetry rule though, as formulated in [25] should be noted for further re-use: "All routers attached to the potential L2 network should have the same view [of the network]". This rule can be used to double check that the results of an algorithm that detects a topology using some other method are correct.

*Detecting subnets* is another important component of topology discovery. Having a subnet fully defined is an important step towards being able to tell whether a host is part of it or not. The straight forward way to define a subnet is to know one or more IPs that belong to it, along with the subnet mask. This information is not always readily available, making subnet definition hard. In those cases, methods based on heuristics, like the one outlined in [13], can be used. Network prefixes of all the possible lengths are first calculated for each IP, and then various conditions and rules are used to narrow the candidates down. The technique focuses on internet-connected systems but one of the rules used may be more generally applicable and is formulated as follows.

"Given a candidate /x subnet, the IP addresses from within this subnet should be at similar distances to a vantage point". The "/x" notation refers to CIDR format of subnetting and the mentioned distance is measured in network hops.

Simple network management protocol (SNMP) is portrayed as the dominant source for active gathering of topology data, and is used by the majority of tools that have achieved a high level of suitability for practical operation. In SNMP-enabled hosts, information is saved in management information bases (MIBs) which have flexible but pre-defined formatting. A tool that is aware of the MIBs used can parse data that is relevant and produce topology information in an effective manner [43]. The useful data found in MIBs can be a list of the network interfaces of the host, IP addresses, subnet masks, whether the host is forwarding packets, default TTL, IP routing tables and other information that can greatly aid a topology discovery effort.

While standard MIBs exist, such as MIB-II [34], which are defined globally and are openly available, vendors may opt to use custom ones that are not always published. This feature of SNMP introduces additional complexity during the design of tools that depend on the protocol, while the networks in which the tools will successfully operate may be limited. Some approaches that use SNMP, have built-in MIB compilers which allow them to "learn" new MIBs [5]. This may be useful when vendor-specific MIBs are used and may increase the number of networks that the tool can successfully operate in.

Additional concerns have been expressed about SNMP-based approaches, as they require partial knowledge of the topology of the network already [17]. Also, authorization and credentials for SNMP management is supposedly not always available due to network administrator security practices. Alternatives to SNMP must thus be seeked, especially if only completely passive approaches are required.

Additional systems for creating topology maps that have been designed and implemented include Microsoft's Link layer topology discovery (LLTD)[5] and network introspection by collaborating endpoints (NICE) [17]. LLTD is a link layer protocol that is natively supported by modern Windows OS versions (later than Vista) using software agents, while it can be retrofitted to earlier Windows variants. Using LLTD information, a host can create an accurate map of the L2 local network, which reaches the boundaries that are formed by routers and other L3 devices. NICE also requires agents installed on devices to locate and map them at the L2 level. Despite the inherent activeness of both approaches, NICE also has limited passive detection capabilities, to cope with non-compliant devices such as printers. This is achieved by monitoring ARP messages, but the method cannot be used in networks where no NICE clients are present at all, completely passive operation is not functional.

A 2003 study on network discovery and mapping [5] found that passive tools were generally underdeveloped. Only one passive tool was rendered relevant, which could only provide limited information such as IP, OS version, limited port detection and limited application status and whether the device is active at a given time. During the last decade there have been some advancements in the area of passive topology discovery, but the available tools are still lacking.

Ettercap[6] provides basic topology discovery features such as device type and network distance recognition. It identifies routers by keeping note of the MAC addresses in packets that come from IPs in different subnets, or observing which hosts transmit ICMP TTL-exceeded or redirect messages [36]. It is important to note, though, that the subnet mask must be known, which is a non-trivial task. Arpwatch[7] similarly associates MAC and IP addresses and monitors MAC pairs for connections and traffic volume, trying to detect routers.

Re-using the limited ideas that have been implemented, and taking passive detection further, an article by Hee So at SANS [36] proposes that protocols that are integral to the operation of the network can be used as sources for topology information. Examples include the spanning tree protocol (STP) and multiprotocol label switching (MPLS), used by switching equipment, routing information protocol (RIP), open shortest path first (OSPF), interior gateway routing protocol (IGRP) and intermediate system to intermediate system (IS-IS), used for interior routing.

[5] https://msdn.microsoft.com/en-us/windows/hardware/gg463061.aspx
[6] https://ettercap.github.io/ettercap/index.html
[7] http://ee.lbl.gov/

Information that can be extracted from observed packets may include IP addresses, subnet masks, types of the devices (e.g. router or managed switch) which can be used to feed an algorithm that will create network topology map. Finally, [36] also mentions that multicast join group messages may be used to extract useful relevant information.

Given the lack of a completely passive topology discovery tool that works reasonably well in ICS environments, we propose a topology discovery mechanism that works in a passive manner. The design, implementation and experiments of the algorithm are thoroughly explained in Chapter 4.

A topic that is slightly outside our scope, but is worth mentioning is *autonomous system (AS) level discovery*. Paraphrasing a definition from RFC 1930 [16], to ease understanding, we can say that an AS is a set of routers and other hosts that use an interior gateway protocol and common metrics to route packets within the AS, and an exterior gateway protocol to route packets to other ASes. Numerous methods can be used to achieve AS discovery. Internet registries such as the internet routing registry (IRR)[8] and regional internet registries (RIRs)[9] contain information about ASes as well as how routing between them is done. Also, border gateway protocol (BGP) is a good source of topology information [36], which can be obtained either by directly querying BGP-enabled devices or by accessing BGP dumps which are gathered by projects such as Route Views[10] asynchronously. Although data can be acquired efficiently in these manners, these techniques only apply to inter-AS topologies, which are outside the scope of this work.

[8] ftp://ftp.radb.net/radb/dbase
[9] http://www.isoc.org/briefings/021
[10] http://www.routeviews.org/

# Chapter 3

# Operating System Fingerprinting

As discussed in Section 2.1, *PRADS* and *P0f* represent the state-of-the-art solutions for passive operating system (OS) fingerprinting. In order to identify the gap that still exists towards desired performance and to provide a solution that extends the state of the art by improving the accuracy and detection rate of passive OS recognition, in this Chapter we will first proceed with an evaluation and comparison of PRADS and P0f (Section 3.1). In particular, we will execute the two tools over the same dataset and discuss their capabilities and limitations. As results of this analysis, we will highlight the improvements we intend to make and in Sections 3.2 and 3.3 we will present our solution, that merges the best characteristics of the two tools and tries to address their limitations. Finally, in Section 3.4 we will validate our approach and present the results of our experiments that demonstrate how our solution enhances the state of the art by increasing the detection rate, confidence in results and accuracy of OS fingerprinting.

## 3.1 Evaluation of Existing Solutions

### 3.1.1 Technical Features

The technical features of PRADS and p0f will be explained, so that a deep understanding of their workings is gained. This is not important only to evaluate these tools, but also because the techniques used by them are effective and will be re-used during the design of our solution.

PRADS is a system for passively discovering network hosts and determining some of their attributes. Numerous protocols are used for the recognition of the various attributes, but OS fingerprinting is performed using *transmission control protocol* (TCP[1]), *user datagram protocol* (UDP) and *internet control message protocol* (ICMP). In the project's website it is noted that "ICMP and UDP fingerprinting is not 100% accurate, but it gives an indication about the OS that can be used for building up a higher confidence level for a total OS detection/fingerprinting.". This suggests that TCP is mainly used for OS detection and the technique used by PRADS is called *TCP fingerprinting*. Specifically, various elements in the header of TCP packets are inspected, and a signature is extracted from each packet. Some elements from the IP layer are also added to the signature, which is then compared to a database which contains fingerprints labeled with OS families or variants, and the closest match is output as the OS of the host in question.

The elements of the TCP header that are used to create the fingerprints, along with which TCP packets will be used and how the matching will be done, can heavily affect the performance of the tool. The most common approach is to use only TCP packets that have the SYN flag set, which are the first two packets of the connection, because the elements in the headers of these packets are more affected by the sender's TCP implementation rather than the communication link or the receiver [31].

[1] https://en.wikipedia.org/wiki/Transmission_Control_Protocol

PRADS's TCP OS fingerprinting is based on p0f version 2 and uses the same fingerprint format and database of known OSes. Consequently, the elements of the header that are used are the same as in p0f, as the fingerprints in the database only carry that information. Specifically, according to various online sources about IPv4[2], TCP [32] and PRADS's source code[3], a fingerprint of p0fv2 and PRADS consists of the following elements:

- Initial TTL (iTTL), an estimation of the TTL as set by the sender upon departure of the packet, extracted from the IP layer.
- Do not fragment flag (DF), which dictates that no fragmentation should be done, extracted from the IP layer.
- Window size (WS), which determines the amount of data that can be sent before receiving an ACK.
- Overall packet size.
- Maximum segment size (MSS), the largest amount of data that TCP is willing to receive in a single segment.
- Window scaling (WSCALE), which allows increasing WS beyond its initial limit of 65.535 bytes to 1 gigabyte.
- Timestamp, sometimes used to determine the order in which packets were sent.
- Selective ACK (SACK), which allows the receiver to acknowledge discontinuous blocks of packets which were received correctly.
- No operation (NOP), acting as padding.
- End of line (EOL), sometimes used to signal the end of the options field.
- Quirks. This is a field that is used to hold a variety of observations about the packet. Not all of these unique features are found in each packet, so the format is flexible. Irregularities may include fields such as the URG or ACK fields (not flags) not being properly zeroed out by the sender TCP implementation, which can be used to identify the latter. A full list of the quirks can be found in PRADS's source code.
- OS genre, such as Windows or Linux.
- OS detailed description, such as (Windows) 7, 8 or (Linux Kernel) 2.6.

*P0f version 3* is the latest tool released by Michal Zalewski. P0f v1 was first published in 2000 and p0f v2 in 2003 and has since been re-used and ported into many fingerprinting toolsets, such as PRADS and Satori. P0f v3 is a complete rewrite of v2, improving TCP fingerprinting and allowing HTTP recognition along with provisions for other Layer-7 protocols. Due to the major changes, p0fv3 utilizes a new signature format, making it incompatible with old signatures. The features used by p0fv3 are the following. Features similar to the ones described before will not be analyzed again.

- IP version, which can be IPv4, IPv6 or "any".
- iTTL.
- Length of IP options.
- MSS.

[2] https://en.wikipedia.org/wiki/IPv4
[3] https://github.com/gamelinux/prads/blob/master/doc/p0f.fp

- WS.
- WSCALE.
- Options layout, referring to the TCP options used and the layout in which they appear.
- Quirks, similar to p0fv2.
- Payload size.
- OS genre and version.

For the rest of this document, p0fv3 will be referred to simply as p0f.

### 3.1.2 Performance

Our next goal is to identify which are the strong areas of each tool and how the tools compare to each other, when used in identical conditions. Initial tests show that each of the tools also has its own deficiencies and limitations, which we will explore in this section.

PRADS is quite verbose with regard to systems for which it provides an OS prediction, while p0f takes a more conservative approach, resulting in output about fewer hosts. In early tests that we performed over small datasets, we consistently observed that p0f is able to recognize the OS of less network hosts than PRADS. In our test lab environment, which consists of 7 hosts simulating a small ICS network, p0f was able to recognize 4 of them, while PRADS managed to give an OS prediction for 6. Possibly as a consequence, we observed that PRADS's output is less accurate and different OS predictions are often given for the same host, whereas p0f rarely contradicts itself.

To investigate this situation further, we ran both p0f and PRADS on a network traffic trace file (PCAP) which had previously been captured at a real ICS network. The results for 4 representative hosts (recognized by their IP addresses) have been selected and are presented below. For each IP we present the responses that each of the two tools provided. Because the tools output one OS prediction per suitable network packet, running them on a network trace resulted in thousands of predictions for each host. These results have been filtered and aggregated, so that only one entry per tool and per IP is shown. Furthermore, a percentage is given next to each OS prediction, to indicate what part of all the responses the prediction represents.

The following results show that the two tools generally "agree" that the related hosts are Windows and Linux systems respectively. What is also demonstrated, though, is that both tools sometimes struggle to give a concrete answer to the user.

- IP 1
  - PRADS
    1. Windows Win2K (UC)/Win7/2008R2 - 50%
    2. Windows 2000 SP2+, XP SP1+ (seldom 98) - 50%
  - p0f
    1. Windows 7 or 8 - 66%
    2. Windows XP - 33%
- IP 2
  - PRADS
    1. Linux 2.6 - 42%
    2. Linux recent 2.4/2.6 - 38%
    3. Linux 2 - 19%
  - p0f
    1. Linux 2.4.x-2.6.x - 99%

2. Linux 2.6.x - 0%

Moving on to different hosts, we can see that sometimes even p0f cannot confidently make a choice about whether a system is running Windows or Linux.

- IP 3
  - PRADS
    1. Windows XP/2000 - 66%
    2. Windows Win2K (UC)/Win7/2008R2 - 33%
  - p0f
    1. Windows 7 or 8 - 66%
    2. Linux 2.6.x - 33%

In the last example, about host with IP 4, PRADS provides a wide variety of 5 possibilities, one of which is just "unknown", with equal percentage to a valid one. It is obvious that the end user, a network operator or administrator, is not left with a confident, reliable OS prediction for each host.

- IP 4
  - PRADS
    1. Windows 2000 SP4, XP SP1+ - 30%
    2. unknown 30%
    3. MacOS 9.1 (OT 2.7.4) (2) - 16%
    4. Cisco LocalDirector - 13%
    5. Windows XP/2000 - 10%
  - p0f
    1. Windows XP - 70%
    2. Windows 7 or 8 - 30%

This behavior is not specific to a network topology or type of fingerprinted system, but has been observed in numerous networks. The generality of this problem led us to seek an improvement over the state-of-the-art OS fingerprinting tools, the instrumentation of which will be discussed in the next sections.

Our aim is to construct a solution that exhibits a high detection rate, equivalent of that of PRADS, while providing a narrow array of OS possibilities for each host, with confidence similar to that of p0f. In other words, it would be ideal if we could have a solution that combines the best behavior of the state-of-the-art tools, while suffering minimally from their limitations. The design and implementation of such an approach is described in the following sections.

## 3.2 Proposed Solution

In order to be able to obtain more predictions of higher accuracy, we propose an improved approach to passive OS fingerprinting. The overall workflow is similar to that of PRADS and p0f, using TCP and IP header fields of captured packets to create fingerprints, which are compared to a database of fingerprints labeled with OS version information. The engine that matches fingerprints collected on the wire to the labeled set is replaced completely. While p0f and PRADS utilize simplistic matching techniques of signatures, such as field-by-field comparisons and limited "fuzziness" or flexibility hard coded into the logic, we will be using a machine learning technique and specifically supervised learning. We chose to utilize the decision tree model to build a classifier using the labeled fingerprint set. The classifier is then used to classify new fingerprints into OS classes, providing a result about the related network hosts.

For practical reasons, one of the existing labeled fingerprint sets was used. If a new labeled set was to be constructed, we would have to gather fingerprints from dozens of different machines, running different OS families and versions. This task would require significant effort to be completed adequately, and it would still be next to impossible for a database created by a single researcher to match the breadth of existing datasets that have been enriched over years by various community members. Furthermore, abandoning existing work and creating an incompatible fingerprint format would only be justified if it would introduce significant performance benefits. Since there was no strong evidence that such an approach would be beneficial, we used the signatures from the latest release of p0f.

The selection of TCP/IP header fields that are used for matching is proven to greatly affect the effectiveness of the approach [22], so it must be done wisely. If too few fields are used, fingerprints of different systems may be identical, because there is not enough differentiation in values. If the fields used are too many, fingerprints may become too specific, leading to significantly lower matches, or the fingerprint database must increase in size to compensate. Similar problems arise if the right number of fields is used but the wrong ones are selected. Due to the fact that we re-use p0f's signatures, the TCP/IP header fields that can be used are limited to the ones present in the fingerprint list. Based on [22] we chose to use only the following elements, as they are suspected to be the ideal combination: MSS, WS, iTTL. Additional fields such as *options layout* and *IP version* are also added to our list, for different reasons. The options layout is a new "invention", introduced by p0f, and its effectiveness was not evaluated in the aforementioned work. The IP version is only added to act as a filter for IPv4 and IPv6 signatures that may be different from each other.

Our final goal is to match the fingerprints gathered on the network, which map to network hosts, to OS labels, which will provide us with an indication of what OS the hosts are running. The method chosen for this matching is *decision tree learning*. Given that the classes are finite, and consist of the OS names and versions, the model used is a *classification tree*. This method has been shown to be the most effective of four machine learning techniques, tested on fingerprint matching by Lippmann et. al [22].

In addition to TCP fingerprinting, we will be utilizing observations made in [19] about OS information that is present in *server message block* (SMB) and *Microsoft Windows browser protocol*. When these protocols are used and these fields are populated, our algorithm will be able to quickly extract a precise OS match without the need to rely on TCP fingerprinting. Early experiments show that SMB traffic in ICS networks is not transmitted by enough hosts to constitute a reliable source for OS information by itself. This situation is depicted in Table 3.1, in which the results from a scan for useful SMB packets are shown. The duration of the datasets is between one and two days, and the networks were highly populated with over 100 active network hosts each. Despite the high amount of traffic, dataset X only contained useful OS information about two hosts over the entire period of almost two days, while dataset Y contained no useful SMB packets at all. This is unfortunate as the quality of SMB OS recognition is theoretically very high so it could constitute the primary OS detection method in networks with heavy SMB utilization.

| Dataset | Duration | Network hosts | Relevant SMB packets | Identified Hosts |
|---|---|---|---|---|
| X | 1 day, 23 hours approx. | 121 | 79 | 2 |
| Y | 1 day, 16 hours approx. | 346 | 0 | 0 |

Table 3.1: Coverage of SMB protocol in two datasets.

Despite its limitations, the method of SMB OS parsing provides highly accurate readings and therefore it is included in our algorithm as a supplementary OS information source. The specific messages that contain OS information are of type *SMB Session Setup AndX*. Because these messages are not very common, one can also scan the text within SMB packets for strings that start with the *Windows* keyword. Obviously, this method will only recognize Microsoft Windows OS versions and may also include false positives, so its results are only offered to the

user to be used after inspection.

There are additional messages that carry OS info, such as *Microsoft Windows Lanman Remote API Protocol* over SMB over NetBIOS. This source contains fields of fixed format, where the OS major and minor versions are given. For example, Windows 8 or Server 2008 R2 are coded as Windows kernel version 6.2, so the relevant fields in SMB would indicate a major version of 6 and minor of 2. Despite the apparent advantages of these packets, we will intentionally not use them to avoid a pitfall discovered by Kollmann [19]. In fact, along with the OS major and minor versions, the packets include *Browser Protocol* major and minor versions. Ignoring the browser protocol version has been shown to lead to wrong interpretations of OS version, because the same OS version must be interpreted as different actual OS, when different Browser Protocol versions are used. To sum up, in order to obtain a good result out of packets of this type, one would have to have a database of OS and Browser Protocol versions and the related actual OS versions. This database would look similar to a TCP fingerprint database and since, according to our knowledge, it does not already exist we decided not to use these SMB packets at all.

## 3.3 Implementation

The solution outlined in Section 3.2 has been implemented as a prototype, so that its effectiveness could be tested in practice. The process consists of two phases, carried out by two distinct components. First, network trace files, in the PCAP format, are imported into a script which parses each packet and extracts the most useful findings. Parts of this data, namely the fingerprints of TCP packets, are saved to a file on disk. The file is then read by a separate process line-by-line. Each line contains a TCP fingerprint captured from a packet that was sent from a specific network host. A classifier is then used to predict which OS label best fits the fingerprint, completing the OS detection process.

### 3.3.1 Phase 1: Information Extraction

Regarding the *first phase*, the algorithm was implemented using the *Python*[4] programming language, due to its versatility, conciseness and availability of libraries. Specifically for manipulating network trace files (PCAPs), the tool named *Scapy*[5] is imported as a library and used to dissect and parse network packets.

Any packet that contains TCP layer information is checked for the SYN flag, which indicates that the packet is one of the first packets of a new connection. This is essential because, as explained in Section 3.1, these packets contain the information that best characterizes the sending host, such as the MSS and TTL. Once the packet is rendered to be useful, it is further processed to extract all the relevant fields, which are the following: MSS, WS, options layout, IP version and TTL, as explained in Section 3.2.

These fields could form a TCP fingerprint, but using them as-is would lead to too specific signatures, which would not be effective. Instead, some of the values need to be filtered or normalized, to better serve as generic identifiers. Specifically, the following pre-processing is performed:

- *WS*. According to p0f's documentation[6], some OSes set window size to a multiple of MSS, MTU or a random number. Thus, upon capturing a packet we check if this is the case. A series of test calculations (divisions by MSS, MTU plus and minus the size of the whole header) are carried out, and if one of them results in a perfect division, the value of WS is changed accordingly. For example, if WS is found to be 4 times the MSS value, the WS field of the signature is substituted with the string "mss*4".

[4] https://www.python.org
[5] http://www.secdev.org/projects/scapy
[6] http://lcamtuf.coredump.cx/p0f3

- *TTL*. Most OSes set the initial TTL value of a packet they transmit to a power of 2 [19]. Usual values are 64, 128 and 255. Consequently, it is beneficial to "normalize" TTL values of captured packets before using them in fingerprints. Since a TTL is only decreased as the packet travels through the network (being decremented by one each time a router is traversed), it only makes sense to correct the TTL upwards. This process is completed before the TTL is stored in the final fingerprint.

Finally, each fingerprint is written to an output CSV file for further processing. This is achieved using *python's csv* module[7] which enables import, manipulation and export of *comma separated value (CSV)* files. This specific CSV file is our *testing set*, namely a collection of fingerprints that will be given in input to our classifier that will predict an OS label for each fingerprint.

A secondary feature of the *first phase*, done in parallel with the aforementioned actions, is to parse SMB packets. As discussed earlier (Section 3.2), some SMB messages contain OS information that is highly accurate and should be salvaged when found. This is done simply by parsing specific bytes of an SMB message to check if it is of the relevant type, and then parsing the block that according to the standard contains the sending system's OS version. As an easter egg, the hostname of the system is also checked for and extracted upon existence.

### 3.3.2 Phase 2: Fingerprint Matching

The *second phase* of the algorithm is responsible for the final fingerprint matching. The aim of this process is to predict the best OS label for each fingerprint, hence to assign an OS to every network host (identified by IP address). To help us in this task, we use the RapidMiner platform[8] which includes implementations of most major machine learning algorithms. The flow of the process is shown in Figure 3.1 and described below.

- The top branch takes a fingerprint *training set* as input, which consists of a list of fingerprints, each one tagged with an OS label. These fingerprints are extracted from p0f's signature database, which we have previously parsed and converted using a script to match our needs.
- The training set is given as input to the decision tree learner. This learner produces a classifier (a *model* in RapidMiner) that is able to predict a OS label for each fingerprint given in input. Note that the model only needs to be learned once; in a real deployment every time a fingerprint is available, the model can be queried to predict the OS label.
- In RapidMiner, the model can be sent to both the user for inspection and to the *Apply Model* block (see Figure 3.1) that is the component in charge of predicting a label for each fingerprint given in input.
- The second branch (lower part of the figure, left side) starts by reading the *testing set* which contains the fingerprints of packets captured in *Phase 1*.
- The attributes of the fingerprints that will be used for matching are then selected. According to earlier analysis in Section 3.2 we use only MSS, WS, iTTL, options layout and IP version.
- The *testing set* is finally fed to the *Apply Model* operator, which predicts an OS label for each fingerprint in the *testing set* by applying the model learned by the decision tree learner.
- The results of the process are written to an output CSV file for evaluation purposes. Each line of the file contains the original TCP fingerprint, the IP of the host that is relevant and the OS label that has been chosen as a prediction.

The full decision tree that results from the first branch of the RapidMiner process cannot be presented as a whole due to space constrains, but a part of it can be inspected in Figure 3.2. Each rectangular with rounded corners represents an element of the TCP fingerprint, and depending

[7] https://docs.python.org/2/library/csv.html
[8] https://rapidminer.com

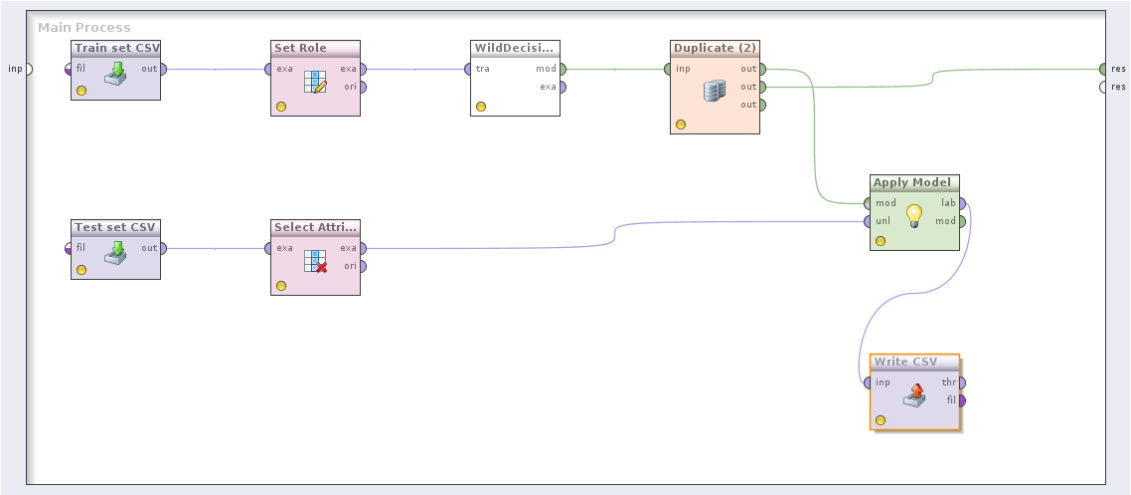



Figure 3.1: Overview of the RapidMiner block diagram that trains and uses the decision tree classifier.

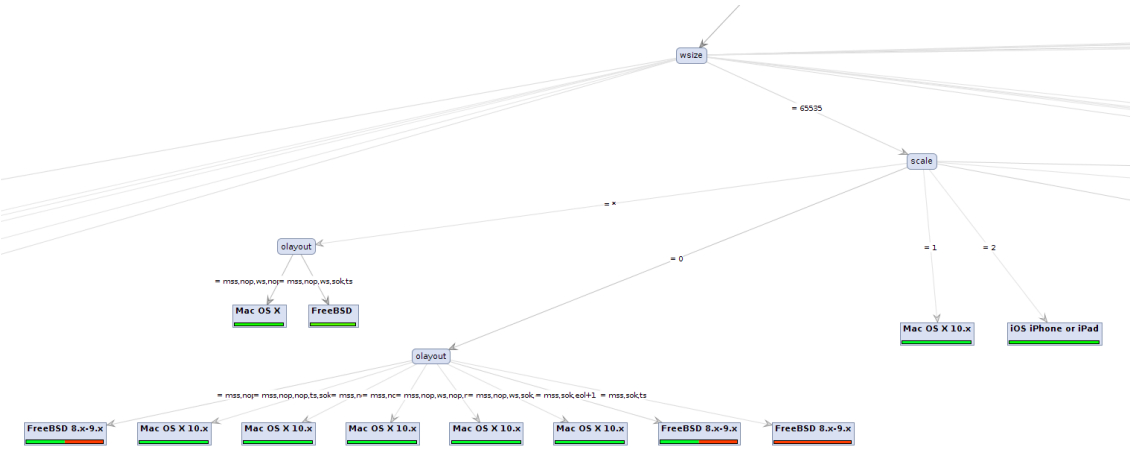



Figure 3.2: Part of the decision tree, generated using p0f's fingerprint database. Nodes with rounded corners represent TCP/IP header elements that are used to reach the leaves of the tree, which represent OS predictions.

on the numeric value, which is noted on the respective edge, a child is chosen. When a leaf of the tree is reached, an OS label can be output as the prediction of the algorithm.

The existence of a special kind of value, the asterisk ("*") is particularly interesting. Traditionally, a decision tree is constructed using specific values found in the training set. P0f's signatures though, contained the asterisk character as a wildcard, to indicate that for some field of the fingerprint, any value can be accepted. This concept is fundamentally simple from a programmer's perspective; a wildcard is an easy method to indicate that anything should match. In contrast, the same concept creates logical problems when used within a classifier, because it alters the logic of the matching non-linearly. The standard implementation of decision tree handled the asterisk as a normal character, which led to poor results, as this character is never to be found in fingerprints extracted from packets on a real network. As such, we modified the algorithm to use the asterisk the way p0f intends it to be used, as a wildcard that matches anything. Effectively, this cancels out the field in which it is used and renders it inactive for the specific fingerprint.

In order to make the interpretation of the results easier, a post-processing script was used. The script goes through the CSV that RapidMiner creates, and parses each IP and respective OS prediction. Additionally, it counts how many times each OS prediction appears for a given IP and then presents the most probable prediction first, along with a confidence percentage, to provide the user with meaningful information.

## 3.4 Evaluation

In this section we will evaluate the performance of the algorithm we described previously. The implementation that was created was used on two datasets that contain traffic captured on the networks of two different, real industrial control system networks. First, we define the metrics that will be used, and then we present and interpret the data from both datasets for each metric separately.

### 3.4.1 Metrics

To evaluate the performance of our solution we will make use of the following metrics:

- *Detection count* is defined as the number of network hosts, identified by IP, that an OS prediction has been made for. Multiple predictions for the same host are only counted once. In essence, detection count shows how many systems were recognized by the tool.
- *Accuracy* is the ratio of OS predictions that are known to be correct over all of the predictions made by the tool.
- *Average Predictions per host* shows how many OS predictions, on average, a tool has made for each network host. A high figure translates in the tool giving many possible answers for each host, which is a sign of low result quality.
- *Average Confidence* is the average of the confidence levels of all the OS predictions a tool has made.

Because all three tools, p0f, PRADS and our prototype work on a packet-by-packet basis, without saving and re-using any state information, they do not natively support confidence readings. The only output the user gets for each packet is an OS prediction, without confidence level. To overcome this, we created a script that measures how many times each OS prediction is given for each host and calculates the confidence for each reading and host. The average confidence is calculated as the average of all the individual figures.

### 3.4.2 Results

#### Detection

Table 3.2 contains the count of the hosts for which at least one OS prediction has been made. For each tool examined, we provide the results from both datasets. Our tool detected 270 and 495 hosts in the two datasets. W.r.t. the first dataset, our tool recognized the most hosts, achieving the 100% score in the last column. For dataset 2 the score is lower, 97.44%, indicating that the performance is very close to the best tool.

Concentrating on the percentages column, and looking through the rows of the other two tools, we can note that in most cases high percentages have been achieved, indicating that the tools are roughly equivalent in detection rate. The exception of p0f must be noted, which in dataset 1 managed to detect less than half of the hosts that the leading tool (ours) did.

To sum-up, all three tools can detect an equivalent amount of hosts, while p0f's and PRADS's worst case rates are lower (47.41% and 94.44% respectively) than our tool's, which always scored over 97.44%.

| | | Detection | |
|---|---|---|---|
| | | # | % |
| Our Tool | Dataset 1 | 270 | 100.00% |
| | Dataset 2 | 495 | 97.44% |
| P0f | Dataset 1 | 128 | 47.41% |
| | Dataset 2 | 500 | 98.43% |
| PRADS | Dataset 1 | 255 | 94.44% |
| | Dataset 2 | 508 | 100.00% |

Table 3.2: Host detection rates for all three tools and two datasets. The first of the *detection* columns refers to the absolute number of hosts that have been recognized, while the second indicates how many of the maximum recognized hosts (by any tool) were recognized for the given tool and dataset.

### Accuracy

The accuracy of the OS predictions of our tool is central to its effectiveness. If the tool's output is not correct, there is no use for it, regardless of its good performance in other metrics. To estimate the accuracy we selected hosts from each dataset randomly, and asked the operators of each network to verify what OS is running on those systems.

As a preliminary test, we chose 271 hosts from dataset 2 and obtained the ground truth for them. Subsequently, we run our tool on the dataset and it managed to successfully identify the OSes of 271 out of 271 hosts, which translates to an overall accuracy of 100%.

Next, we randomly selected 15 of these hosts and measured the performance of the other two tools, p0f and PRADS. The results of this test are shown in Table 3.3. Calculated over the total, the overall accuracy of each tool as a percentage of all hosts is shown in the last row. In this dataset, p0f and PRADS did adequately well but still worse than our tool which achieved the highest accuracy possible.

| | Our tool | p0f | PRADS |
|---|---|---|---|
| Correct OS predictions | 15 | 12 | 11 |
| Correct OS prediction percentage | 100.00% | 80.00% | 73.33% |

Table 3.3: OS prediction accuracy, for three tools, tested on 15 hosts of dataset 2.

Next, we selected 18 hosts from dataset 1, for which ground truth was retrieved. The tools successfully recognized the OSes of a subset of all the hosts, as shown in Table 3.4.

| | Our tool | p0f | PRADS |
|---|---|---|---|
| Correct OS predictions | 9 | 10 | 15 |
| Correct OS prediction percentage | 50.00% | 55.56% | 83.33% |

Table 3.4: OS prediction accuracy, for three tools, tested on 18 hosts of dataset 1.

The score of our tool is remarkably low, which is unsettling, but investigation showed that this nonperformance can be attributed to a distinct event. Six of the eighteen hosts were identical systems, running the same OS, Solaris 10. Unfortunately, every single host running this OS was erroneously recognized as Linux kernel 3.11+ by our tool, ruining its accuracy performance. These results underline the fact that a single bad fingerprint match, recurring over and over, can have a significant effect in a tool's performance. Nevertheless, this weakness is an isolated incident and does not characterize the overall performance of the algorithm. If that signature was handled properly, our tool would have identified 15 OSes, bringing it up to par with PRADS.

### Average Predictions and Confidence

To further evaluate the performance of the OS fingerprinting tools, we discuss the *average number of predictions per host* and the *average confidence* of each tool, for each dataset. Each 4-row part of Table 3.5 relates to data coming from a different dataset.

The first two lines about dataset 1 contain the total of network hosts for which OS predictions have been made (*Detected hosts*) and the total number of predictions for the whole dataset (*All predictions*). Dividing the later by the former gives us the *average predictions per host*, presented in the third line of the group. A lower number in this row indicates that the respective tool provided the same answer more consistently, when different packets from the same host were processed. Intuitively, we can note that this translates to the tool "changing its mind" less, than if the number is higher. In the first three lines of dataset 2, the equivalent metrics are calculated. Let us delay discussion about *average confidence* until we analyze the rest three metrics for each dataset.

| Dataset | Metric | Our tool | p0f | PRADS |
|---|---|---|---|---|
| Dataset 1 | Detected hosts | 270 | 128 | 255 |
| | All predictions | 288 | 247 | 464 |
| | Avg. predictions per host | 1.07 | 1.93 | 1.82 |
| | Avg. Confidence | 97.51% | 70.73% | 67.40% |
| Dataset 2 | Detected hosts | 495 | 500 | 508 |
| | All predictions | 502 | 970 | 1304 |
| | Avg predictions per host | 1.01 | 1.94 | 2.57 |
| | Avg. Confidence | 99.97% | 90.86% | 55.91% |

Table 3.5: Average predictions per host and confidence in predictions for all three tools and two datasets. The number of detected hosts and all OS predictions made are included to provide perspective. Lower average prediction per host and higher average confidence are better.

In both datasets, our tool exhibits the lowest number of predictions per host, indicating that it provides the fewest possibilities of OSes per host, on average. Furthermore, the fact that the number is very close to 1, for both datasets, means that in most cases only one prediction is provided to the user, which is a positive characteristic of our tool.

The last metric we present in the table, is the *average confidence* each tool has in its predictions. This metric is calculated separately for each tool, by averaging the confidence of each prediction made for any host, whether right or wrong. Intuitively, a high percentage means that the tool is on average almost completely confident in the OS prediction it outputs to the user. Lower percentages indicate that the tool had trouble effectively evaluating the possible predictions against each other. In a situation like this, the tool may output the prediction (for example) "Microsoft Windows 10" with confidence 51% while the second in line was "Microsoft Windows 8" with confidence 49%. A user that reads the tool's output cannot, in turn, have a lot of confidence in such a result.

Reading the respective lines for both datasets, we can conclude that our tool scores highest in all cases, while the second tool (p0f) scores significantly lower. This figure can be intuitively connected to the low *predictions per host* metric described before, because being confident about a result enables the tool to predict the same OS for a host consistently. Finally, PRADS, which we have suspected to be "changing its mind too often" during preliminary tests, is now proved to exhibit this kind of behavior, with average confidence reaching as low as 55.91% for one dataset.

### SMB Parser Evaluation

The performance of the part of the algorithm that parses SMB messages was evaluated using the same datasets. For dataset 1, the SMB parser recognized the OSes of 9 hosts, but we were unable to obtain ground truth for any of them. Regardless, the fact that the TCP fingerprinter (discussed

previously) predicted OSes for 270 hosts while SMB was only available for 9, supports our preliminary experiments (Section 3.2) and assumptions about SMB not being a fully dependable solution in ICS networks. Running the SMB parser on dataset 2, which consists of over 500 identifiable devices, only yielded 6 OS predictions, which we were able to verify and were found to be correct in reality.

**Take-aways**

To sum up, we will be making a few notes on our results and contributions, that will provide additional insight.

The comparison we did on the state-of-the-art tools showed that PRADS has the upper hand over p0f in detection rates, recognizing the most hosts in all situations tested. In contrast, p0f is superior in the confidence it has in its predictions, while PRADS scores much lower in some cases, verifying our assumptions about its "uncertainty" in its results. The accuracy of the predictions of the two tools was shown to be roughly equivalent, with each one of them surpassing the other in one of the two datasets. These observations enable us to conclude that previously to the existence of our solution, there was a trade-off between how many hosts a tool can detect and how certain about its prediction it will be. Increasing one metric would lead to the decrease of the other, while accuracy roughly stands unaffected.

With the introduction of our tool we managed to break this equilibrium, shifting the metrics' interdependence. The new tool is able to detect more or the equivalent amount of hosts (2.66% less) to what the top tool in detection (PRADS) can while having significantly higher confidence in its predictions than p0f! At the same time, prediction accuracy is better (for dataset 1) or slightly worse (for dataset 2, despite our tool exhibiting bad behavior due to an isolated incident) than p0f, rendering it reliable enough for practical use. If all the aforementioned parameters are added to an overall, weighted score, we believe that we have created a tool that successfully propels the state of the art towards better results for the end user. The experience of the network administrator will be qualitatively better, as they will receive more consistent predictions, about more devices on their network, at the same or better accuracy than they used to. This advancement strengthens the position of passive OS detection as a tool that can be realistically used to detect safety and security hazards in ICS networks, such as misconfigurations or rogue devices.

# Chapter 4

# Topology Discovery

Network administrators of large industrial control systems (ICS) experience difficulty keeping an accurate and up-to-date diagram of all the operational devices. The multitude of network hosts and complexity of the network make the task of manual book-keeping extremely resource intensive. At the same time, knowing which devices are supposed to be connected to a network, and what kinds of communications between them are considered normal can be crucial for the timely and effective mitigation of safety and security incidents that can occur.

While the above problem exists in most complex IT networks, operators in ICS environments are confronted with additional difficulties, specific to those networks. Actively querying systems is often dangerous for the industrial process itself, and given that critical infrastructure may be involved, the stakes are high.

Along with these restrictions, ICS networks also exhibit some characteristics that may aid automated enumeration tasks; because many of the communications happening on the network are parts of automated processes, most connections are characterized by regularity. This particularity has different implications, depending on the specific network and the reason for which the topology needs to be known. For example, one can possibly exploit the fact that the hosts of an ICS network do not change significantly over time to issue alerts once new ones appear, or employ the same logic on connections between existing hosts.

Currently available tools for network topology discovery, as discussed in Chapter 2, do not solve the problem adequately in an ICS setting, either because they are active and unaware of ICS protocols or because they require features such as internet connectivity, rendering them inapplicable in the case of many ICS networks.

In this chapter, we will present a method that can be used by or be incorporated within a passive monitoring system such as an intrusion detection system (IDS), to enable the automatic creation of topology maps of otherwise unknown and uncatalogued networks. The principles of the method are described in Section 4.1 and the detailed algorithm is outlined in 4.2, while an implementation of the algorithm is described and evaluated in 4.3.

Before continuing into the subject matter, we should pro-actively define some concepts that will be used frequently.

- A *fully defined subnet* refers to a part (subnetwork) of an IPv4 network that contains IP hosts. The format used to define a subnet is the first address of the subnet, followed by the slash character (/) and the bits of the network prefix, also known as the CIDR notation[1]. A typical example for a LAN would be 192.168.0.0/24. A subnet is said to be fully defined when both the first IP address and the network prefix are known.

- A *network host* or simply *host* is a computer system that is connected to an IPv4 network. A host may have multiple network interfaces, attached to different subnets.

[1] CIDR refers to Classless Inter-Domain Routing, which has been explained in more depth in Section 2.2.

- A *network asset* or simply *asset* refers to a network interface of a host. Although it is usually the case, we avoid tying an asset to a host because the latter may have more than one network interfaces and when viewed from the networks' perspective it is not trivial to identify that the interfaces belong to the same host (a problem known as aliasing, described in Section 2.2).
- The *L1 to L7* notation refers to Layers 1 to 7 of the OSI model, as explained in Section 2.2. To jog the reader's memory, we briefly remind that MAC addresses are L2 elements, while IP addresses are L3.

## 4.1 Solution

In this section, we describe a technical solution that enables automatic, passive network topology discovery. In order to achieve high performance, we will use ideas and techniques that have either been shown to be effective in bibliography or their perspectives look promising based on our observations and preliminary tests on real ICS network data. The methods used to obtain topology data are separated in two sections. The first is about exploiting properties and information of basic networking protocols such us Ethernet or IPv4, while the second deals with more specialized and advanced protocols what useful information they carry.

### 4.1.1 Techniques Based on Basic Protocols

By understanding how network topologies work in regard to the lower OSI layers (up to L3), we formulated rules that can be used to make conclusions about the topology of a network.

In local area networks (LANs), network hosts are assumed to be able to reach each other directly, only being separated by L2 devices such as switches. Ethernet frames are thus delivered only based on their 48-bit MAC addresses and no information from higher layer protocols needs to be used. We will consider a LAN to be *tapped* when frames are captured by a host that is connected to it. The host will in turn be referred to as a *network tap* or *sensor* or simply *tap*. In some cases, when tapping takes place for network monitoring purposes, the sensor is attached to a special port of a switch that is connected to the LAN. The port is known as a *mirror port* or *Switched Port ANalyzer (SPAN)* on Cisco Systems switches, and provides the host connected to it with a copy of each packet that is received by the switch on any of its other ports, regardless of whether the packet was destined for the host.

Based on knowledge of basic network protocols, we have identified the following rules that can enable topology discovery:

1. When a sensor is connected to the mirror port of a switch on a LAN, it will receive a copy of each frame that goes through the switch. This copy will be intact, in the sense that it will include the original ethernet MAC addresses. In contrast, when a packet from a different LAN arrives to the monitored LAN (destination LAN), its ethernet part is modified by the router that is located at the edge of the two LANs. The router substitutes the source MAC address with its own and the destination address with the one matching the recipient in the destination LAN. Higher layers, such as the network layer (L3) which may contain IPv4 addresses, are not modified[2]. The above observations enable us to compose the following rule:

   > When different frames are captured on a given LAN and their ethernet source MAC addresses match, while their IPv4 source addresses differ, the packets most probably originate in a different LAN. Furthermore, the MAC address may belong to the local interface of a router, which is located at the edge of the monitored LAN.

[2]Given that network address translation (NAT) is not in use.

2. Information can also be derived from the network layer. Taking the case of the most popular protocol at this layer, IPv4, we have observed that a part of the header called Time To Live (TTL) may help us make useful conclusions. Specifically, TTL is initially set to a high value and is decremented by one each time the packet traverses a router. If the value reaches 0 the packet is dropped. While this feature was built as a safeguard against forwarding loops in problematic network topologies, it can be exploited for our purposes. When a packet is captured, we can calculate the possible initial TTL value as set by the sender and then subtract the current value from it. The resulting figure corresponds to the network distance between the sender and the sensor, and it is referred to as *network hops*. For example, if a packet with a TTL value of 125 is received, we can increment the value until the closest power of 2 is reached ($2^7 = 128$). Subsequently, subtracting the initial value (125) from 128 will give 3, which can be read as "the packet comes from a host that is located in a LAN 3 hops away from the sensor/tap". It must be noted that the calculation of the initial TTL is an estimation that is based on the observation that most operating systems set the TTL to powers of 2, as explained in Section 2.1.

3. Additionally, a heuristic can be devised to detect how many LANs are being tapped by the same sensor. This may be useful because a sensor with multiple interfaces may be capturing packets on different LANs and saving them in one trace file, or because a switch or other network device may be aggregating packets from different LANs before serving a copy of each to the sensor. To recognize these situations, one can check if the same packet appears in a dataset more than once. In that case, the number of occurrences indicates the number of sensors or taps that are in use. Furthermore, the network distance between two different LANs can be estimated, given that both of them are tapped. When the same packet[3] is captured by both taps at the different LANs, the absolute difference of their TTL values will be equal to the hop distance of the two LANs.

4. Address Resolution Protocol (ARP) packets can help with associating MAC and IP addresses, as this is the main purpose of the protocol itself. The same effect can be achieved by observing most other IP packets and the underlying ethernet frames, so this method does not provide much added value.

5. Active scanning techniques like traceroute and ping probes can provide quick and reliable insights into the network but they are active and can even create negative consequences such as IDS false positives and device reboots in ICS networks. Furthermore, they are not applicable when captured data is analyzed off-line, so their value is limited to our purposes.

The last two techniques in the above list were not considered effective and in line with our objectives and were thus not utilized.

### 4.1.2 Techniques Based on Specialized Protocols

Apart from basic, lower layer methods a series of higher layer protocols may typically contain clues about the topology, so harvesting them may be beneficial. A short overview of each source, along with applicability motivation is provided below.

- In networks monitored using simple network management protocol (SNMP), information related to specific devices or the subnet may be available. Although SNMP is widely used, the data format is not defined by the protocol but by the management information bases (MIBs) instead. Since different MIBs may be used in different networks, to use SNMP information one should be able to parse many different MIB, which requires substantial effort and increases the complexity of the solution. In addition, as discussed in Section 2.2, the use of SNMP comes with other disadvantages including the need for authentication (i.e. credentials need to be available) and additional traffic injection due to the active nature of the approach. For these reasons, this method is not adequate for our purposes.

[3]Contents above and including L3 should be equal. L2 addresses are expected to be different.

- External routing protocols such as border gateway protocol (BGP) can supply information about the interconnection of autonomous systems (AS). As defined in 2.2, an AS can be summarized as a set of routers and other hosts that use an interior gateway protocol and common metrics to route packets within the AS, and an exterior gateway protocol to route packets to other ASes. In our case, external routing protocols can provide a limited amount of useful information, as the problem we are trying to tackle involves discovering topologies within ASes and not between them.

- Internal routing protocols such as routing information protocol (RIP), open shortest path first (OSPF) and interior gateway routing protocol (IGRP) are used for communication of network related data between routers.

- Simple service discovery protocol (SSDP) facilitates the advertisement of services running on network hosts and constitutes the basis for the universal plug and play (UPnP) protocol. Unfortunately, exchanged messages do not include enough information to fully define an L3 subnet.

- The Cisco discovery protocol (CDP) can provide information about the existence and capabilities of neighboring hosts.

- Spanning tree protocol (STP) and rapid spanning tree protocol (RSTP) are used by ethernet switches to determine which links should be used for communication within a LAN while avoiding loops. Although these protocol messages contain useful L2 information, they do not typically provide information about the IP destination and host (L3), so their suitability was not investigated further.

Internal routing protocols RIP, OSPF and IGRP as well as CDP have been considered for use with regard to the information they provide and their use in ICS environments and have been found to be suitable. These protocols are regularly used to exchange messages that contain information about routers or other hosts and the respective subnets they are connected to. This information is usually adequate to identify the hosts that use them and the respective fully defined subnets.

To verify the feasibility of a completely passive approach to network topology discovery and to gather information about how effective such an approach could be, we decided to focus on the analysis of a single protocol. Because not all protocols are used by all networks, it is important to select the ones that are more frequently used in ICS environments, maximizing the possibility of being able to gather valuable information.

In order to select the protocol on which we will focus our attention, we carried out a basic experiment to see which of the protocols that we discussed earlier is more prevalent in ICS environments. To this end, we used eight different datasets, consisting of traffic captured in real ICS networks and we counted how many datasets contain each of the protocols we are considering. Table 4.1 shows the result of such a test. As we can see, w.r.t. the data we are considering, SNMP is the most commonly used protocol but we are discarding its usage for the reasons listed above. Additional considerations for the final decision included the fact that some of the protocols are vendor-specific. A prominent example is Cisco's CDP, which we avoid using as it would restrict the environments in which our approach will work effectively to networks using Cisco equipment.

Our final choice was to parse OSPF Version 2, which is an open standard defined in RFC 2328 [29] but is also used by large vendors such as Cisco. Furthermore, OSPF is believed to be the most used interior gateway protocol in large enterprise networks[4], which makes it ideal for our purposes.

The information carried by OSPF's *hello packets* includes the IP address of the router transmitting the OSPF packet, the mask of the subnet it belongs to, the backup router of the area and zero or more neighbors that are also OSPF-capable. Based on this, we can identify which devices are routers, and in which subnet they are located.

[4] https://en.wikipedia.org/wiki/Open_Shortest_Path_First

| Protocol | Detected in # of datasets |
|---|---|
| SNMP | 4 |
| CDP | 3 |
| STP | 3 |
| OSPF | 2 |
| SSDP | 2 |
| RSTP | 0 |
| RIP | 0 |
| IGRP | 0 |

Table 4.1: Protocol presence in eight different datasets.

Given that OSPF packets do not traverse IP routers, they never travel more than one hop. This is ensured by setting the IP layer's TTL value to 1 upon transmission of an OSPF packet. Consequently, when such a packet is captured, we can safely assume that one of the sensors resides in the subnet of the router. Using the network mask provided in the packet, we can fully define the network segment that the sensor is located in, which is a very important piece of information for topology discovery.

## 4.2 Algorithm

Based on the observations presented in the previous section, an algorithm that uses them to discover a network's topology has been designed and implemented. The approach is bound by the following assumptions and limitations.

**Assumptions.**

1. **Unknown Sensor Number.** The algorithm is designed to work with data that has been or is being captured using an unknown number of network sensors or taps connected to different LAN segments. The sensor is assumed to be connected to the mirror or SPAN port of a network switch and as a result is expected to receive a copy of each packet that travels through the given switch. Despite this requirement, it is realistic to expect that the mirroring function of the switch is not perfect, resulting in some packets not being forwarded to the sensor. One or more sensors connected to different LANs may be used at the same time, so the algorithm must be able to cope with this situation and differentiate the LAN each packet originates in. The number of sensors used and their location need to be detected automatically.

2. **Completely Passive Operation.** Traffic is assumed to be captured in a completely passive way. Ethernet frames captured must have the original destination address in place, and the sensor's hardware MAC address should not be found in their contents.

3. **No NAT in place.** Network address translation (NAT) must not be in use within the network.

4. **Aliasing.** Each network host may have more than one network interfaces, on different or the same LANs. The interfaces will be treated as independent assets, and *alias resolution* (as introduced in Section 2.2) will not be performed.

**Consequences of the Assumptions.**

1. Hosts that do not transmit ethernet frames during the observed time period will not be detected.

2. Routers that do not use OSPF may or may not be recognized.

3. LANs in which no router is using OSPF will not be fully defined.

**Network Entities and Data Structures.** The algorithm examines captured network packets sequentially, and saves the relevant information for further processing and use. The main entities and their relationships are shown in Figure 4.1 and analyzed below.

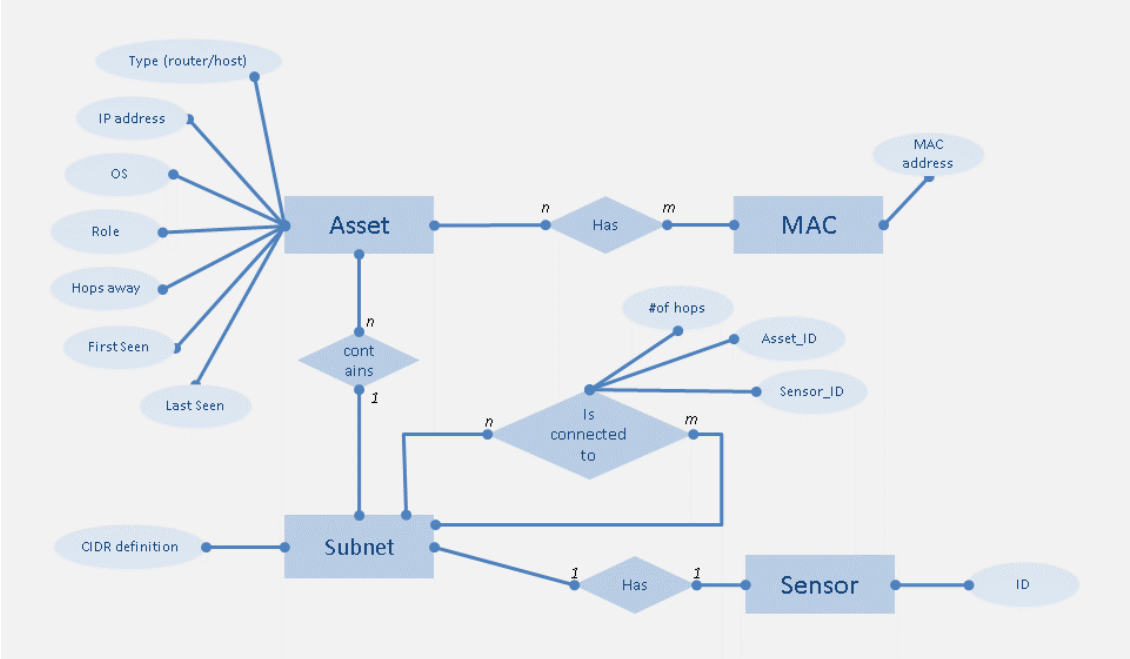



Figure 4.1: Entity-relationship diagram of the information that the algorithm produces.

- *Assets* are the identified network interfaces and mainly characterized by their main attribute, the IP address. Other attributes include the type of the asset (whether it's simply a network host or a router etc), operating system, the role of the asset in the network, the times that the host was first and last seen on the network, as well as the network distance between the sensor and the asset in hops. As mentioned in the definition of an asset, in the beginning of this Chapter, more than one assets may relate to one network host, so an asset does not always match a host one to one.

- *MAC* addresses are closely related to assets, but do not necessarily match them. MAC addresses are only detected in the tapped LAN(s) and each one of them can be connected with many assets. This situation is normal because packets that originate in different LANs will have the same source MAC but different IPs when they reach the local one.

- *Subnets* refers to the IPv4 subnetworks that have been identified and fully defined. A subnet can have many assets belonging to it. There can also be relationships between different subnets, known as interconnections between exactly two subnets. These interconnections are characterized by the two subnets involved, their network distance measured in hops and the asset that connects them (belonging to at least one subnet of the two).

- *Sensors* or network taps are the physical or virtual devices that are used to capture network traffic. Each sensor resides in exactly one subnet.

The algorithm examines captured network packets sequentially, and saves the relevant information in memory. A number of lists are used to save the state of the algorithm, but periodically a part of those lists can be exported to a database for further use. Information that is exported is considered unlikely to change, whereas some of the lists in memory contain states that may change over time, when new observations are made, and are primarily used by the algorithm itself.

As noted before, while assets can be thought to map to network hosts, they may technically differ. This is due to the fact that a host may have multiple network interfaces, each with different IP addresses, and will thus occupy multiple places in the assets list. Still, for the most part, we can loosely think of assets as network hosts.

**Algorithm Tasks.** The principle of operation of the algorithm is presented in Figures 4.2, 4.3 and 4.5. The algorithm can be run continuously on live network traffic, or in batch mode, when recorded traffic is replayed. For each packet, the process shown in Figure 4.2 is run, which contains all the necessary functions to extract useful data from the packet.

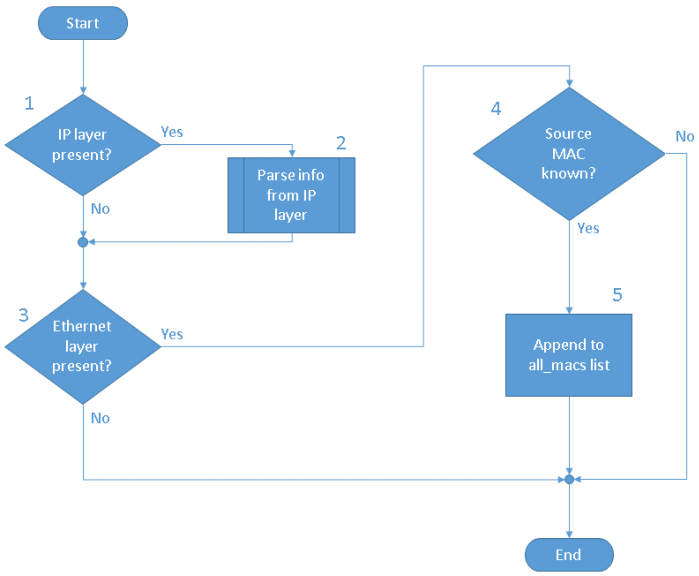



Figure 4.2: Overview of the main *topology discovery* subprocess. Decision blocks are represented as diamonds, while processes are noted in rectangles.

The main purpose of the subprocess in Figure 4.2 is to check for IP and Ethernet layers (blocks 1 and 3), and run appropriate actions or subprocesses. Conditional block 4 checks whether the source MAC of the packet is already in the list of known MACs that is stored in memory. If this is not the case, block 5 saves the MAC to the aforementioned list. The subprocess *Parse info from IP layer (2)* is complicated and involved, and its details are thus presented separately, in Figures 4.3 and 4.5. These figures contain the main workhorse of topology discovery which parses the IP layer of each packet. The main tasks carried out can be outlined as follows.

- In Figure 4.3, block 6, the network distance, measured in hops, is estimated based on the packet's TTL. The exact method used has been outlined in Section 4.1.1.
- If OSPF is present (block 7), and the message being processed is a *hello message* (8), information such as the IP, subnet mask and neighboring routers is parsed (9). The hello message is the one that contains the most useful pieces of information and that is why it is the only one we need to parse. An anonymized sample of hello message that has been

captured on an ICS network is shown in Figure 4.4. The most useful information, source IP address and subnet mask, are marked with red.

- In block 10, the subnet that the router belongs to is calculated using the information from OSPF. Converting a subnet from IP/mask format to CIDR is trivial.
- Block 12 checks if the detected subnet is already in the known subnets list. If not, it is saved to memory.
- Next (14), the router's information is saved in the known assets list, along with neighboring routers that may be referred-to in the OSPF hello message. This last piece of information is also extracted from the same hello message, and shown as the last line in Figure 4.4.
- If OSPF was not present, information from the IP layer is saved (15).
- In conditional block 16, it is checked whether the subnet from which the packet comes is still unknown. This would be the case if the packet has no OSPF layer, or if the OSPF subnet extraction has failed (e.g. the check of block 11 failed). If no information exists, the source IP is checked against all known subnets in the database, in an effort to find a match.

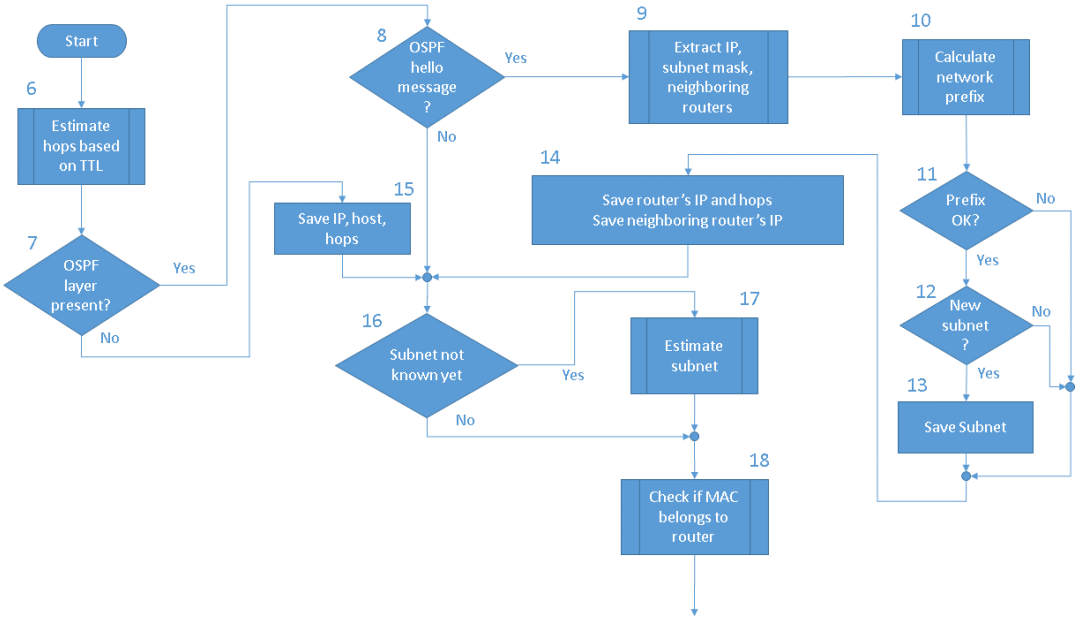



Figure 4.3: Overview of the algorithm that parses the IP layer, part 1.

- Block 18 contains a subprocess that checks the source MAC against the MACs of known routers in memory. If it turns out that the MAC belongs to a router (19 in Figure 4.5), block 20 is chosen.
- Subsequently (20), if the packet originates in a remote subnet (not monitored by the sensor), and that source subnet is known already, an inter-subnet relationship can be deduced, so it is saved to the appropriate list.
- The MAC and IP pair of the current host is saved to the appropriate list if not already known (23, 24).
- If the packet comes from a host in the local (in relation to a sensor) subnet (25) and it is not yet known (26) it is deduced that there is a sensor in it, so it is saved to the list of tapped subnets (27).

```
▶Frame 1: 94 bytes on wire (752 bits), 94 bytes captured (752 bits)
▶Ethernet II, Src: Cisco_                          , Dst: IPv4mcast_00:00:05 (01:00:5e:00:00:05)
▶Internet Protocol Version 4, Src: a.b.c.d                  , Dst:
▼Open Shortest Path First
 ▼OSPF Header
    OSPF Version: 2
    Message Type: Hello Packet (1)
    Packet Length: 48
    Source OSPF Router: a.b.c.d
    Area ID:
    Packet Checksum: 0xb6d0 [correct]
    Auth Type: Null
    Auth Data (none)
 ▼OSPF Hello Packet
    Network Mask: 255.255.255.0
    Hello Interval: 10 seconds
  ▶Options: 0x12 (L, E)
    Router Priority: 1
    Router Dead Interval: 40 seconds
    Designated Router: e.f.g.h
    Backup Designated Router: a.b.c.d
    Active Neighbor: e.f.g.h
```

Figure 4.4: A sample of an OSPF Hello packet, with sensitive data masked. The IP and subnet mask fields that are noted with red color are the pieces of information that enable us to fully define a subnet.

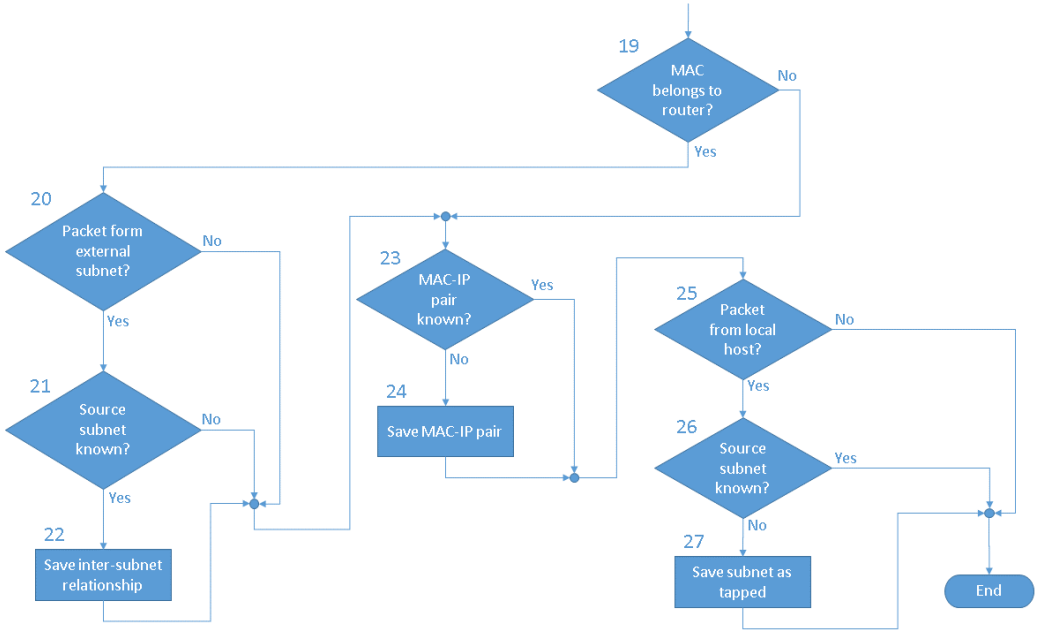



Figure 4.5: Overview of the algorithm that parses the IP layer, part 2.

## 4.3 Implementation & Evaluation

### 4.3.1 The Prototype

The algorithm described earlier has been implemented in a prototype application. As in Chapter 3, *Python*[5] was used, along with *Scapy*[6] to manipulate PCAP network traces. Additionally, *iptools*[7] facilitates the process of converting IP addresses and subnet definitions between notations, such as 192.168.0.0/24 which is equivalent to IP 192.168.0.0 and mask 255.255.255.0. Vendor lookup based on MAC addresses is achieved by issuing HTTP requests to an external service using the *requests*[8] library for Python. To enumerate files and directories and run other system functions, *sys*[9] and *glob*[10] are used.

During the initial PCAP manipulation tests with scapy it became evident that memory management would be challenging. Although performance is generally not a priority when building a prototype, some measures had to be taken to make processing of large datasets feasible. As soon as a PCAP file is read by scapy, it is parsed and loaded into memory but the process can be considered to be inefficient. Early tests showed that the system memory used by scapy when a 2 MB PCAP file was loaded can reach 117 MB, which translates to an effective multiplier of approximately 58.5. Testing with a 22 MB PCAP led to the use of 1186 MB as shown in Figure 4.6.

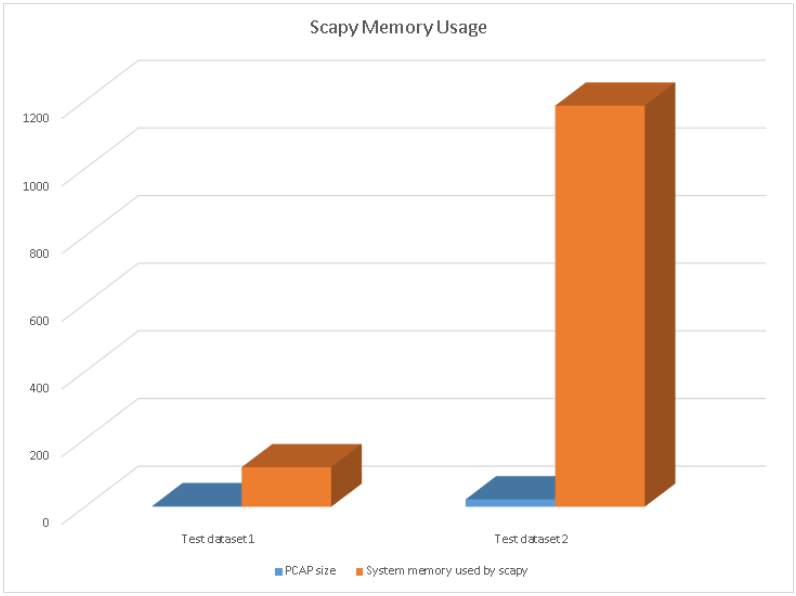



Figure 4.6: Memory used by scapy for two different PCAP files. The system memory used to load a PCAP is over 50 times its size.

In order to make the processing of large PCAPs possible, a script was used to pre-process files before they were loaded to scapy. Using the linux utility *editcap*[11], the script scans directories that contain large PCAPs and dissects them into new files that contain at most 100000 packets. The resulting files can be handled easily by scapy, and the topology discovery prototype can scan a directory and iterate over them using the *glob* python library. This technique effectively renders

[5] https://www.python.org
[6] http://www.secdev.org/projects/scapy
[7] https://pypi.python.org/pypi/iptools
[8] https://pypi.python.org/pypi/requests
[9] https://docs.python.org/2/library/sys.html
[10] https://docs.python.org/2/library/glob.html
[11] https://www.wireshark.org/docs/man-pages/editcap.html

the prototype able to process files of up to a few GB, when run on a system with only 4 GB or system memory (RAM).

The implementation follows the design described in Section 4.2. The final step of the process is to export the results, which may be done periodically or when the algorithm terminates, and contain the following information.

- The subnets that are fully defined,
- the sensors that have been used for tapping,
- all the assets that have sent at least one packet,
- all the MAC addresses of local devices along with their manufacturers,
- the MAC and IP addresses of all devices detected,
- the MAC addresses of local devices that have been associated with IPs,
- the interconnections between subnets that have been discovered.

### 4.3.2 Evaluation

The prototype was initially tested on several datasets that were available in the form of PCAP network trace files. The datasets contained traffic that had previously been captured using network sensors installed at various ICS installations. These industrial control systems belong to different companies operating in different sectors, spanning from gas distribution to electric energy production, and operate in different countries.

For the final evaluation of the prototype, two of those datasets were chosen, based on two criteria: (i) if the OSPF protocol is used in the network, which is integral to the operation of our algorithm; and ii) the availability of (part of the) ground truth, which will enable us to evaluate and discuss the results we obtain. Note that complete knowledge of the network topology is not available for any of the datasets we used, however we have partial information useful to make interesting observations.

Our goal for this task is twofold; first, we seek to evaluate the performance of our algorithm, in regard to its ability to successfully recognize the topology of a network based on the data that it is provided with. Based on the ground truth we have obtained, we will be able to judge how the algorithm's findings compare to reality. Second, we aim to determine what is the minimum observation time needed, for any passive algorithm, to reliable map an ICS network. Most algorithms will give results as soon as they process the first few packets, but later packets may provide information that adds to or changes the perceived topology. We need, thus, to know what amount of time should be chosen, after which the results can be considered solid. We will achieve this by measuring what information is gained over time, while monitoring and ICS network, and we will conclude to an optimal minimum observation interval.

The network time covered by the datasets, which is the time during which the respective networks were being tapped, is shown in Table 4.2.

| Dataset | Duration |
|---|---|
| 1 | approx. 80 hours |
| 2 | approx. 24 days |

Table 4.2: Time durations of the two datasets used.

It is evident that the datasets cover a long time of network traffic, rendering them suitable for a historic development study. Subsequently, we decided to divide the datasets in smaller *subsets*, and run the prototype separately on each one. Each subset maps to a few hours of actual network traffic, but their exact length depends on technical restrictions such as disk space on the test

system. In any case, the subsets of each dataset are guaranteed to be of equal length. This method will enable us to determine how the processing of each consecutive subset influences the end recognition result.

Unfortunately, both datasets were acquired by using one tap or sensor, while the subnet interconnection feature of the topology discovery algorithm requires a minimum of two sensors to be in place, and was thus not tested.

The results of the algorithm runs are presented in two separate sections. First, an aggregated overview is given, with metrics that characterize each dataset as a whole. Then, the analysis of the progression of features over time is given, to discover variables that are sensitive to timing.

**Holistic Analysis**

The metrics that are used to evaluate the overall performance of the prototype are the following:

- The number of IP source addresses of hosts that are present in the dataset.
- The number of assets that have been identified successfully (by IP address).
- The number of MAC source addresses that are present in the dataset.
- The number of MAC addresses of hosts that have been correlated with IPs.
- The multitude of detected network taps or sensors.
- The multitude of network taps or sensors that were used in reality during the packet capturing process. External sources need to be used to obtain this information, because detecting them in the dataset by hand is impractical.
- The subnets that have been fully defined by the algorithm.
- The subnets that we expected the algorithm to be able to define fully.
- The OSPF-enabled and all routers detected.
- The OSPF-enabled routers that are present in the dataset. It must be noted that it is impractical to estimate non-OSPF routers in a dataset manually, so this value is unavailable.

The above metrics are provided for datasets 1 and 2 in Tables 4.3 and 4.4 respectively. For both cases, the recognized IPs total count is actually higher than the one for IPs present in the dataset. Although this may seem like a mistake at first, it is the result of one of the IPs being listed as a neighbor of one of the OSPF-capable hosts, despite the fact that there are no packets originating from it in the dataset. This discovery highlights the advantages of using the neighbor feature of OSPF and shows that it can help increase discovery rates in practice.

| | IPs | MACs | Sensors | Fully defined subnets | OSPF-enabled routers | All routers |
|---|---|---|---|---|---|---|
| Detected | 368 | 6 | 1 | 1 | 5 | 43 |
| Ground truth | 366 | - | 1 | 1 | 4 | - |

Table 4.3: Overall results for dataset 1, when run on the complete dataset.

The OSPF capable routers in dataset 1 are 5, which is more than the expected 4, and is in line with the above finding. The same holds for dataset 2, where 3 routers were found, while 2 were expected.

The total of recognized routers in dataset 1 is 43, while only 6 MAC addresses are detected in the tapped LAN. This discrepancy hints problems with the algorithm, and upon inspection it was found that packets coming from hosts in the tapped subnet appeared to be arriving through a router-like device. In reality, a part of the subnet was behind a firewall, which filtered traffic

| | IPs | MACs | Sensors | Fully defined subnets | OSPF-enabled routers | All routers |
|---|---|---|---|---|---|---|
| Detected | 225 | 24 | 1 | 1 | 3 | 6 |
| Ground truth | 224 | 24 | 1 | 1 | 2 | - |

Table 4.4: Overall results for dataset 2, when run on the complete dataset.

and retransmitted it onto the tapped LAN. This unusual topology led our algorithm to mistakenly deduce that these hosts were routers, while they were simple hosts. The router recognition part of the algorithm can be fitted with detection of this topology, if successful recognition of it is required. The improvement is outlined in Chapter 5.

In dataset 2, 6 routers were found, which seems plausible, but we cannot know for sure if more exist that the algorithm missed. As shown in Tables 4.3 and 4.4, and verified through-out datasets 1 and 2, the detected sensors value was always 1, which is accurate compared to our ground truth.

### Time Progression Analysis

To track how the network and its recognition change fluctuate, we measured the number of IPs, MACs and identified subnets over time for both datasets. Each dataset is split in 12 parts of equal length, called *subsets*. The number of new elements discovered when running the prototype on each consecutive subset of data will illustrate the development of host detection over time. For dataset 1, each subset covers a real time duration of 6 hours and 40 minutes. For dataset 2, the corresponding duration is 1 day, 23 hours and 8 minutes.

**Dataset 1.** Table 4.5 provides the sums of IPs per algorithm run on each subset of dataset 1. The data is presented per subset, each column representing one of the 12 subsets that were processed. The first two rows of data contain the ground truth about IPs that are present in the dataset. The first of those is the *Count* of IPs found in each subset. The second row contains the measure of how many new IPs appeared in each subset. This figure was calculated by comparing the IPs that appear in each dataset with all the previously known addresses.

The last two rows of the table contain the equivalent figures, as detected by our algorithm. We can observe that all the IP addresses present in the dataset are successfully detected by the algorithm, with the addition of 2 extra ones which belong to neighbors of OSPF enabled devices, which were discussed earlier.

| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Ground truth | Count | 317 | 322 | 331 | 321 | 331 | 318 | 323 | 317 | 321 | 319 | 328 | 333 | 366 |
| | New in this subset | 317 | 8 | 14 | 3 | 4 | 0 | 2 | 0 | 0 | 0 | 11 | 7 | - |
| Detected by the algorithm | Count | 319 | 324 | 333 | 323 | 333 | 320 | 325 | 319 | 323 | 321 | 330 | 335 | 368 |
| | New in this subset | 319 | 8 | 14 | 3 | 4 | 0 | 2 | 0 | 0 | 0 | 11 | 7 | - |

Table 4.5: Time-based progression analysis of IP addresses that are present in dataset 1, and how many of them were detected by the algorithm. Each time division (subset) corresponds to almost 7 hours of traffic.

A measure that is not depicted is the detected subnets over time. For dataset 1, the subnet detected by the algorithm matches ground truth in all 12 subsets.

Regarding the development of IPs over time, Figure 4.7 illustrates that the number of new appearances after the first subset, which lasts for approximately 6.5 hours, is very low.

In addition to the above measurements, we tracked how many of the assets that have initially been characterized as hosts have later been changed into routers. This provides us with an indication of how much observation time is needed for the algorithm to decide if a network host is just that or if it "turn into" be a router. The characterizations of hosts as routers include three notable cases:

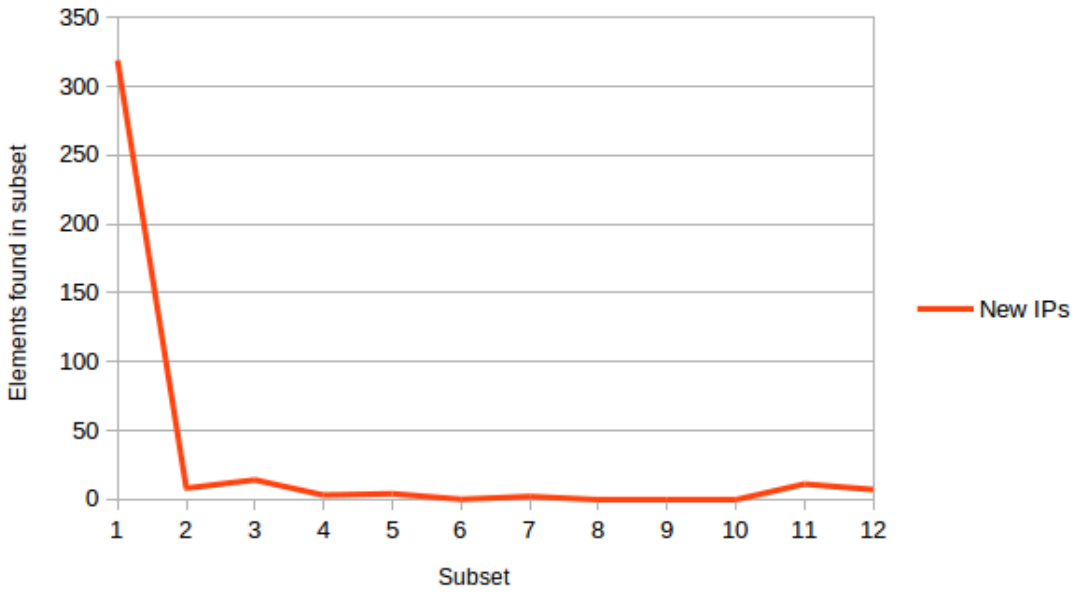



Figure 4.7: The development of detected IP addresses over time in dataset 1, as detected by the algorithm.

- One of the IPs that is first recognized in subset 2, is directly identified as a router.
- An IP identified as a router appears only in subsets 2, 5, 7 and 9.
- Another router's IP is recognized in subset 3.
- There are no IPs that appear to be hosts in some of the subsets and change into routers in others.

The above observations enable us to estimate that within the first 3 to 4 subsets, which translate to approximately 24 hours of operation on a new network, our algorithm has adequately mapped the network accurately enough for practical use. Neither of the detected IPs, routers or sensors exhibit significant fluctuations after the first few subsets.

**Dataset 2.** The second dataset we obtained is much more extensive than the previous. A real ICS network was monitored for approximately 24 days, and we processed the data in 12 runs of the algorithm. Consequently, each subset covers almost 2 days of data. The measurements made for this dataset were more extensive than the previous ones and included MAC addresses. Table 4.6 provides the MAC and IP counts for dataset 2.

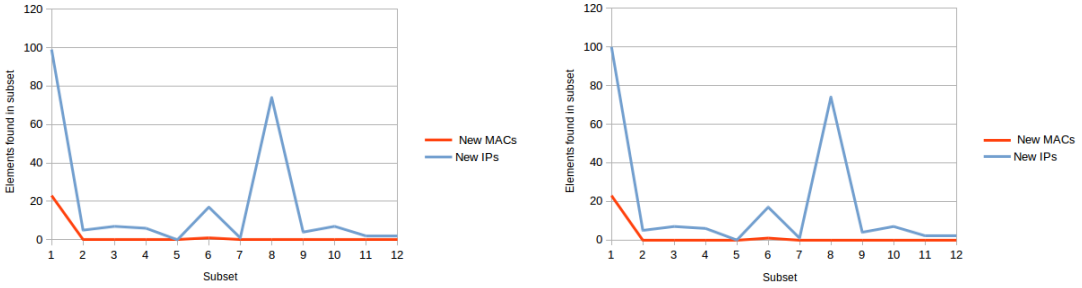
The first four rows of data refer to the MAC addresses, the first two of which present the MACs that exist in the dataset. Similar to the previous dataset, one line provides the *Count* of the metric per subset (which covers 2 days in this case), while the next shows how many new elements were identified in the specific subset. By comparing the rows about ground truth with the rows about our algorithm's detection, we can observe that the total of MACs existing in the dataset is 24 and the algorithm recognized all of them. During subsets 2 through 5, one of the MACs is not recognized although it is active, but it is included in the overall results.

Regarding IPs, the format of the second half of the table is similar to that of MACs. We can note that the detected IPs almost completely match the ones present in the dataset, with the exception of the first subset and the total. Also, the new IPs that appear in each subsequent dataset are relatively low, as depicted in Figure 4.8, with the notable exception of subset 9 which

| | | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MACs | Ground truth | Count | 23 | 23 | 23 | 23 | 23 | 24 | 24 | 24 | 24 | 24 | 24 | 24 | 24 |
| | | New in this subset | 23 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | - |
| | Detected by the algorithm | Count | 23 | 22 | 22 | 22 | 22 | 24 | 24 | 24 | 24 | 24 | 24 | 24 | 24 |
| | | New in this subset | 23 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | - |
| IPs | Ground truth | Count | 99 | 97 | 104 | 111 | 108 | 129 | 113 | 201 | 117 | 130 | 121 | 190 | 224 |
| | | New in this subset | 99 | 5 | 7 | 6 | 0 | 17 | 1 | 74 | 4 | 7 | 2 | 2 | - |
| | Detected by the algorithm | Count | 100 | 97 | 104 | 111 | 108 | 129 | 113 | 201 | 117 | 130 | 121 | 190 | 225 |
| | | New in this subset | 100 | 5 | 7 | 6 | 0 | 17 | 1 | 74 | 4 | 7 | 2 | 2 | - |

Table 4.6: Time-based progression analysis of MAC and IP addresses detected per subset of dataset 2. Each time division (subset) corresponds to almost 2 days of traffic.

contains 74 new IPs. After further investigation, it was found that these IPs are only present in subsets 8 and 12.




(a) New MACs and IPs per subset, ground truth.

(b) New MACs and IPs per subset, as detected by the algorithm.

Figure 4.8: MAC and IP development over time in dataset 2. Each time division covers approximately 2 days of traffic.

Further measurements, that are not depicted in these tables, include the detection of one subnet consistently throughout the dataset, which is correct based on ground truth. Also, of the six routers in dataset 2 (Table 4.4), one was present only in subset 9, while another was marked as a host in subsets 1-3 and as a router only in 4. The detected sensors value was always 1, which was correct.

**Take-aways** Based on the results of both datasets and their analysis, we can make the following concluding remarks. In Section 4.2 we mentioned that the algorithm can be run continuously on new traffic, while outputting its findings for further use by a user or IDS. The information should only be output once it is considered unlikely to change. The following points aim to aid a future developer in deciding which pieces of information can be considered final and after how much observation time.

- Regarding IP addresses, the variance in the timing of discoveries was found to be low in dataset 1 which lasted only 80 hours, while it was high in dataset 2 which covered 24 days. We can conclude that an observation period of 24 days may not be adequate to obtain a full "picture" of the situation in some ICS networks. IPs of new hosts and even routers may appear within or beyond that period. This irregularity is probably caused by links (interconnections) to remote subnets being enabled or disabled sporadically, and can be considered to be a consequence of having large and complex networks.

  The above recommendation though, does not mean that no conclusions about the network can be made before the 24-day interval. Hosts that existed in the early days of monitoring were promptly detected, so the user can be notified within the first few hours of operation

and be given a fairly complete overview of the monitored subnets along with their close neighbors. The 24-day interval only concerns the full image of a network, and is influenced by the particularities and dynamics of specific networks.

- The tapped LAN does not exhibit any significant changes beyond the first subset of observation, which translates to approximately 6 hours and 2 days of network traffic, for the two datasets. We can thus consider a 6 hour interval adequate for local device mapping.
- Our algorithm promptly recognizes new IPs and MACs that appear on the network as soon as they do. We have every reason to believe that there is no delay between the time when a host first starts transmitting on the network and when it is detected by the algorithm.
- According to our measurements most of the routers are recognized as such fast, within the first minutes of operation. A small percentage though (10% of them) have first appeared to be simple hosts and were later detected as routers. Due to the small sample size, only one router exhibited this behavior. To be completely safe, one could state that the detection of the type of a device (whether it's a router or other host), should be given 8 days to settle. This means that once a host is identified, it should be given at least 8 days before it's characterized as a host. If this is not obeyed, there is the possibility of router behavior emerging later. We must note that this only affects non OSPF-capable routers.
- The algorithm's performance on host recognition is outstanding, even detecting systems that have not explicitly communicated on the tapped LAN, by using extra information in OSPF messages.
- Detection of sensors and tapped subnets works reliably, from the first subset, although multiple sensor behavior was not tested.

# Chapter 5

# Conclusions and Future Work

In the beginning of this work we introduced the example of the administrators of an ICS network with 300 devices distributed in 5 locations and the challenges they face trying to maintain situational awareness. Manually discovering and recording which device runs what operating system (OS) version or firmware and how it is connected to the rest of the network is labor intensive and can be dangerous. Active device query must be avoided so the operators are in need of tools or systems that discover and characterize network devices in a passive way.

Existing OS fingerprinting tools do not suit these needs, so in Chapter 3 we introduced a novel approach that utilizes machine learning to offer high performance passive OS fingerprinting. When compared to the latest tools available, our implementation was shown to be a solid performer, detecting almost as many hosts as PRADS (which came first in that area) and predicting OSes with accuracy equivalent to p0f (which used to be the best in that aspect). Additionally, our tool's predictions were given with a much higher confidence than any other tool, with an average value of over 97.5% for the two examined datasets.

In the topology discovery front, we showed that a mix of basic and specialized protocols can be used to achieve results in a passive manner (Chapter 4). We introduced an algorithm that extracts the most relevant features from the various layers and protocols and draws conclusions about the detected hosts. An implementation of the algorithm was used to test the logic, and was shown to have excellent performance, even recognizing some hosts that did not explicitly send data on the wire. Furthermore, the explored datasets were also used to identify how often new devices appear on ICS networks, a value that determines how soon a topology discovery tool can produce reliable information after being run on a network.

While the proposed solutions performed well, their scope of use was purposely limited. The topology discovery algorithm currently works most efficiently on networks where the *open shortest path first* (OSPF) routing protocol is used, which despite probably being the most widely used routing protocol inside autonomous systems, can not be considered ubiquitous. Also, the requirements for no internal *network address translation* (NAT) or firewall devices within the autonomous system or subnet are in place, which are reasonable to make for this first iteration. Further work can be based on our topology discovery findings and extend the algorithm by adding more supported routing protocols or adding support for NAT and mid-subnet firewalls. More extensive parsing of protocols such as SMB can provide further insight into ICS networks.

With regard to OS fingerprinting, numerous options exist towards further development and improvement of the technique. One direction would be to try different predictive models, such as random forests which are based on decision trees but offer advantages over them, towards increasing performance. A different approach would be to experiment with different TCP header fields that compose TCP fingerprints. The choice of the options definitely has an effect on the end result, but the optimum set of options needs to be determined in a robust way. A more detailed analysis of possibilities for future development are given in the next few paragraphs. Nevertheless, given the performance increase introduced by our tool, we can be confident that passively monitoring large ICS networks and detecting devices and their characteristics can now be done in a more effective

and efficient manner.

**OS Fingerprinting: Future Work.** The tests of our implementation showed that applying machine learning is an effective technique to improve OS fingerprinting in challenging environments such as completely passive application. The important implication of this is that a new road is opened towards further performance improvements. We believe that by tuning two of the parameters of the algorithm, further performance gains will follow. One such area is the predictive model used; while we used decision trees based on research that has shown them to be effective in our application [22], other methods such as random forests have not been tested, and due to the relationship of the two techniques it is probable that a forest's predictive performance will be better. The same holds for the TCP header fields that are used to construct fingerprints. Lippmann et al. have concluded, in the same paper, on what the optimum set of features is, but the research is now over 12 years old, and new systems, TCP implementations and predictive models may be affecting the effect of the chosen features. Thus, we propose that different combinations of models and fields are evaluated using real network data to conclude whether accuracy, detection rate and confidence of OS predictions can be improved further.

Other areas that can be extended include SMB parsing and performance. The former refers to the messages of type *Microsoft Windows Lanman Remote API*, which were deliberately not used to avoid a possible pitfall, as explained in Section 3.2. If a sufficiently extensive network is available, SMB signatures can be obtained comprising of the OS minor and major fields, along with the *browser protocol* major and minor versions. These four elements can form signatures that will identify OSes with very high accuracy and enable the use of *Microsoft Windows Lanman Remote API Protocol*.

Finally, if system performance issues arise on the sensor, future solutions can utilize a trick that has been introduced in Satori to reduce system load. A pre-processing stage is added, so that the network stream is filtered to exclude packets known to not give any useful information to the OS fingerprinter, which is in turn fed only the few select packets that it needs. For TCP fingerprinting, such filtering would simply drop packets that do not have the SYN flag set.

**Topology Discovery: Future Work.** As shown in Section 4.3 the implementation of our algorithm operates well and provides an effective step into discovering the topology of previously unmapped networks. Nonetheless, there are changes and additions that may improve its performance further and an overview of them is presented below.

- The tool can be run continuously or periodically on live traffic, given that it is modified to export some of the results appropriately. This export can be into a database and should include most of the information that is internally available to the algorithm, with the exception of elements that may change or are only destined for internal use. Examples of temporary information include the list that includes all the relationships ever observed between MAC and IP addresses, because only the entries of local hosts are of interest to the end user.

- Operating system detection may be incorporated to the information kept about each asset, as there are already provisions for it and it can add a lot of value to an asset inventory. Interoperability between the OS fingerprinting tool can be build to achieve this.

- OS detection can also be used in conjunction with topology discovery to achieve NAT detection. The way this works is that if multiple OSes are detected for an IP address, the IP may be a router that utilizes NAT and serves numerous hosts "behind" it. This network topology was explicitly not supported in the design of the current algorithm because it is not common in ICS networks, but adding the capability of detecting it may add to the robustness of the algorithm.

- Processing ICMP TTL_exceeded messages that may be sent from routers in situations like traceroute execution can provide insights of the network that are normally available only when active techniques are employed.

- Additional specialized protocols can be parsed to enable the algorithm to operate in more datasets, where OSPF may not be in use. According to previous measurements (presented in Section 4.1.2), parsing CDP would be the next logical step to gain better coverage with minimal effort. Protocol fields such as *device capabilities* can allow the flagging of a device as a managed switch, router, wireless access point or other types. Further information may include all the subnetworks (including netmasks) known to the device, detailed system description and other.
- One of the challenges that has not been addressed in this work but is important to tackle is the ability to correlate multiple network interfaces to one host. This situation arises when two tapped LANs are connected to each other through a router. That router's two interfaces have different MAC and IP addresses on each LAN. Packets coming from or through the router may be captured in both LANs, which will cause the algorithm to generate an entry for each MAC-IP pair. It is non-trivial but also not infeasible to identify which of the MAC-IP pairs belong to two different interfaces of the same physical host.
- *ICMP router solicitation messages* (type 10) contain information such as router IP addresses, which may be useful if parsed.
- As explained in Section 2.2, information parsed off DHCP packets is shown to be extremely accurate and detailed. Although DHCP is not widely used in ICS networks, parsing it may help obtain the network mask and other attributes faster in some networks.
- Long duration measurements, such as the one done on dataset 2 in Chapter 4, should be repeated on different datasets, coming from a variety of diverse ICS networks. The results of such a study will determine if the sudden appearance of tens of IPs is a regular phenomenon in these types of networks.
- Based on the algorithm's findings when run on dataset 1, we can propose two future improvements that will enable an implementation to cope with firewalls positioned inside subnets. Specifically, the router recognition needs to be augmented, so that it is not confused by firewalls that mangle traffic and a firewall recognition module should be added. The necessary changes should be in the context of a rule such as "when packets from the tapped subnet arrive, and many of the source IPs seem to be positioned behind the same MAC, declare that MAC to be a firewall in the middle of the subnet, and inhibit router detection for those IPs".

# Bibliography


[1] Humberto J Abdelnur, Olivier Festor, et al. Advanced Network Fingerprinting. In *Recent Advances in Intrusion Detection*, pages 372–389. Springer, 2008. 6

[2] Taher Al-Shehari and Farrukh Shahzad. Improving Operating System Fingerprinting using Machine Learning Techniques. *International Journal of Computer Theory and Engineering*, 6(1):57, 2014. 3, 6

[3] Ofir Arkin and Fyodor Yarochkin. Xprobe v2. 0: A fuzzy approach to remote active operating system fingerprinting, 2002. 5

[4] Patrice Auffret. SinFP, unification of active and passive operating system fingerprinting. *Journal in computer virology*, 6(3):197–205, 2010. 5

[5] Sergei Bantseev and Isabelle Labbé. Study of tools for network discovery and network mapping. Technical report, DTIC Document, 2003. 9

[6] Jason Barnes and Patrick Crowley. k-p0f: A high-throughput kernel passive OS fingerprinter. In *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on*, pages 113–114. IEEE, 2013. 5

[7] Robert Beverly. A Robust Classifier for Passive TCP/IP Fingerprinting. In *Proceedings of the 5th Passive and Active Measurement (PAM) Workshop*, pages 158–167, April 2004. 6

[8] Marco Caselli, Frank Kargl, and Valentin Tudor. D4. 4 Device fingerprinting. 7

[9] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. Traffic classification through simple statistical fingerprinting. *ACM SIGCOMM Computer Communication Review*, 37(1):5–16, 2007. 7

[10] Bürak Dayıoğlu and Attila Özgit. Use of passive network mapping to enhance signature quality of misuse network intrusion detection systems. In *16th International Symposium on Computer and Information Sciences*. Citeseer, 2001. 5

[11] Ryan M Gerdes, Thomas E Daniels, Mani Mina, and Steve Russell. Device Identification via Analog Signal Fingerprinting: A Matched Filter Approach. In *NDSS*, 2006. 7

[12] Mehmet H Gunes and Kamil Sarac. Analytical ip alias resolution. In *Communications, 2006. ICC'06. IEEE International Conference on*, volume 1, pages 459–464. IEEE, 2006. 8

[13] Mehmet H Gunes and Kamil Sarac. Inferring subnets in router-level topology collection studies. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 203–208. ACM, 2007. 8

[14] Mehmet H Gunes and Kamil Sarac. Resolving ip aliases in building traceroute-based internet maps. *IEEE/ACM Transactions on Networking (ToN)*, 17(6):1738–1751, 2009. 8

[15] Patrick Haffner, Subhabrata Sen, Oliver Spatscheck, and Dongmei Wang. ACAS: automated construction of application signatures. In *Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*, pages 197–202. ACM, 2005. 7

[16] J Hawkinson and T Bates. Ietf rfc 1930, guidelines for creation of an as, 1994. 10

[17] Darrell M Kienzle, Nathan S Evans, and Matthew C Elder. Nice: endpoint-based topology discovery. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, pages 97–100. ACM, 2014. 9

[18] Tadayoshi Kohno, Andre Broido, and Kimberly C Claffy. Remote physical device fingerprinting. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):93–108, 2005. 7

[19] Eric Kollmann. Chatter on the Wire: A look at excessive network traffic and what it can mean to network security. http://chatteronthewire.org/download/OS%20Fingerprint.pdf, 2005. 4, 15, 16, 17

[20] Eric Kollmann. Chatter on the Wire: A look at DHCP traffic. http://chatteronthewire.org/download/chatter-dhcpv6.pdf, 2007. 4

[21] Eireann P Leverett. Quantitatively assessing and visualising industrial system attack surfaces. *University of Cambridge, Darwin College*, 2011. 7

[22] Richard Lippmann, David Fried, Keith Piwowarski, and William Streilein. Passive operating system identification from TCP/IP packet headers. In *Workshop on Data Mining for Computer Security*, page 40. Citeseer, 2003. 5, 6, 15, 40

[23] Gordon Fyodor Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, 2009. 5

[24] Takashi Matsunaka, Akimasa Yamada, and Ayumu Kubota. Passive OS Fingerprinting by DNS Traffic Analysis. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 243–250. IEEE, 2013. 4

[25] Pascal Mérindol, Benoit Donnet, Olivier Bonaventure, and Jean-Jacques Pansiot. On the impact of layer-2 on node degree distribution. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 179–191. ACM, 2010. 8

[26] Pascal Mérindol, Virginie Van den Schrieck, Benoit Donnet, Olivier Bonaventure, and Jean-Jacques Pansiot. Quantifying ases multiconnectivity using multicast information. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 370–376. ACM, 2009. 8

[27] D MONTIGNY, A LEBOEUF, and F MASSICOTTE. Passive network discovery for real time situation awareness. In *Proc of NATO/RTO Symposium on Adaptive Defence in Unclassified Networks*, 2004. 4

[28] Andrew W Moore and Konstantina Papagiannaki. Toward the accurate identification of network applications. In *Passive and Active Network Measurement*, pages 41–54. Springer, 2005. 7

[29] John Moy. Rfc 2328: Ospf version 2, 1998. 26

[30] Duane Norton. An ettercap primer. *SANS Institute InfoSec Reading Room*, 2004. 5

[31] Vern Paxson. Automated packet trace analysis of TCP implementations. *ACM SIGCOMM Computer Communication Review*, 27(4):167–179, 1997. 3, 5, 11

[32] Jon Postel. Rfc793: Transmission control protocol. usc. *Information Sciences Institute*, 27:123–150, 1981. 12

[33] ITUTX Recommendation. 200 (1994)— iso/iec 7498-1: 1994. *Information technology–Open systems interconnection–Basic reference model: The basic model*, 1994. 7

[34] Rose K McCloghrie K RFC1213 and M Rose McCloghrie. Management information base for network management of tcp. *IP-based internets: MIBII*, 1991. 9

[35] Gordon Rueff, Corey Thuen, and James Davidson. Sophia proof of concept report. March 2010. 7

[36] Hee So. SANS Institute - Intrusion Detection FAQ: How can passive techniques be used to audit and discover network vulnerability? https://www.sans.org/security-resources/idfaq/passive_vuln.php. 9, 10

[37] Lance Spitzner. Passive fingerprinting. *FOCUS on Intrusion Detection: Passive Fingerprinting (May 3, 2000)*, pages 1–4, 2000. 5

[38] Greg Taleck. Ambiguity resolution via passive OS fingerprinting. In *Recent Advances in Intrusion Detection*, pages 192–206. Springer, 2003. 5

[39] M Tozal and Kamil Sarac. Tracenet: an internet topology data collector. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 356–368. ACM, 2010. 8

[40] M Tozal and Kamil Sarac. Subnet level network topology mapping. In *Performance Computing and Communications Conference (IPCCC), 2011 IEEE 30th International*, pages 1–8. IEEE, 2011. 8

[41] Franck Veysset, Olivier Courtay, Olivier Heen, IR Team, et al. New tool and technique for remote operating system fingerprinting. *Intranode Software Technologies*, 2002. 4

[42] Fedor V Yarochkin, Ofir Arkin, Meder Kydyraliev, Shih-Yao Dai, Yennun Huang, and Sy-Yen Kuo. Xprobe2++: Low volume remote network information gathering tool. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 205–210. IEEE, 2009. 5

[43] Yun-Sheng Yen, Tung-Lung Chan, Chia-Yi Liu, and Chwan-Yi Shiah. Topology discovery service in the universal network. In *Computer Research and Development (ICCRD), 2011 3rd International Conference on*, volume 1, pages 319–323. IEEE, 2011. 9