UNIVERSITY OF OSLO
Department of Informatics

Investigating Passive Operating System Detection

Petter Bjerke Falch

Network and System Administration

Oslo University College

May 24, 2011

Investigating Passive Operating System Detection

Petter Bjerke Falch

Network and System Administration
Oslo University College

May 24, 2011

Abstract

Today computer and network security are a big part of a system administrators life. New methods and applications appear and several makes the system administrator's job easier.

Gathering information is a big part of detecting threats and staying one step ahead of the black hats.

This thesis looks at and investigates a specific area in network security, namely passive operating system detection. Information is important in network security and knowing your enemies are important in securing your network. Passive operating system detection helps collecting information passively, which can be used to the administrators advantage.

The thesis looks at passive operating system detection applications and looks especially on the applications p0f and prads.

By running both applications in a larger network and testing them in a controlled environment, the weaknesses of both applications are revealed and improvements suggested and tried implemented.

Improvements discussed and tried implemented in this thesis, are adding new signatures and creating Perl scripts that improves the applications itself. The scripts deals with the output from the applications which tends to be overwhelming and needs new presentation methods.

Acknowledgements

I would like to express my gratitude to the following people:

- My supervisor Hårek Haugerud for helping me and guiding me through a hectic master thesis.
- Kyrre Begnum for useful thesis seminars throughout this semester
- The other teachers at Oslo university College for a friendly attitude.
- My classmates for good discussions and nice lunches
- Edward Fjellskål for introducing me to the topic and answering my questions concerning prads
- Jonas Taftø Rødfoss for lending me his network setup
- My patient and lovely wife and my soon to be born child, for waiting a little bit longer

Contents

Chapter 1

Introduction

1.1 Motivation

The big increase in computing worldwide and the ”Internet revolution” demands better and better security in computing and networking in today’s complex society. Today ”everyone” are connected to the Internet and this is why the security is so important. What would happen if the stock market or banks were hacked and taken down? Or what would happen if a regular business depending on Internet for selling their goods went down for only a day? Downtime today could be catastrophic to any company or business, therefore computer security and network security are incredible important to anyone with their computers connected to the ”outside”.

Today worldwide network security is used to prevent and monitor unauthorised access, misuse, modification, or denial of the computer network and network-accessible resources. There are different ways of monitoring networks, like intrusion detection systems that monitors network threats from the outside and network monitoring systems that monitors the internal systems problems caused by overloaded or crashed servers and network connections.

Most hackers and script kiddies are trying to get through the security measures that protects the computer assets from the outside. Security measures protecting the network and computers from the outside are firewalls, network intrusion detection and protection systems, antivirus and authentication and identification systems. Even if we got all these systems, they won’t protect us from employees or students doing something stupid like connecting an insecure wireless network or an unpatched computer leaving the machine totally open.

The article ”Top 10 vulnerabilities inside the network”[1] looks at a computer network different than usual. Inside a computer network we encounter threats that are not that obvious to the everyman. Threats like unidentified thumb drives, miscellaneous USB devices, social engineering, optical media and e-mail are things that should be considered as inside security threats. Also unknown laptops, wireless access points and smart phones constitute a major threat when connected in the network inside a company or school. Security policy’s and common sense aren’t enough to protect the network and the net-

work administrators need tools that can be used to monitor and protect their network.

The ”network security race” consists of many applications trying to keep networks and computer secure. Most companies and schools use several security applications but can't keep up with the hackers. It's a constant war. In addition to firewalls, intrusion detection and prevention systems. Applications like operating systems detection play an important role in the security realm. This thesis will look deeper into operating system detection more accurate, passive operating system detection.

1.2 Problem statement

The questions and formulations asked in the problem statement will be worked on throughout the thesis, and the conclusion will be based on what the problem statement says. Since this is an investigative thesis, all the problem statements are related to what passive operating system detection is and does. The focus will be on existing passive operating system detection applications[2].

The problem statements are:

Investigate Passive Operating System Detection

- *Investigate which passive operating system detection systems that exist today.*
- *Compare p0f and prads in the 128.39.73.0 network.*
- *Discover the weaknesses in existing fingerprinting methods.*
- *Investigate which improvements that can be done to make operating system detection a helpful tool for network administrators.*

Chapter 2

Background and literature

This chapter will introduce the reader to basic concepts of network security[3]and operating system detection/fingerprinting. The focus will be on giving the reader basic knowledge, to be able to keep up in the rest of the thesis, and to look at previous work done in this area of network security.

2.1 History of Network Security and Operating System detection

Network security has existed a long time, and even if network security had existed for a while before the Internet, the Internet gave network security a real boost. When the Internet spread a cross the world, administrators were forced to take security measures to protect their assets, networks and computers against attacks from the "outside". One of the most famous hackers, Kevin Mitnick [4]has contributed to network security.[5] By showing how easy it was to steal intellectual property from many worldwide companies, he showed that there exists a threat and that we need network and computer security.

There have been some important phases in computer and network security. In the paper Network Security: History, Importance, and Future written by Bhavya Daya [6] we see a timeline that spans all the way from about 1930 to todays "Internet age".

The first Polish encryptor's created the famous enigma encryption machine that was hacked or decrypted by the mathematician Alan Turing.

In the 1960s, ARPANet[7], the predecessor to todays Internet, where made by the Department of Defence. Arpanet was a conduit for electronic exchange of information.

In the 1970s the telnet protocol was made. This protocol made data networks open for the public and also invited hackers to participate in the computer networks.

During the 1980s hacking became more and more common. Hackers like the 414 gang and Ian Murphy, that stole information from military computers, enforced the Computer Fraud and Abuse Act in 1986. At about the same time the student Robert Morris was convicted to have made and unleashed the Morris

Worm, that again made the authorities to establish the CERT or the Computer Emergency Response Team. CERT's main goal was to alert the crowd of network security issues.
In the 1990s, the security threats increased drastically. The Internet became public and more and more people were connected. Today you belong to the exception if you don't have Internet access, and the security industry is bigger than ever.
Operating system detection or operating system fingerprinting have not been around for a long time. Nmap[8]that is a famous tool for doing active fingerprinting didn't introduce fingerprinting before December 12, 1998. The first passive operating system detection tools where not developed before early 2000.

2.2 Operating system detection / fingerprinting

In computer security, operating system detection is a way to detect what is running on your network. Not only can fingerprinting be used to detect which operating systems a computer has, but also detect what kind of network components there are in the network. This way a system administrator can get an overview over his own network.

Operating system detection can be looked at in two ways, either from the hacker's point of view, or the network or system administrators point of view. At first sight, it is easy to see what a hacker could do using fingerprinting. An example of use, is to detect vulnerabilities in the sense of an old operating system. Which are easy to exploit for an experienced hacker. When it comes to the network or system administrator, it's maybe not that implicit how this "tool" can be used. In computer security and for most administrators, it is important to collect as much knowledge about their network as possible. For example it could be wise to know which network components and computers you have in the network. It could be that one of the employees has hooked up a dodgy wireless router with no security, to your companies network, which again means that your network is exposed for attacks from the outside. It could also be that one of your machines runs an old version of an operating system, that could be easy to exploit. The bigger the networks are, the more difficult and important it is to have full control of what is connected to the network. For the system administrator, it's always important to be one step ahead of the attacker. This way, the attacker can't make use of the latest vulnerabilities.

There are two main ways of doing operating system detection in a network, active and passive.

2.2.1 Active operating system detection

The active version of doing operating system detection[9] rely on stimulus-response. Meaning that the source will send a special packet, that we can call stimulus, to the target, and get a response that the source can analyse to

identify the operating system of the target.[10] There is actually a legal matter when it comes to active fingerprinting, meaning that in many countries it is illegal to perform an active fingerprint against others. This is one of the reasons why the passive fingerprinting method is developed. Active and passive fingerprinting have a lot in common when it comes to the detection of the operating system. They both use signatures that they match against to discover the operating system.

2.2.2 Passive operating system detection

Contrary to the active way of doing operating system detection, the passive [9]way is about sniffing packets instead of making a special packet and sending it to a remote host. Most passive fingerprinting tools looks at the headers of the TCP SYN packet, and from there determines which operating system we are dealing with. When a SYN packet is sniffed, it can be compared with a database, that contains signatures from different operating systems, and then determine what operating system this packet comes from. Looking at the signatures, they often consist of the TCP SYN packets headers. Different operating systems have different headers, and therefore we can determine the operating system based on the signatures.

For many network administrators today, it's important to have a passive way of doing fingerprinting. Many IDS deployments consist of a computer / devices with two network interfaces. One contains a promiscuous link to a remote network and the other link is for management and lives in the DMZ. The problem then with active scanners is that the IDS team do only have a one- way link into the network. Meaning that they only can listen to traffic, not being able to produce any traffic.

The figure 2.1 is a typical packet, captured in snort, with fields that can be analysed to do passive operating system detection. We see the Time To Live, type of service, window size and so on. Looking at these fields, an operating system detection application can determine which operating system it is.

```
04/20-21:41:48.129662 129.142.224.3:659 -> 172.16.1.107:604
TCP TTL:45 TOS:0x0 ID:56257
***F**A* Seq: 0x9DD90553   Ack: 0xE3C65D7   Win: 0x7D78
```

Figure 2.1: TCP header

For the detection, important fields in the header are TTL, window size, don't fragment bit(DF) and type of service(TOS). Only looking at one of the fields don't give which operating system it is, but when adding these together, detecting the operating system can be done. The more fields set, the more accurate is the detection.

The big advantage with passive fingerprinting is that you can learn about the enemy without them knowing it. Though not 100% accurate, you can determine the assets of a remote host without leaving any traces. [11]

Below is the format of how a signature could look like. This is taken from the signature file p0f.fp in p0f, one of the more known passive operating system detection systems. p0f.fp contains the SYN signatures used in p0f. We see the format of the entry and what the different entries mean. Together, all these fields can determine an operating system and the more fields known, the more precise is the determination of the operating system.

```
Fingerprint entry format:

wwww:ttt:D:ss:OOO...:QQ:OS:Details

wwww - Window size
ttt  - Initial TTL
D    - don't fragment bit
ss   - overall SYN packet size
OOO  - option value and order specification
QQ   - quirks list
OS   - OS genre (Linux, Solaris, Windows)
details - OS description
```

2.2.3 Passive fingerprinting methods

There is not only one way of doing passive fingerprinting. Several methods exist, and different applications use different methods. Typical methods are:

- TCP SYN
- TCP SYNACK
- TCP RST
- TCP FIN
- Telnet data
- ICMP echo request

Looking at the TCP/IP header in figure 2.2, the TCP packet has several flags that can be set. These flags indicate the connection status of a TCP connection. TCP is a stateful protocol that means that it is imperative that one knows the purpose of each and every packet.

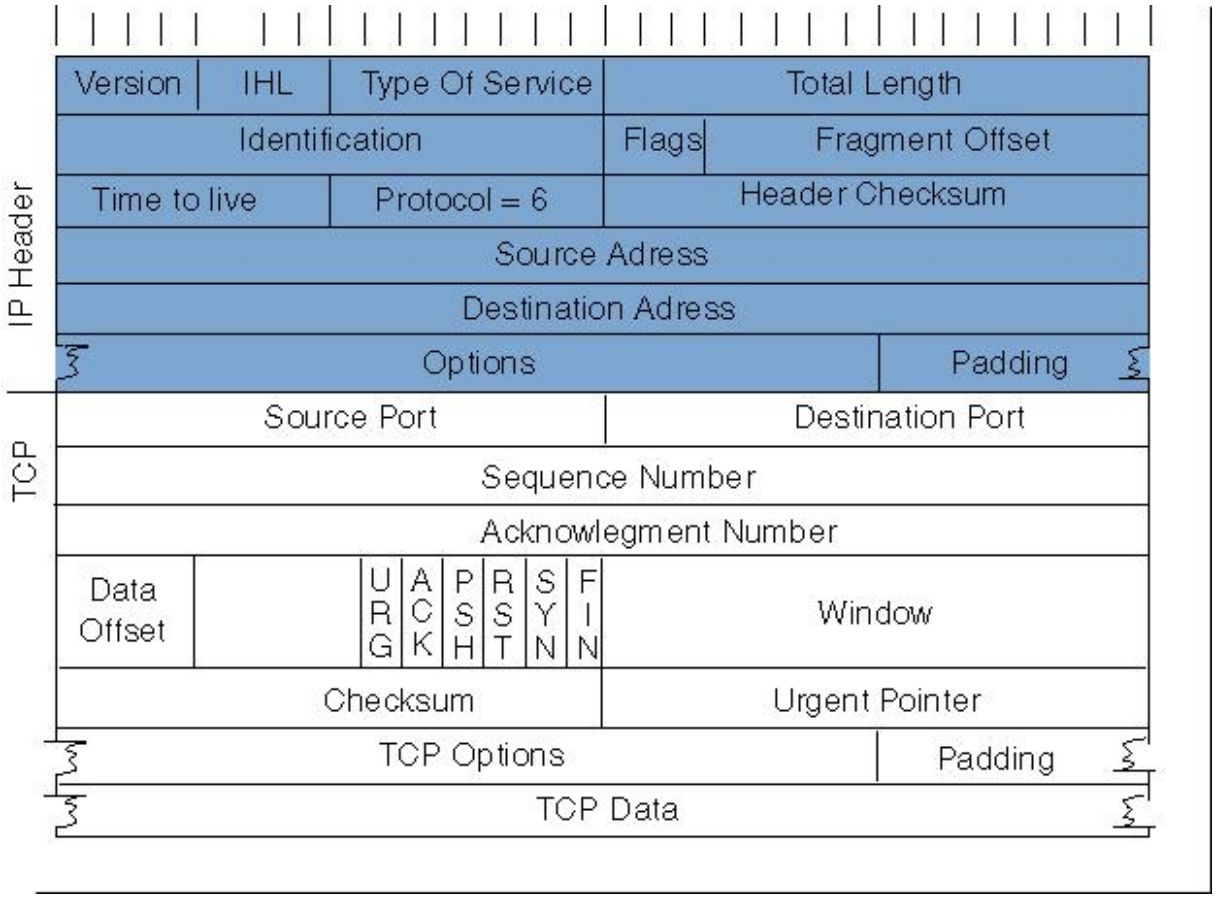

Figure 2.2: The TCP header [Wikimedia commons]

An initiation of a TCP connection always starts with the 3-way handshake. First the initiator sends a request, that is a TCP packet with the SYN flag set, to for example to a web server. Then the server either reply with a SYNACK, if the server is ready to open a connection, or the server could send a RST that tells the initiator that the server is not open for connection. Then as the last packet in the 3-way handshake the initiator replays with the ACK packet, and the connection is established. The reason for looking at the 3-way handshake is that passive fingerprinting takes use of the different states of the TCP connection. As mentioned above, important states are the SYN, SYNACK, RST and FIN. When a packet has one of these flags set, often a passive detection of the operating system can be done. The nice way of doing operating system detection this way is that the sniffer doesn't need to collect many packets. Often the first TCP SYN packet is enough to determine the operating system

2.2.4 Fingerprinting applications

When it comes to applications that are developed to do fingerprinting, there are a few that are worth mentioning, most are open source and under development. Beneath several are mentioned, but not explained in detail. Only three, p0f, PADS and prads will be look at in detail. Figure 2.3 shows an overview of the different applications discussed in this thesis.

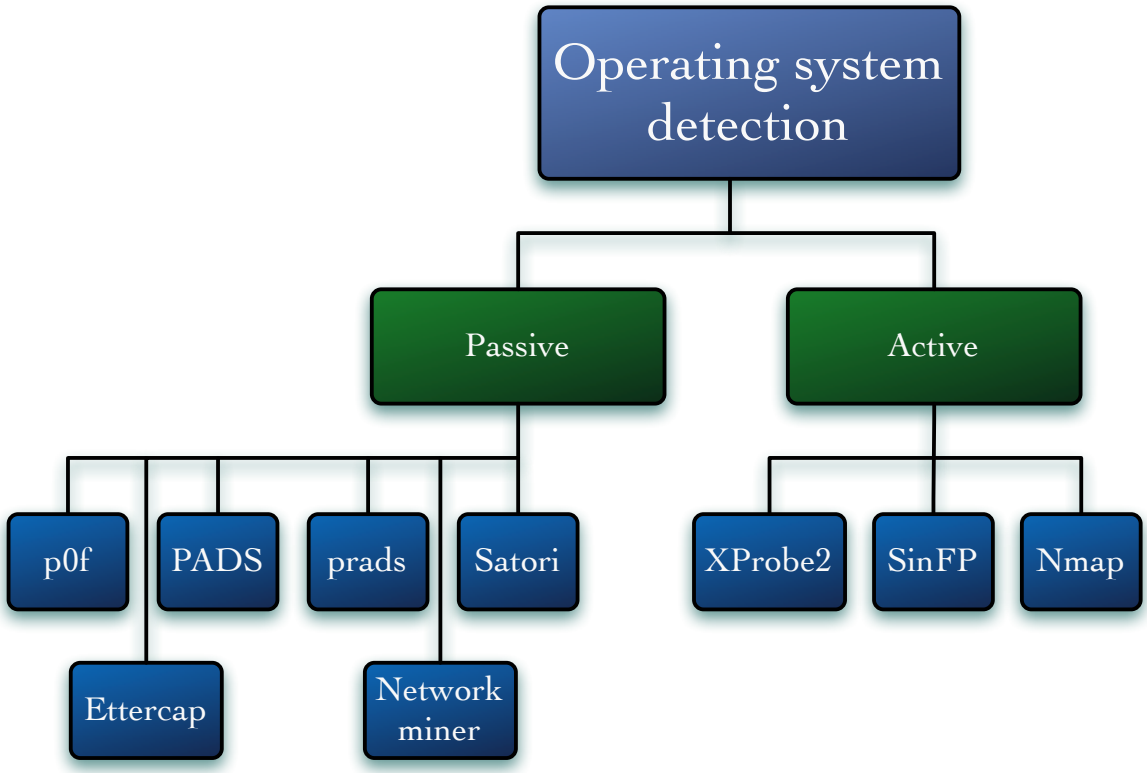

Figure 2.3: A figure of different fingerprinting applications

p0f

p0f[12]is a tool designed to only do passive operating system fingerprinting. p0f can run in different modes, SYN, SYNACK and RST mode. It is said that p0f can detect firewalls, the existence of a load-balancer, the distance to the remote system and it's uptime and others network connection and ISP. p0f is now in version 2 and has been under development for some years. It looks like p0fv2 is the most used passive operating system detection tool, reading at forums and webpages. It gets honour for it's simplicity and it's ease of use. The new p0f is easy to install with aptitude or yum, and requires few additional packages. The most important package it demands is the libcap library.

p0f looks at the signature of a TCP packet and compares this to the signature file that contains signatures for different operating systems. There is one file for each mode p0f runs. One for SYN, one for RST or reset and the last one is for the SYNACK mode. In the signature file, the signatures are given one line each. As an example, we can see the signature for a windows 98 operating system.

```
S44:32:1:48:M*,N,N,S:.:Windows:98 (low TTL) (1)
```

The p0f signature has the following format:

```
wwww:ttt:D:ss:OOO...:QQ:OS:Details
```

And the different fields have the following meaning:

```
wwww:ttt:D:ss:OOO...:QQ:OS:Details

wwww - Window size
ttt  - Initial TTL
D    - don't fragment bit
ss   - overall SYN packet size
OOO  - option value and order specification
QQ   - quirks list
OS   - OS genre (Linux, Solaris, Windows)
details - OS description
```

Window size Is the size of the receive window, that specifics the number of bytes that the receiver is currently willing to receive. The limit of the window size[13] can be between 2 and 65,535 bytes. In high-bandwidth networks, increasing the window size can help the efficiency of the network.

TTL The time to live[13] is a limit of how long a network or computer network technologic item can transmit before it should be stopped. Typical, is for how long a packet should be able to retransmitted before it should be discarded.

Don't fragment bit If the don't fragment bit[13] is set on a packet the packet won't be fragmented under any circumstances. This means that if the packet can't be delivered without fragmenting it, the packet would be discarded instead .

SYN packet size Is a field where the size of the SYN packet i set.

Options Options provide some extra control functions that could be useful, but not necessary for the most of time. The different options are end of list, no operation, windows scaling, maximum segment size, selective ACK OK, timestamp, timestamp with zero value and unrecognised option number.

Quirks Some systems could have bugs that set certain values that should be zeroed in a TCP packet ton non-zero values. This information could be valuable since it could be specific for one operation system.

OS genre Is the main name of the operating system. This is what the previous fields lead up to.

Details Is a more detailed explanation of the operating system. It often states which kernel, or version number the operating is.

Since there always comes new operating systems there will always be a need of updating the signature databases. The developers of p0f have made a solution to this by making a web page where people using p0f can add new signatures. Even though this web page exists, it looks like the signatures need a major upgrade to meet today's new operating systems.

PADS

PADS[14] or Passive asset detection system is a tool to help the system administrator to keep track of the networks assets. Pads do not include operating system detection, only asset detection. Similar to a passive operating system detection tool, PADS uses signatures to detect the networks assets. PADS complement IDS technology by providing context to IDS alerts. In PADS manual pages, it says that PADS has the aim of being passive, lightweight and portable. Passive in the meaning that it will record traffic, but never send any packets out from the application. Lightweight in the sense that there is no need for any database or any repository installed on the local machine. Portable, meaning that it can be placed on remote system without requiring any additional libraries expect those associated with libpcap.

Some features that PADS possess is the possibility to:

- Retrieval of MAC addresses via ARP reply packets
- ICMP support, via ICMP echo replay packets
- A network or several networks can be specified, meaning that it will only record assets from those networks.
- SLL Frame Relay Support
- The ability to work in the background as a daemon

- Tools for archiving PADS data to permanent storage and making structured reports out of PADS data.

Running asset detection gives the following output.

```
root@gateway:/etc/pads# cat assets.csv
asset,port,proto,service,application,discovered
128.39.73.249,22,6,ssh,OpenSSH 4.7p1 (Protocol 2.0),1304678612
```

First is the ip address, then comes the port number on which the service is running, then the protocol, service type and which applications. The last number is a timestamp of when the service was detected.

PADS has several signature files that is used when doing detection. One is the pads-ether-codes that is a list of mac addresses and vendors of computer equipment. This file contains vendor codes used to map MAC addresses to vendor names.

```
00:03:93  Apple Computer, Inc.
00:05:02  Apple Computer
00:0A:27  Apple Computer, Inc.
00:0A:95  Apple Computer, Inc.
00:30:65  Apple Airport Card 2002
00:50:E4  Apple Computer, Inc.
00:A0:40  Apple Computer
08:00:07  Apple Computer Inc.
```

The other, and more important signature file, is the pads-signature-list. This file contains a database of device signatures to be used with the passive asset detection system.

This file has the following format:

```
<service>,<version info>,<signature>
```

Service Describes the service name used by the signature. It could be SSH, SMTP, FTP, HTTP, etc.

Version info Contains an NMAP-like template for the service discovered by the signature. The format of the version info is as follows:

```
v/vendorproductname/version/info/
```

Signature The signature is a PCRE compatible regular expression without the surrounding slashes (/). PCRE means that it is using the same syntax and semantics as Perl 5. The signature should have one or two sets of parenthesis () depending on the version Info field.

A typical line in the pads-signature-list could look something like this

```
ssh,v/OpenSSH/$2/Protocol $1/,SSH-([.\d]+)-OpenSSH[_-](\S+)
```

Where ssh is the service, OpenSSH is the vendor product name $2 is a variable used in a regular expression to pick out a string in the line the regular expression works on. $2 is the version, and Protocol $1 is the protocol name. Next comes the main signature, a regular expression where one uses parenthesis to pick out strings to use in the version info.

prads - Passive real-time asset detection system

prads[15] or passive real-time asset detection system, listens passively in a network and gathers information about hosts and services it sees in the network. The information can be used to map a network, or figure out which hosts/services that exists. prads has a Perl implementation, but after talking to one of the developers, Edward Fjellskl he gave the impression that a new C implementation is soon to be released.

prads is not only an operating system detection tool like p0f, but it also tries to detect assets. Meaning that it can detect what is running on the detected computer.

In contrast to p0f, prads has the ability to run detection on all the different states, regarding the TCP packet. When running p0f, one must decide if one should run in SYN, SYNACK or RST mode. prads covers all these modes at the same time, meaning that it can detect packets with SYN, SYN+ACK, RST and FIN at the same time. Even though RST and FIN modes exist, SYN and SYNACK are the most important modes for doing operating system detection. Meaning that the signature files for these modes contain the majority of signatures.

prads has a lot in common with p0f and PADS. The operating system detection side of prads is similar to p0f, and the asset detection is similar to PADS. prads uses the same format on the signatures as p0f does. Meaning that the same signature files can be used in both applications. Reading the TCP-syn.fp file, that consist of the TCP SYN fingerprints, in prads/etc directory shows that it is taken directly from p0f. The same can be said about the vendor/mac address file that is almost a direct copy of the pads-ether.sig file.

Below is a typical output from prads. Looking closer one can see that it looks much like the PADS output. The difference is that there also is detected an operating system.

```
128.39.73.90,0,1033,6,SYN,[65535:127:1:48:M1460,N,N,S:.:Windows:2000 SP4, XP SP1+:link:ethernet/modem]
```

The next output is an asset detection from prads. Looking at the service it is a ssh service running the OpenSSH 4.7p1 implementation.

```
128.39.73.90,0,22,6,SERVER,[ssh:OpenSSH 4.7p1 (Protocol 2.0)],0,1302891973
```

Collecting all these lines in the detection, one can tell a lot about one ip address. With the tool asset-report, prads gather everything about every ip address and prints it in a report giving a better view of what the ip contain. Below is a typical example of how ip address 195.93.80.36 is shown in this report. Seeing what operating systems and services it runs. In this case ip 195.93.80.36 runs a web server at Windows 2008 server. Interesting is it also to see that prads is rating how good the operating system detection is. Looking at the unknown operating system it is of course 0% but looking at the Windows server 2008 it is a 60% chances that this is correct.

```
167 -----------------------------------------------------
IP:   195.93.80.36
DNS:  blog-win-cs-blogs-frr.evip.aol.com
OS:   Windows Server 2008 (UC) (60%) 1

Port  Service    TCP-Application
80    SERVER     @www

168 -----------------------------------------------------
IP:   198.78.205.126
OS:   unknown unknown (0%) 1

Port  Service    TCP-Application
80    SERVER     @www
```

Having worked with p0f, PADS and prads one fast realise that prads is a direct merge between p0f and PADS.

Ettercap

Ettercap[16] is not only a passive fingerprinting tool, but also a suite for the man in the middle attack. Ettercap has a lot of functions some of them are sniffing of live connections, content filtering on the fly and so on.

NetworkMiner

Networkminer[17] is a network forensic tool for Windows. Passive operating system detection is one of Networkminer's features. Networkminer focuses on the host rather than the packets. A good example is that the information created by Networkminer is grouped by host, rather than as a list of packets.

Nmap

Nmap[18] or network mapper is one of the most famous security tools known to system administrators. Nmap was actually Security Product of the Year by Linux Journal, Info World, LinuxQuestions.Org, and Codetalker Digest. Active fingerprinting is only one of Nmap's many features. Nmap can do port scanning, operating system detection, version detection, ping sweeps, and more.

Satori

Satori[19] that is another passive fingerprinting tool, uses WinPCap. This program listens on the wire for all traffic and does operating system Identification based on what it sees. Satori does not only look at the TCP's SYN or SYN+ACK packet but also looks at the rest of the IP stack. Satori can identify: Windows Machines, HP devices (that use HP Switch Protocol), Cisco devices (that do CDP packets), IP Phones (that send out Skinny packets), and a lot of DHCP related stuff recently, plus some other things

SinFP - single-port active/passive fingerprinting

SinFP[20] is and passive and active fingerprinting tool, developed in Perl.

XProbe2

XProbe2[21] is an active fingerprinting tool, that not uses the TCP protocol, but rather uses a combination of the ICMP(Internet Control Message Protocol) and the UDP(User Datagram Protocol) protocol.

2.2.5 Counter measures / Fingerprint scrubbing

Looking at network security and fingerprinting, there always is two sides. In this case, those who do the operating system detection, and those who want to deny it. Looking at active fingerprinting, preventing fingerprinting can be done by limiting the type and amount of traffic going in and out from a network. Typical with a firewall. But when it comes to passive fingerprinting, no traffic is sent, meaning that it does not help with a firewall. Maybe the simplest way to defeat passive fingerprinting must be to modify the default values of the TCP/IP stack implementation. It could be to modify the TTL, some TCP options or the window size. This way the fingerprinting applications will not be able to detect the correct operating system. Fingerprint scrubbing is another approach that is interesting. Doing Fingerprint scrubbing we have applications often called a TCP/IP fingerprint obfuscator. There exist applications for all the major operating systems Windows, Linux and MacOS. The obfuscator or "fingerprint scrubber" is often placed between the internet, and the network under protection. Typically in the firewall. Here the obfuscator conduct a set of kernel modifications to avoid recognition of the operating system based on the characteristics of IP and TCP implementations. An application called IP

Personality is a patch to the Linux kernel. IP Personality[22] has the ability to have different 'personalities' network wise, that means to change some characteristics of the network traffic, depending on different parameters.

The characteristics that can be changed are:

- TCP Initial Sequence Number (ISN)
- TCP initial window size
- TCP options (their types, values and order in the packet)
- IP ID numbers
- answers to some pathological TCP packets
- answers to some UDP packets

This patch depends on the framework created by Rusty Russel called Netfilter[23]. More precisely, the patch adds a new Iptables target (in a kernel module) that can be used in the mangle table with (a patched) Iptables. This target is very configureable.

Another application is OSfuscate[24]. This is an application made for Windows. This application takes use of profile files(.ini-files) that gives the user the ability to change the profile of the operating system. This application has a very simple graphical user interface, like shown in figure 2.4. Here one can change the profile of the operating system tricking applications like p0f and prads.

Figure 2.4: Screenshot of Osfuscate

2.3 Honeyd

Honeyd[25] is a small daemon that creates virtual hosts on a network. The hosts can have different ”personalities” and services. Honeyd makes a single host have the possibility to have several ip addresses, the author of Honeyd Niels Provos has tested up to 65536 on a lan for network simulation. Honeyd was made to improve cyber security by providing mechanisms for threat detection and assessment. Honeyd can also be used to hide real systems in the middle of a virtual system. Even though Honeyd is open source, released under GNU General Public License, several bigger companies use Honeyd.

The big advantage with Honeyd is the possibility to run several different operating systems on the same computer. If this should be done without running it virtual it would demand many computers and operating system licenses. Also running this with a real virtual system like Xen or VMware, it would demand a powerful server and costly licenses. Now running and testing operating system detection can be done on every operating system that exist.

With Honeyd, not only computers can be virtualised the whole network can be made, with routers and switches like a real computer network. All the common network distributors like Cisco and Zyxel, are in the database over which vendors and what equipment can be virtualised.

So how does Honeyd work? The installation of Honeyd is actually quite straightforward. The latest release of Honeyd makes us install it with aptitude in Ubuntu. When installing Honeyd with aptitude several other packages goes along with Honeyd. Packages that must be installed with Honeyd to make it work are:

libevent The API libevent[26] gives a way of executing a callback function when a specific event occurs on a file descriptor or after a timeout has been reached. It also support callbacks due to signals or regular timeouts. Libevent can be used to replace the event loop that can be found in event driven network servers.

libpcap Libpcap the packet capture library gives to user a high level interface to packet capture systems. This gives the user the possibility to access all the packets on the network even those that are destined for other hosts on the system.

libnet-dev What libnet does is to make a generic interface to many different network drivers. What this again does, is to help us write network applications without having to pin it to a particular type of network.

libdumbnet Libdumbnet[27] or libdnet is a library that provide a simplified, portable interface to several low-level networking routines.

libedit-dev Libedit[28] is a non-GPL replacement for readline library. A spin-off from Net BSD code.

zlib1g-dev zlib[29] is a library that is used to deal with lossless data-compression on almost every hardware and operating system that exist. The authors of zlib, Jean-loup Gailly, that has done the compression part, and Mark Adler that has done the decompression part, are both known for their work in the "compression world". Both have worked on gzip which is much used in the Linux world, and Mark Adler was also the author of Zip a very known compression method.

2.4 Tcpdump

Tcpdump[30] is a packet analyser that typical runs under the command line in a linux based operating system. Tcpdump can capture and display TCP/IP and other packets going through or into the machine containing Tcpdump. Tcpdump is distributed under the BSD license and is free software. Tcpdump uses the libpcap library to capture packets in Unix-like operating systems, but it also exist a port to windows that is called WinDump. This port uses the WinPcap library which is a port of libpcap to windows. Many network applications support the pcap format which is the way Tcpdump stores packet sniffed by Tcpdump. This way different applications can be run on the same Tcpdump. This is typical used in forensics or network analysis. Below one can see a typical output from Tcpdump[31].

```
12:00:19.042948 IP 52.210.16.62.customer.cdi.no.60552 > gateway.ssh: . ack 357824 win 33208 <nop,nop,timestamp 571221253 1035533192>
12:00:19.042950 IP 52.210.16.62.customer.cdi.no.60552 > gateway.ssh: P 289:337(48) ack 357440 win 33304 <nop,nop,timestamp 571221253 1035533192>
12:00:19.043157 IP gateway.ssh > 52.210.16.62.customer.cdi.no.60552: P 364624:365360(736) ack 337 win 478 <nop,nop,timestamp 1035533201 571221253>

2010 packets captured
2010 packets received by filter
0 packets dropped by kernel
root@gateway:~#
```

2.5 Previous work

In the area of passive operating system fingerprinting or active operating system fingerprinting, not many research papers are written.

In the article ”Passive fingerprinting” by Lanze Spitzner[11] He explains what passive fingerprinting is, how it works and how to use it. It covers the difference and similarities in passive and active fingerprinting. The article describes the different TCP packet headers and how these can be used to do a passive fingerprint against a remote host.

The same author Lanze Spitzner talks also about knowing your enemy in the article ”Know your enemy”[32]. In networking knowing your enemy and your assets are really important. Knowing your enemy makes it much easier to protect yourself against dangers. The threat Spitzner talks about is a typical threat, and that is the script kiddie. What the script kiddie does is to focus on a small number of exploits and then searching the internet for just these vulnerabilities which his exploits can take use of. Becoming a script kiddie is easy and almost everyone with some knowledge to computing can use tools for hacking. this means that eventually you will be hacked.

Another article, not actually covering fingerprinting but covering important basic security, by Gerald A. Marin Network ”Security Basics” [33] looks at general network security. The author describes different attacks like Smurf attack, land attack and the DDos attack. It also looks at different countermeasures, looking at what intrusion detection system is and how to stop worms, trojans and malicious code.

Masking approach to secure systems from Operating system Fingerprinting by Surbhie Kalia and Manider Singh is about the vulnerability the operating system is. When the hacker knows which operating systems is running on a host, it is much easier for the hacker to make an attack. Different operating systems have different vulnerabilities. Further more the paper discusses the primary phases that every operating system fingerprinting tool uses to detect the operating system of a remote system. The paper looks at tools like Nmap and Xprobe2 that are tools for active operating system detection. The most important coming out from this paper are countermeasures that shall prevent operating system detection.[34]

In the paper Ambiguity Resolution via Passive OS Fingerprinting by Greg Taleck [35] he looks at attacks that exploit differences in common operating systems to evade IDS detection. The paper describes an approach to use passive operating system detection to correctly resolve ambiguities between different network stack implementations. The paper also looks at a new technique to increase the confidence level of a operating system detection, looking closer at the TCP connection negotiations.

2.6 Summary background chapter

There are to main types of fingerprinting. Active and passive. The active way uses an application to make a custom made packet that can be used to determine a remote hosts operating system. The passive way contains a packet sniffer that sniffs packets and compares the packet to a database or files with signatures. The database/files consist of different fields that are demanded to determine which operating system it is. Not all the fields in a signature exist for every operating system, but put together the operating system can be detected, though not with 100% accuracy. Even though calling it operating system fingerprinting or detection, we can also use some of these applications to determine services running on the remote host. Therefore the name asset detection. Later in this thesis I will use some of the passive operating system detection applications mentioned earlier to do some testing. The research done in the area for passive operating system detection is not extensive. Looking a bit wider in the network security realm, more research is done. The reason for looking at Honeyd is to use this application in testing detection. Honeyd can simulate different network profiles and can maybe be used to simulate many operating systems.

Chapter 3

Approach

This is an investigative study and the investigation will be based on methods that will be described in this section. The experiments will be based on the problem statements, and will be described in detail.

3.1 Test environment

Experiments that will be conducted in this thesis will be either run virtually on a Xen server or on a computer connected to the 128.39.73.0 network. Later in the thesis, a controlled environment is mentioned. What this means is a set up where we control all the computers, meaning that we can control the traffic going in and out of all the computers. The virtual controlled setup, is a setup of six virtual computers connected together in a virtual network. The gateway has a public ip address, that is 128.39.73.249. This gives the ability to connect to the computer remotely through ssh and vnc. The figure of the virtual network can be looked closet at in the figure 3.1.

The virtual computers in the network will have different operating systems, Windows and Linux distributions. This virtual system can be accessed through the gateway from everywhere, just having a computer with an internet connection.

In figure 3.2 we see an illustration of how the other test environment looks like. All traffic going through port 16 is mirrored with the mirror port and this traffic is being captured on the computer 128.39.73.9. To port x are computers in the 128.39.73.0 connected and when those computers connect out of this network the traffic most pass through port 16 that is mirrored

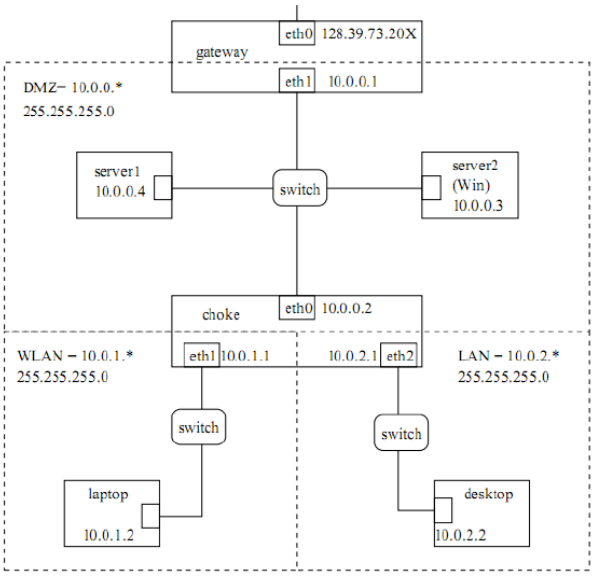

Figure 3.1: Test environment [Haarek Haugerud]

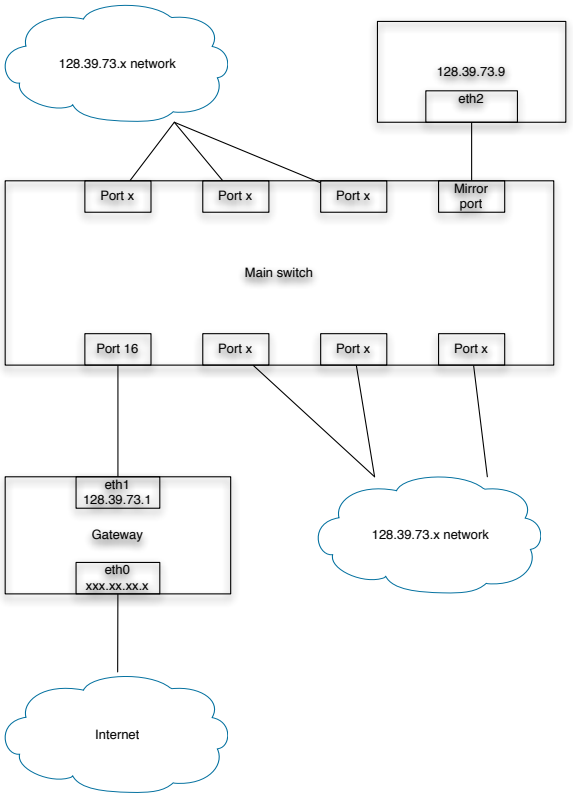

Figure 3.2: 128.39.73.0 Network

3.2 Comparing p0f and prads in the 128.39.73.0 network

In this experiment, learning about passive operating system detection, and finding out how p0f and prads works, is the main goal. This means to discover the applications functions and their limitations. How this will be done is to first collect packets that are going to be processed by the applications. Then, running the applications on these packets with different options enabled. At the end try to find some options that will give as similar output as possible to make a comparison and see how the applications differ and are alike. The comparison will be made out of observations and not through any thorough investigation or analysis.

When both applications are to be compared to each other, the importance of working on equal data is present. Meaning that when comparing p0f and prads, a comparison can't be done without a identical base. Both applications have the ability to read from pcap files(Tcpdump) instead of doing a live run and this gives the assignment the predictability it needs.

The first important decision to make is where to gather the data to run through the applications. To make the test as real as possible, running the Tcpdump in the schools 128.39.73.0 network is being done. This is accomplished by creating a mirror port on one of the important switches in this network. In this network we find the lab computers that are used in computer classes in the school. Using a mirror port gives the ability to capture all the traffic going in and out of the network and it also gives a predictability in the form that we have knowledge of when there is the most traffic on this system. In a school situation the most traffic are during the school day. More specific, during scheduled lab hours.

Connected to the mirror port is a Linux machine. On this computer is a user with super user privileges made. This computer has a public ip address and has the possibility to be connected to remotely. On this computer there will be installed Tcpdump, p0f and prads. These applications will also be installed on the virtual controlled system.

The Tcpdump will be run two times. One time for one day, the other will be for about four days. Being able to detect operating systems on both dumps is important in comparing the applications.

3.2.1 Installing applications

To be able to use the applications mentioned in this section, installing them must be done. Most applications are easily installed with aptitude, but prads need some additional configuring.

The first application to be installed is Tcpdump. Tcpdump is installed with the following command:

```
apt-get install Tcpdump
```

Installing Tcpdump it asks for some additional packages to be installed. Important is libpcap.

Next to be installed is p0f. As with Tcpdump, p0f can be installed with aptitude and the following command will install p0f.

```
apt-get install p0f
```

The last application is prads. prads cannot be installed with aptitude as the rest of the application, but with git. The procedure for installing prads is taken from its INSTALL documentation and it states as follows

```
sudo aptitude install libnet-pcap-Perl libgetopt-long-descriptive-Perl git-core libdbd-sqlite3-Perl
git clone git://github.com/gamelinux/prads.git
sudo ln -s $PWD/prads/etc /etc/prads
sudo Perl prads/sbin/prads.pl --help
```

Problems occurred trying to install prads this way, so after talking to the developer of prads he came up with the solution of doing it this way:

```
sudo aptitude install libnet-pcap-Perl libgetopt-long-descriptive-Perl git-core libdbd-sqlite3-Perl
git clone git://github.com/gamelinux/prads.git
cd prads/src/
make
sudo ln -s $PWD/prads/etc /etc/prads
sudo Perl prads/sbin/prads.pl --help
```

Now when running this application it is important to be in the correct directory. The directory to run the application from is the src folder of prads.

3.2.2 Running Tcpdump

Tcpdump needs some options set to be able to dump from the correct network interface and to enable to dump to a file instead of only dumping to the screen. A snapshot from Tcpdump's manual[36] says the following about choosing which interface to listen to:

```
-i     Listen  on interface.  If unspecified, Tcpdump searches the system interface list for the lowest
       numbered, configured up interface (excluding loopback).  Ties are broken by choosing the  earli-
       est match.
```

One can see that the way Tcpdump is to be run is with the options -i that is the interface option. Meaning that if a computer has several network interfaces one have to choose which network interfaces to dump from.

Next is to write the packets to a file instead of printing them out in the terminal window. This is also described in the Tcpdump manual pages:

```
-w     Write  the  raw  packets  to  file rather than parsing and printing them out.  They can later be
       printed with the -r option.  Standard output is used if file is -.
```

The following Linux command will run Tcpdump on the correct network interface and save to the correct file

```
Tcpdump -i eth2 -w petter.Tcpdump
```

After the Tcpdump is run and the collection of packets are completed, both p0f and prads must be tested with the Tcpdump. Default both applications run on a network interface, so to be able to run these applications towards a Tcpdump, specifying a file instead of a network interface must be done. For testing the performance of the applications, running the time command in linux will be done. This command will show how long each of the applications will run when doing an operating system detection. Running this ten times will be sufficient to say something about how long each application uses to run the operating system detection.

3.3 Running p0f on the Tcpdump

Below we can read from the snapshot of the p0f manual[37]. Here we see that we read from a pcap file by using the -s option followed by the pcap file name. As also seen in the Tcpdump manual, the p0f manual gives us the advice of running Tcpdump with the -w option that means that it writes to a file instead of just showing what is happening in the terminal window.

```
-s file
            read packets from Tcpdump snapshot; this is an alternate mode of operation,
            in which  p0f  reads  packet from pcap data capture file,
            instead of a live network.
            Useful for forensics (this will parse Tcpdump -w
            output, for example).
```

To log the output from p0f there is another option that must be set, and that is the -o option.

```
-o file
             write to this log file. This option is required for -d and implies -t.
```

The way p0f was run is with the following Linux command:

```
p0f -s /home/petter/logfiles/petter.Tcpdump -o /home/petter/log files/petter_Tcpdump.p0f
```

To collect the time p0f uses to run, just add time before p0f.
In the result chapter, the results from this run will be presented.

3.4 Running prads on the Tcpdump

From the prads manual pages[38], we see that reading from a pcap file has the option -r. Logging to a file has option -l followed by the filename.

```
-r <file>
 Read pcap <file>.


-l <file>
 Log assets to <file> (default: '/var/log/prads-asset.log')
```

This way prads was run:

```
prads -r /home/petter/tcpdump/petter.Tcpdump -l /home/petter/log files/petter_tcpdump.prads
```

Important is to be located in the correct directory when running prads. The directory is the src directory of prads. When running prads this way, prads will test all different methods of fingerprinting it knows. Not only SYN, but SYNACK, FIN and RST as well.

When comparing the applications, timing them, can show their performance. In Linux, there is a time[39] command that collects different statistics about the time the applications run with time, uses. This will be used to measure the performance of p0f and prads.

In the results chapter the results will be presented.

3.5 Comparing p0f and prads

Both p0f and prads have an extensive output with a lot of lines of text as output. In some way this has to be organised to be able to directly compare these two applications output. The lack of a good statistical tool in prads and p0f is the main motivation for a script, that will count the instances of an operating system and list this together with the operating system. This script will also count how many operating systems each ip address has got.

The way chosen to do this, is to make a Perl script that counts the operating systems detected either in p0f or prads. It's important that the same operating system is not counted several times for a single ip address. Still, one ip can have different operating systems in example, when a computer is used to do NATing. The script will also count the number of operating systems on every ip address. The script must be able to receive a file, either from p0f or prads and have the possibility to output a file with statistics about the detection. There will be made two script. One for p0f and one for prads. This script will enhance the operating system detection and help the system administrator by providing a better statistical view of the detection.

The other approach to comparing these to applications is to look directly into different ip addresses and check the operating system detection for those specific ip addresses. This approach will be interesting because of looking at a single ip address, one can look at the details of the passive detection, also being able to see if the detection is correct or not. The way this will be done is to grep[40], that means to search for something in a detection and print the line matching the grep, for the ip address. This way one can compare single ip addresses in p0f and prads.

A grep command will look like this:

```
cat petter_1504.prads | grep 128.39.73.73
```

This command will pick out all lines containing the ip address 128.39.73.73 from the prads file.

3.6 Summary comparing p0f and prads in the 128.39.73.0 network

The problem statements says that one should compare p0f and prads in the 128.39.73.0 network. The approach to this problem statement will be to try out the applications and compare them through observation. Observations means to look at the output from each application and explain what they are and what they could mean.

The idea by running both of the applications with a Tcpdump, instead of running each application live, is that one can work on the same material. To measure the actual output of the applications comparing the two, will not only be done by observing their output, but by writing some Perl scripts that will make statistics for each application. The script will count each instance of an operating system in this network, and count number of operating systems for a specific ip address.

3.7 New signatures

The background for this experiment is to investigate on the weaknesses on existing fingerprinting methods and at the same time look at what can make fingerprinting better. This experiment will be about exploring the signatures and their role in passive operating system detection. The idea behind the method used in p0f is used in many passive fingerprinting applications and it is this method this investigation will be about. Both prads and p0f uses the same method in doing operating system detection. So the application that will be tested in this experiment is prads. A typical passive fingerprinting application first sniffs packets, then based upon what kind of packet and what the packet contains, makes a decision of which operating system it is. Many of todays passive operating system detection systems bases their detection on the TCP

SYN and TCP SYNACK packets. In this experiment the testing has been done using prads. Since the signatures are different in the different modes prads operate, the focus will be on the SYN signatures and SYN signature file.

Often passive operating system detection does not do what it was intended to do. Meaning that the applications says that the operating system is ”unknown”. This is the motivation for this experiment. Adding new signatures is a way to improve the passive operating system detection applications. In short terms, this experiment will consist of first running prads, and provoke prads to produce an unknown operating system. Then add a new signature together with the new operating system, then run prads one more time to prove that the new signature works.

The approach to this assignment will differ a bit from the first assignment. As for the detection application, prads will be used not p0f. As in the first experiment, Tcpdump will be used together with prads. Tcpdump will be run in the controlled environment. The controlled environment is a setup of six virtual machines. 3.1 The test environment consist of five linux machines and one Windows machine. The gateway and choke is Ubuntu 8.04 machines. The desktop, laptop and server1 are Debian 4.0 (etch). The Windows machine called server2 is a Windows XP SP 3 machine. To be able to make a regular TCP connection from one computer to the other, in the network, a decision is made of installing a web server on the gateway. This way, using a browser, a TCP connection can be made from every computer to the gateway. As for web server software apache2 is preferred because of the ease of use and the simple installation process. In figure 3.3 we can see how the network will be, and what is installed where.

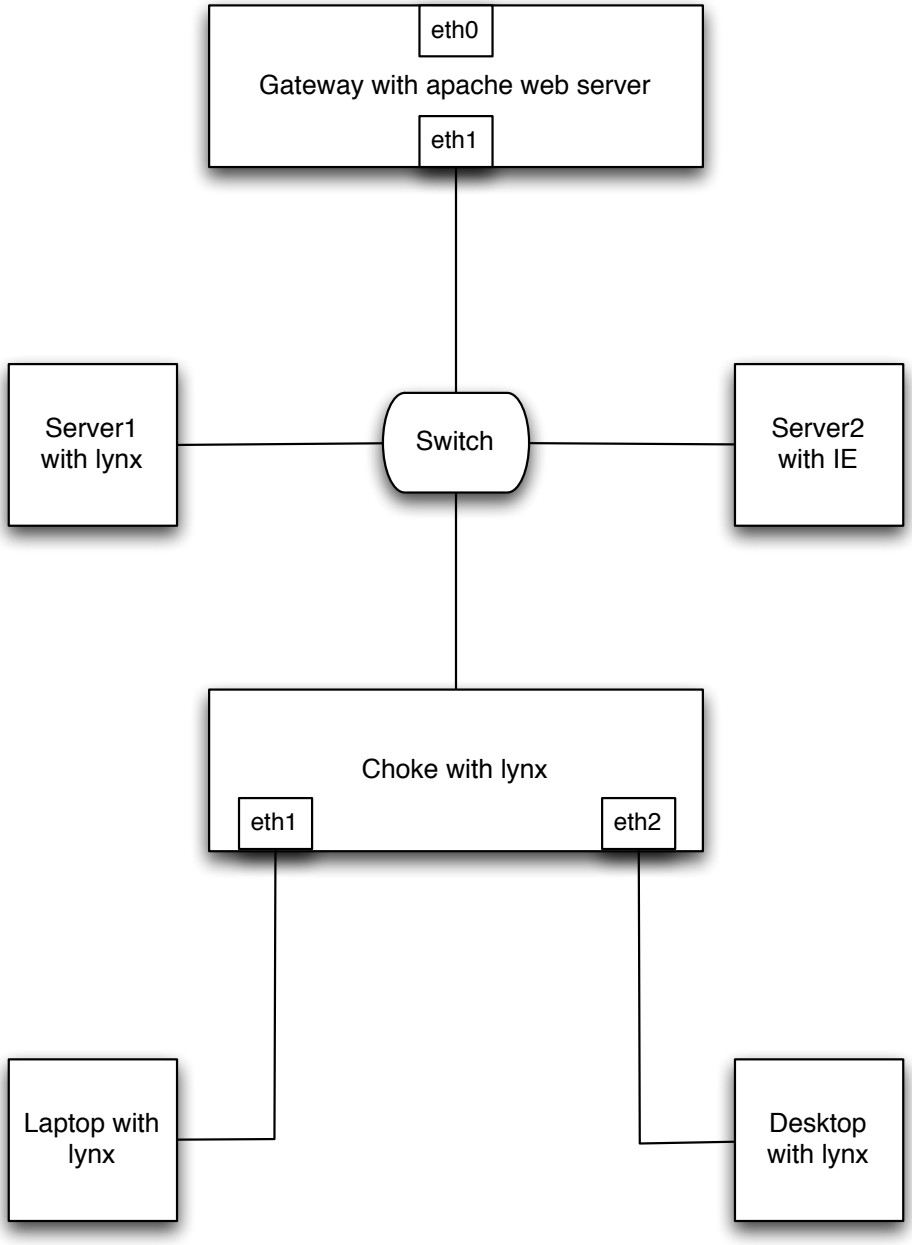

Figure 3.3: An overview of the network with the applications installed

Installing apache2 web server on the gateway is done with the following command:

```
apt-get install apache2
```

Testing the web server when finished installing, is done with opening a web browser and input the gateway's ip address. If it is working you get "it works" in the browser window.

In this experiment, Tcpdump must be run on one of the computers in the network. The question is on which. Looking at the overview of the network figure 3.1, the decision was made to run it on the gateway. This way when the computers connect to the apache web server on the gateway, all computers connect to it through the same network interface eth1. To be able to connect to the web server, an web browser must be installed on all of the computers. Since most of the computers are command-line based, a text based web browser called lynx will be installed. The installation was done with following the command:

```
apt-get install lynx
```

Next step is to do the testing. The testing will consist of one computer connecting to gateway that has a web server. This connection will be intercepted by Tcpdump that records what is happening on network interface eth1 on the gateway. This will then be done for all the computers in the virtual network, one by one.

After the Tcpdump is collected, prads will be run on the Tcpdump. From the output from prads, hopefully some unknown signatures will appear. If there exist an unknown signature there will be created a new signature with the knowledge of the unknown operating system.

Next, the same test will be conducted one more time, to see if there still exist any unknown signatures.

Chapter 4

Results

In the result chapter, the results given from running p0f and prads will be shown. The superior goal of this thesis is to investigate passive operating system detection The following chapter will first look at how p0f and prads work, it will look at the performance of the applications and the output from the applications.

4.1 Comparing p0f and prads in the 128.39.73.0 network

This section will first consist of running the applications, then timing the applications, next is to count instances of operating systems and at the end comparing one specific ip address. All this will be done for both p0f and prads and each section will have a comparison section that will look at the similarities and difference of p0f and prads.

4.1.1 Running the applications

This results will look at both of the applications, when running the applications. The focus will be on the behaviour of the applications during run and when the run is finished.

Running p0f

This result shows how p0f looks like when it does a regular run, and running p0f gives the following output in the terminal window 4.1

```
jonas:/home/petter/tcpdump# p0f -s petter.tcpdump -o /home/petter/logfiles/petter_tcpdump.p0f
p0f - passive os fingerprinting utility, version 2.0.8
(C) M. Zalewski <lcamtuf@dione.cc>, W. Stearns <wstearns@pobox.com>
p0f: listening (SYN) on 'petter.tcpdump', 262 sigs (14 generic, cksum 0F1F5CA2), rule: 'all'.
[+] End of input file.
jonas:/home/petter/tcpdump#
```

Figure 4.1: p0f terminal output petter.Tcpdump

The output shows that p0f is the version 2.0.8. It also states that it runs in SYN mode on the file petter.Tcpdump. When p0f is running, it shows the line that starts with p0f: and so on. When p0f is finished it writes

```
[+] End of input file.
```

p0f's interface is command-line based. It gives little information when p0f is running. If the run of p0f don't work, it will feedback an error message. Often this error message is cryptic and not very informative. When running p0f without a log file to write to, p0f writes directly to the screen. When cancelling a live run, p0f writes out the average packet ratio.

Next the output from p0f will be presented. It will be presented by a screenshot taken from the p0f log.

In figure 4.2 we see the log file, that has been constructed by p0f. Every entry in the log file have two lines in the file. The first line contains the date and time of when the detection has found place. Then comes the ip address of the computer where p0f detected the operating system. Next comes the main operating system, with specific information of kernel or version of the operating system together with different version information. On the next line we see the ip address of the destination computer.

```
jonas:/home/petter/logfiles# head petter_tcpdump.p0f
<Fri Apr 15 15:33:27 2011> 128.39.74.102:5367 - Windows XP SP1+, 2000 SP3 (2)
  -> 128.39.73.51:443 (distance 1, link: ethernet/modem)
<Fri Apr 15 15:33:30 2011> 128.39.73.29:54555 - Linux 2.6, seldom 2.4 (older, 4) (up: 12 hrs)
  -> 72.21.194.12:80 (distance 1, link: ethernet/modem)
<Fri Apr 15 15:33:31 2011> 128.39.28.22:59323 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
  Signature: [8192:127:1:52:M1460,N,W2,N,N,S:.:Windows:?]
  -> 128.39.73.51:443 (distance 1, link: ethernet/modem)
<Fri Apr 15 15:33:32 2011> 128.39.73.29:54556 - Linux 2.6, seldom 2.4 (older, 4) (up: 12 hrs)
  -> 72.21.194.12:80 (distance 1, link: ethernet/modem)
<Fri Apr 15 15:33:32 2011> 128.39.74.102:5368 - Windows XP SP1+, 2000 SP3 (2)
```

Figure 4.2: head petter Tcpdump p0f

p0f's output is not extensive. It contains the most important information, the ip address and the operating system. If the operating system could not be detected, p0f prints UNKNOWN - and the signature of the operating system. This way one can add this signature to the signature file, of course, if one knows which operating system that is detected. p0f does not tell what kind of detection the packet has gone through, so which signature file to add the signature to, must be checked in advance.

Running prads

Running prads gives the following output in the terminal windows shown in figure 4.3

```
jonas:/home/petter/prads/src# ./prads -r /home/petter/tcpdump/petter.tcpdump -l /home/petter/logfiles/petter_tcpdump.prads
logging to file '/home/petter/logfiles/petter_tcpdump.prads'
[*] Loading fingerprints:
  CS_MAC       ../etc/mac.sig
  CO_SYN       ../etc/tcp-syn.fp
  CO_SYNACK    ../etc/tcp-synack.fp
  CO_ACK       ../etc/tcp-stray-ack.fp
  CO_FIN       ../etc/tcp-fin.fp
  CO_RST       ../etc/tcp-rst.fp
  CS_TCP_SERVER ../etc/tcp-service.sig
  CS_UDP_SERVICES ../etc/udp-service.sig
  CS_TCP_CLIENT ../etc/tcp-clients.sig

[*] Running prads 0.2.3
    Using libpcap version 0.9.8
    Using PCRE version 7.6 2008-01-28
[*] OS checks enabled: SYN SYNACK RST FIN ACK
[*] Service checks enabled: TCP-SERVER TCP-CLIENT UDP-SERVICES ARP MAC
[*] Reading from file /home/petter/tcpdump/petter.tcpdump
[*] Sniffing...

-- prads:
-- Total packets received from libpcap    :     6590033
-- Total Ethernet packets received        :     6590033
-- Total VLAN packets received            :           0
-- Total ARP packets received             :       40839
-- Total IPv4 packets received            :     6443496
-- Total IPv6 packets received            :       77186
-- Total Other link packets received      :       28512
-- Total IPinIPv4 packets received        :           0
-- Total IPinIPv6 packets received        :           0
-- Total GRE packets received             :           0
-- Total TCP packets received             :     6192707
-- Total UDP packets received             :      299155
-- Total ICMP packets received            :        8021
-- Total Other transport packets received :       20799
--
-- Total sessions tracked                 :      151789
-- Total assets detected                  :        1011
-- Total TCP OS fingerprints detected     :        2419
-- Total UDP OS fingerprints detected     :           0
-- Total ICMP OS fingerprints detected    :           0
-- Total DHCP OS fingerprints detected    :           0
-- Total TCP service assets detected      :         426
-- Total TCP client assets detected       :         125
-- Total UDP service assets detected      :          46
-- Total UDP client assets detected       :         199

[*] prads ended.
jonas:/home/petter/prads/src#
```

Figure 4.3: prads terminal output petter.Tcpdump

The output from prads is extensive. First prads shows where the signature files are located. Next prads shows which version it is and which libpcap version it uses. Then prads shows which signatures it is testing, in this case it is all possibilities. SYN, SYNACK, RST, FIN and ACK. prads tells which services it looks for and from which file it reads from. Then it says sniffing...

When prads is doing the operating system detection prads says sniffing... This means that prads is still running. When the sniffing is done, the statistics is shown. Both when prads runs a live detection and when it runs a detection from a Tcpdump, it does not print the detection to the screen. If not specified, it uses the default log file called prads-asset.log. Using an option, prads can print the log to a self chosen file.

After prads is finished detecting operating systems it prints some statistical information. The statistical information given is different information about what kind of packets that have been processed by prads. It also says something about the detection done, when it comes to asset and operating system detection.

In figure 4.4 we see a snapshot from the prads log file. Every entry in the log file contain one line and one line only. The line start with the ip address of the computer which are fingerprinted, then comes what kind of vlan the computer is in. Next comes the port number. After the port number we see packet information if there is a SYN, ACK, SYNACK, RST or FIN. After that a lot of information like the fingerprint itself, and the operating system with version or kernel information. Followed by uptime and distance, at last, when the packet was discovered.

```
jonas:/home/petter/logfiles# head petter_tcpdump.prads
asset,vlan,port,proto,service,[service-info],distance,discovered
128.39.73.9,0,22,6,ACK,[120:64:1:0:N,N,T:AT:Linux:2.4(newer)/2.6:uptime:1693hrs],0,1302874406
128.39.89.9,0,36914,6,ACK,[1002:63:1:0:N,N,T:AT:Linux:2.4(newer)/2.6:uptime:1676hrs],1,1302874406
65.54.89.222,0,80,6,ACK,[6296:55:1:0:.:A:Windows:XP],9,1302874406
128.39.73.51,0,58725,6,ACK,[65335:127:1:0:.:A:Windows:XP],1,1302874406
128.39.28.22,0,58388,6,ACK,[17155:127:1:0:.:A:Windows:XP],1,1302874406
128.39.73.1,0,0,0,ARP (Cisco),00:12:44:81:88:00,0,1302874406
128.39.73.135,0,53,17,CLIENT,[unknown:@domain],0,1302874406
128.39.89.8,0,53,17,SERVER,[domain:DNS SQR No Error],1,1302874406
128.39.73.135,0,123,17,CLIENT,[unknown:@ntp],0,1302874406
```

Figure 4.4: head petter Tcpdump prads

prads prints the format of what is what in the log file. This is the start line in every log file prads makes. The different fields are more or less self explanatory. One field, the field called proto in the log file is the protocol. The protocol is stated as a number, so to know which protocol the detection is about, one must know the number of the protocols. In the first line in figure 4.4, the protocol states number 6, this is actually the TCP protocol.

The detection prads does is smart. This means that prads does not print every detection to the log file. If prads detects an operating system with a specific ip address one time, this detection will not happen again. A problem is that since prads runs in different modes, a specific operating system can have small name differences meaning that one operating system can be detected several times. When running detection in a bigger network, the prads output can seem a bit overwhelming. There is no sorting and there is now other way of viewing the detection other than viewing them in the log file or running them through the prads-asset-report script. This script sorts the detections

based on ip address. The problem is that when there is many ip addresses, this report also get very chaotic.

4.1.2 Run time command on the applications

Testing the performance and to see what resources both applications uses is important in comparing the two applications. Knowing what resources both applications use can be an important factor in choosing which application to use, and also looking at what hardware one needs, to run the applications.

To test the performance of p0f and prads, running a comparison on how long each applications uses has been done. Using the time command in Linux, the time it takes to run each application will be recorded. This command together with the application command itself, will give an output in the form of three different times. The real time, user time and sys time. The real time is the total time the application uses. The user and sys time is processor times.

When running the command with time, both applications are tested with the Tcpdump file collected through a day.

Timing p0f

The following command will time p0f running on the Tcpdump file collected through a day.

```
time ./p0f -s petter_1504.Tcpdump -o petter_Tcpdump.p0f
```

To be sure of the results, the test has been run ten times. A small difference will always occur, but if there is not much of a difference in the testing, the test is more or less reliable. Beneath is a table that shows the time in the ten runs.

real	**user**	**sys**
0m17.315s	0m1.232s	0m1.288s
0m16.863s	0m1.396s	0m1.340s
0m19.346s	0m1.324s	0m1.272s
0m18.609s	0m1.412s	0m1.240s
0m18.009s	0m1.336s	0m1.280s
0m20.509s	0m1.268s	0m1.528s
0m17.622s	0m1.396s	0m1.260s
0m18.989s	0m1.200s	0m1.320s
0m19.453s	0m1.444s	0m1.244s
0m20.820s	0m1.432s	0m1.168s

The focus will be on the real time. Real time is for how long the applications uses to run.

From the time numbers one can find the mean. The mean is all the numbers added together divided by the number of numbers. In this particular case the mean is 18,753 seconds. Looking at the numbers, there is not a big difference from the biggest to the smallest number, so stating that p0f uses about 18 seconds on the particular Tcpdump, is well documented.

Another and maybe better view of these number is the graph in figure 4.5. Here one can see that there is little difference from the shortest to the longest time. The graph consist of the ten runs, and the mean line is also added to see how the different runs differ from the mean.

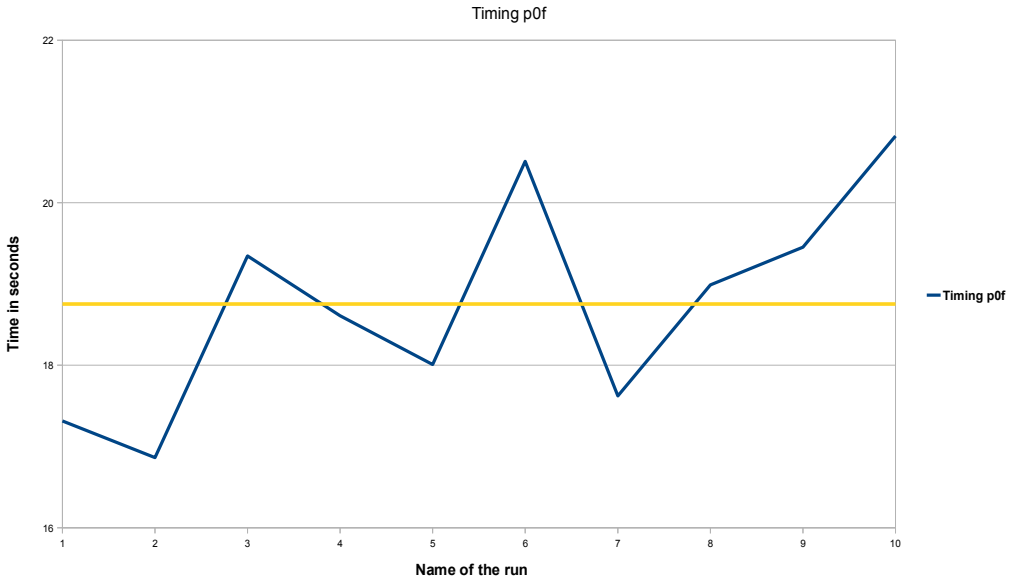

Figure 4.5: Graph over p0f time command

Timing prads

As with p0f, the time command has been used to time prads. Again the test is run ten times and on the same Tcpdump file.

The following command gives the time of the prads command:

```
time ./prads -r petter_1504.Tcpdump -l petter_Tcpdump.prads
```

The following numbers are the results of the ten runs.

real	user	sys
5m53.070s	5m48.730s	0m1.112s
5m52.408s	5m47.930s	0m1.872s
5m44.270s	5m40.205s	0m1.216s
6m2.244s	5m58.046s	0m1.292s
5m57.201s	5m53.202s	0m1.132s
5m52.141s	5m46.890s	0m1.344s
5m55.507s	5m51.586s	0m1.336s
5m46.596s	5m42.801s	0m1.032s
5m53.928s	5m50.466s	0m1.192s
5m47.690s	5m43.821s	0m1.320s

prads differ a bit from run to run. The mean which is 5,52 is good measure for how long prads takes to run on this particular Tcpdump is correct.

In figure 4.6 we see a graph that describes the prads run, also added is the mean line. This line shows how the runs differ from the mean. It's important to know that the times is converted to seconds from minutes.

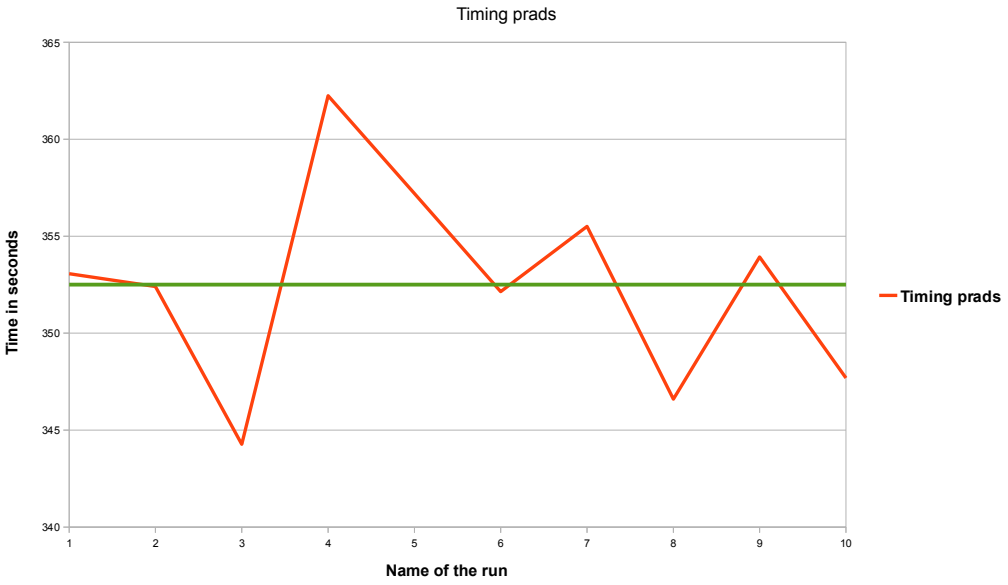

Figure 4.6: Graph over prads time command

Some of the runs is a bit away from the mean and a possible explanation is that other processes could be running in the background when prads is run. Since the computer that the test is run on is shared with someone else, knowing what is running at the same time is difficult to be aware of.

Timing prads with the -XS option

To compare the applications, running prads with the -XS option produces a more similar test. Running prads with -XS turns of asset detection and will let us test prads with only the SYN fingerprints. If this will make prads use less time, will be interesting two observe.

The following command tests prads with only SYN operating system fingerprinting.

```
time ./prads -XS -r petter_1504.Tcpdump -l petter_Tcpdump.prads
```

As the other tests, this will be tested ten times to see that the numbers given are more or less correct

real	user	sys
5m6.942s	5m3.003s	0m0.804s
5m20.456s	5m16.400s	0m1.200s
5m29.931s	5m24.840s	0m1.376s
5m27.756s	5m22.692s	0m1.384s
5m36.948s	5m31.769s	0m1.180s
5m24.304s	5m19.604s	0m1.360s
5m30.169s	5m25.268s	0m1.180s
5m32.451s	5m27.624s	0m1.288s
5m26.580s	5m22.036s	0m1.196s
5m28.133s	5m23.208s	0m1.240s

The result indicates that only running SYN detection decreases the run time, but the time is still far away from p0f.

The results can also be viewed in figure 4.7.

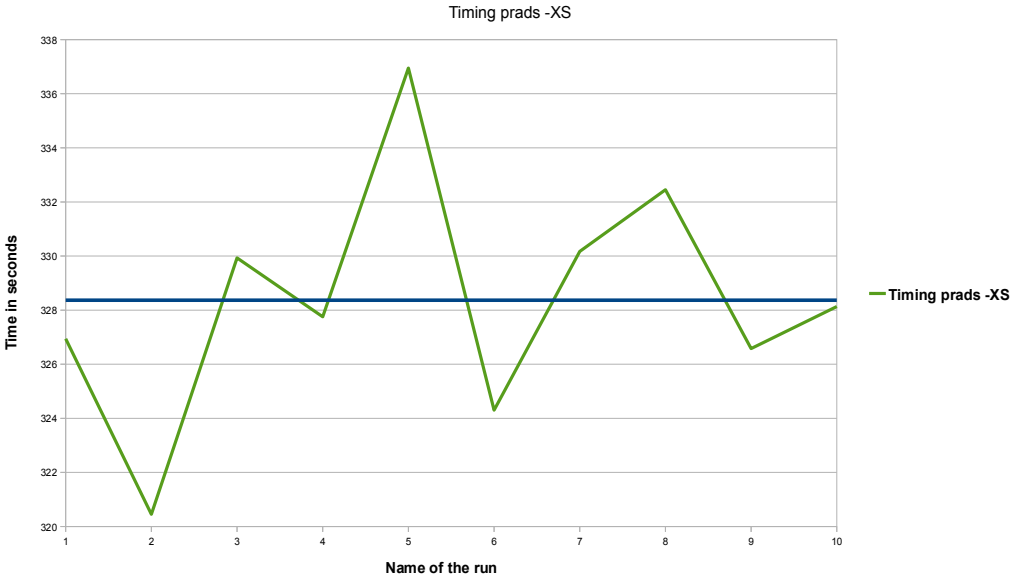

Figure 4.7: Graph over pradsXS time command

Summary of comparing run time for p0f and prads

Looking at the run time, it's quite interesting to see how fast p0f can do the detection. With only 18 seconds p0f runs detection on the same amount of packets that prads do. Before running the test where prads was run only with SYN mode, the suspicion was that since prads runs detection on all the different modes and asset detection, the run time is much longer, but after testing only running prads with SYN detection and no asset detection, prads still uses much more time than p0f does. As mentioned earlier prads is smarter than p0f. It could be that this "smartness" has a big impact on prads performance. It could be the test that checks if the operating system is detected before, that has an impact on the performance. Even though this could have some impact on the run time, it would probably not mean several minutes in difference. Comparing source code could be an approach to see the actual reason for prads using much more time than p0f.

4.1.3 Counting ip addresses and operating systems

prads and p0f is applications yet to be developed further. They can detect operating systems, but both applications have problems especially with the presentation of the data coming out from the applications. The results presented in this section, is from the output of some scripts, created by the author of this master thesis, to improve the applications further. This script looks at the ip address and the operating system and tries to count the number of operating systems existing in the network. It's important that this script exclude operating systems that is already counted. Meaning that for one ip address, a specific operating system will not be counted several times. In addition to counting instances of operating systems, the script also counts how many operating systems each ip address have. Unknown operating systems is also counted. Looking at each ip address one can see if the detection is correct or if the detection has detected to many or to few operating systems. In the result chapter will only a few ip addresses be shown or else the list of ip address would be to long. To see how p0f and prads works, looking into the data from both applications will be done.

In the appendix chapter is the scripts oscounter_p0f.pl section 6.1 and oscounter_prads.pl section 6.2 added. The script is tested both with outputs from p0f and prads. Both with a Tcpdump run over one day and over four days.

Counting operating systems p0f

Counting instances of operating systems is done with following command:

```
oscounter_p0f.pl -f petter_1504.p0f -o petter_1504_p0f.counted
```

The output from the script could either be written to a file or written to the screen.

Operating system	**Number of instances**
Linux 2.6 (newer, 1)	36
Windows XP/2000 (RFC1323+, w+, tstamp-)	8
Linux 2.6 (newer, 3)	7
Windows 2000 SP2+, XP SP1+ (seldom 98)	2
Linux 2.6 (newer, 2)	1
Linux 2.6, seldom 2.4 (older, 4)	1
UNKNOWN [S4:64:1:60:M1460,S,T,N,W4:.:?:?]	1
UNKNOWN [S4:62:1:60:M1460,S,T,N,W4:.:?:?]	1

The script also prints out the total number of ip addresses that have been detected, containing an operating system, in the specified network, that is the 128.39.73.0. The total number of ip addresses that have been detected is 45. Summarising the total number of operating systems we get 57. This means that in average there is: the number of operating systems divided by the number of ip addresses detected. Which is 57 / 45 = 1,26 operating systems per ip address.

The scripts is also tested on the Tcpdump that has been run over four days.

Operating system	**Number of instances**
Linux 2.6 (newer, 1)	38
Linux 2.6 (newer, 3)	33
Windows XP/2000 (RFC1323+, w+, tstamp-)	11
Linux 2.6 (newer, 2)	7
Windows 2000 SP2+, XP SP1+ (seldom 98)	7
Linux 2.6, seldom 2.4 (older, 4)	1
UNKNOWN [S4:64:1:60:M1460,S,T,N,W7:Z:?:?]	3
UNKNOWN [S4:63:1:60:M1460,S,T,N,W4:.:?:?]	2
UNKNOWN [S4:64:1:60:M1460,S,T,N,W5:Z:?:?]	2
UNKNOWN [1024:56:0:44:M1460:.:?:?]	1
UNKNOWN [3072:46:0:44:M1460:.:?:?]	1
UNKNOWN [3072:50:0:44:M1460:.:?:?]	1
UNKNOWN [3072:58:0:44:M1460:.:?:?]	1
UNKNOWN [2048:41:0:44:M1460:.:?:?]	1
UNKNOWN [S4:64:1:60:M1460,S,T,N,W10:.:?:?]	1
UNKNOWN [S4:62:1:60:M1460,S,T,N,W4:.:?:?]	1
UNKNOWN [3072:54:0:44:M1460:.:?:?]	1

In four days, the total number of ip addresses detected is 65. Summarising the total number of operating systems we get 112. This means that in average there is: the number of operating systems divided by the number of ip addresses detected. Which is 112 / 65 = 1,72 operating systems per ip address.

In principle one could believe that the total number of operating systems should be equal to the number of ip addresses. This is not the case. Looking at the total number of operating system detected in both cases, we see that there is detected more operating systems than ip addresses. The majority of the operating systems is Linux. Looking at the output we can see that there are different Linux 2.6 versions. Called newer 1, newer 2 and so on. Could this actually be the same operating systems detected several times?

One possible explanation to why it is detected more operating systems, than ip addresses, is that in this network there can be several computers and operating systems behind an 128.39.73.0 ip address. Often there is one Linux computer and one windows machine. But there also occur cases where there is five Linux machines and one windows machine behind a single ip address. Looking at the output one can see that there is some UNKNOWN signatures. These are signatures where p0f could not detect the operating system. Knowing what these are, one can add these signatures to the signature file and detect those operating systems on a later point.

Counting Operating system for a specific ip address p0f

Since there are so many ip addresses detected, there will be a small selection of ip addresses in this result. The ip address selection is chosen because in

this range of ip address, we know that there should only exist two operating system on each ip address. Every ip should have one Linux and one Windows computer.

IP address	**Number of operating systems**
128.39.73.71	1
128.39.73.72	1
128.39.73.74	1
128.39.73.75	2
128.39.73.78	1
128.39.73.79	1
128.39.73.249	2

The results indicate that there is some ip addresses with one operating system, and a couple with two operating systems. The reason for this is that only one of the computers could have been used when the tcpdump was collected.

The results shown is interesting, and to be able to understand why the results shows different number of operating systems, one has to look closer at the ip address. In this case looking at two different ip addresses. One ip that has two operating systems, and one ip that has one operating system detected.

The first ip address is 128.39.73.73. Beneath we see that the detection has been as expected. Two computers is located under the ip address. One Linux machine and one Windows machine.

```
<Sat Apr 16 01:05:29 2011> 128.39.73.75:43484 - Linux 2.6 (newer, 1) (up: 10 hrs)
<Sat Apr 16 01:06:09 2011> 128.39.73.75:1158 - Windows 2000 SP4, XP SP1+
```

The next ip address 128.39.73.74 has only detected one operating system. This is ok, since it could be that the Windows machine could have been ”silent” when the Tcpdump has been collected.

```
<Sat Apr 16 00:45:31 2011> 128.39.73.74:54890 - Linux 2.6 (newer, 1) (up: 10 hrs)
```

Even though there is only shown a few lines in the example above, p0f has detected the same operating system for one ip address many times.

Counting operating systems prads

An almost equal script to the p0f script, that counts operating systems for p0f, is also made to count the instances of operating systems for prads. The script will be tested on the Tcpdump running one day and four days. In addition it will be tested on the prads file that is run with the -XS option meaning that it will be more similar to p0f. This way it will be easier to compare p0f and prads.

Running the script oscounter_prads.pl 6.2 on the prads file run for one day, gives the following results:

Operating system	**Number of instances**
Linux 2.6 (Generic 2, SYN from Windows)	128
Linux 2.6 (newer, 5)	76
Linux 2.6 (newer, 7)	55
Linux 2.6	45
Linux Ubuntu 8.04	36
Linux recent 2.4 (2)	10
Windows XP/2000 (RFC1323+, w+, tstamp-)	8
Windows 2000 SP4, XP SP1+	7
Linux 2.6 (newer, 6)	4
Windows 2000 SP2+, XP SP1+ (seldom 98)	2
Windows 98 (SE)	2
Linux 2.6 (Generic dfrag+)	2
Linux 2.6 (newer, 3)	1
Linux 2.6, seldom 2.4 (older, 4)	1
Windows 2000 SP4	1
Linux recent 2.4 (1)	1
Linux 2.6 (Syn from Unknown)	1
Brother HL-1270N	1
unknown unknown 64:1:52:N,N,T:ATFN	81
unknown unknown 63:1:52:N,N,T:ATFN	5
unknown unknown 127:1:52:M1460,N,W0,N,N,S:A	2
unknown unknown 64:1:48:M1460,S,E,E:PA	2
unknown unknown 64:1:52:M1260,N,W4,S,E,E:PA	2
unknown unknown 120:1:44:M1460:A	1
unknown unknown 64:1:60:M1460,S,T,N,W10:ZAT	1
unknown unknown 64:1:*(787):N,N,T:ATFDN	1
unknown unknown 64:1:44:M1460:A	1

Number of total ip addresses detected in the 128.39.73.0 network were 141. Summarising the total number of operating systems we get 477. This means that in average there is: the number of operating systems divided by the number of ip addresses detected. Which is 477 / 141 = 3,4 operating systems per ip address.

Running the same test on the four day file gives the following results:

Operating system	**Number of instances**
Linux 2.6 (Generic 2, SYN from Windows)	150
Linux 2.6 (newer, 5)	102
Linux 2.6	58
Linux 2.6 (newer, 7)	45
Linux Ubuntu 8.04	38
Linux recent 2.4 (2)	31
Windows XP/2000 (RFC1323+, w+, tstamp-)	11
Linux 2.6 (newer, 6)	9
Windows 2000 SP4, XP SP1+	7
Windows 2000 SP2+, XP SP1+ (seldom 98)	7
Linux 2.6 (Generic dfrag+)	4
Linux 2.6 (newer, 7 fedora12)	3
Linux 2.6 (newer, 3)	2
SunOS 4.1.x	2
Linux 2.6 (newer, 4)	2
Linux recent 2.4 (1)	1
Windows 2000 SP4	1
Windows 2000 (1)	1
Linux 2.6, seldom 2.4 (older, 4)	1
Windows 98 (SE)	1
Brother HL-1270N	1
Unknown signatures	207

Total number of ip addresses detected in the 128.39.73.0 network is 159. Summarising the total number of operating systems we get 684. This means that in average there is the number of operating systems divided by the number of ip addresses detected. Which 684 / 159 = 4,3 operating systems per ip address. The unknown operating systems is added together to keep the list short enough for displaying it in the thesis.

The next results is from the prads -XS option that only looks at the SYN packets. The test is run on the Tcpdump that was captured during one day.

Operating system	**Number of instances**
Linux Ubuntu 8.04	36
Windows XP/2000 (RFC1323+, w+, tstamp-)	8
Windows 2000 SP4, XP SP1+	7
Linux 2.6 (newer, 7)	7
Linux 2.6 (Generic dfrag+) 2 Windows 2000 SP2+, XP SP1+ (seldom 98)	2
Linux 2.6, seldom 2.4 (older, 4)	1
Linux 2.6 (newer, 6)	1

Total number of ip addresses detected in the 128.39.73.0 network is 46. Summarising the total number of operating systems we get 62. This means

that in average there is the number of operating systems divided by the number of ip addresses detected. Which 62 / 42 = 1,47 operating systems per ip address.

Looking at prads, it detects many more operating systems than p0f does. prads also detects more unknown signatures. This must be because prads tests many more packets than p0f does. It does testing not only on SYN packets, but also on SYNACK and FIN. Looking at the test where prads is run with the -XS command, p0f and prads have detected about the same amount of operating systems. The operating system that is counted 36 times both in prads and p0f have actually the same signature. The reason for it being Linux 2.6 (newer, 1) in p0f and Linux Ubuntu 8.04 in prads is that the signature file is changed in prads. Meaning that Linux 2.6(newer,1) equals Linux Ubuntu 8.04.

In prads and as with p0f, in theory one could think that there should be as many operating systems that there is ip addresses. But there are many more operating systems. In every test run. "Worst" is the test that is run over four days. As with p0f it looks like some of the same operating systems can have small variations in the signature. This small variation means that detection for one operating system with a specific ip addresses could be done several times. Does this mean that a computer can send packets with small variations in the header? To investigate this, we have picked out an ip address to see how the detection is done on this address.

```
mln6:/home/petter/Tcpdump# cat petter_1504.prads | grep 128.39.73.72
128.39.73.72,0,53,17,CLIENT,[unknown:@domain],0,1302874507
128.39.73.72,0,45743,6,SYN,[S4:64:1:60:M1460,S,T,N,W5:.:Linux:Ubuntu 8.04
128.39.73.72,0,80,6,CLIENT,[unknown:@www],0,1302874507
128.39.73.72,0,22,6,SYNACK,[S4:64:1:48:M1460,N,N,S:ZA:Linux:2.6 (Generic 2, SYN from Windows)
128.39.73.72,0,80,6,SYNACK,[S4:64:1:52:M1460,N,N,S,N,W5:ZA:Linux:2.6 (newer, 5)
128.39.73.72,0,80,6,CLIENT,[http:Apache 2.2.8 ((Ubuntu)],0,1302877367
128.39.73.72,0,80,6,SYNACK,[5792:64:1:60:M1460,S,T,N,W5:ZAT:Linux:2.6 (newer, 5)
128.39.73.72,0,80,6,CLIENT,[unknown:@www],0,1302889616
128.39.73.72,0,3167,6,SYN,[65535:127:1:48:M1460,N,N,S:.:Windows:2000 SP4, XP SP1+
128.39.73.72,0,22,6,SERVER,[ssh:OpenSSH 4.7p1 (Protocol 2.0)],0,1302891972
```

The knowledge we have about the 128.39.73.72 ip address, says that it should contain two operating systems. A Linux and a Windows operating system. Instead this ip address has four different operating systems, in addition a fifth that is the Linux 2.6 (newer 5) with a different signature than the other Linux 2.6 (newer 5). This means that when detecting an operating system, there can be small differences in the signatures. These small differences can trick prads into believing there are more operating systems than it should be. Another possibility can be that when there is a network where one computer does NATing this computer has an impact on the signature of the other computers, again tricking prads into believing that there are more operating systems than there actually is.

Counting Operating system for a specific ip address prads

A similar selection of ip address to p0f is selected to be able to compare the two applications. The two applications could not detect every ip address, so this is why not all the ip address is used in both tables.

IP address	Number of operating systems
128.39.73.72	5
128.39.73.73	3
128.39.73.74	4
128.39.73.75	8
128.39.73.76	3
128.39.73.77	3
128.39.73.78	5
128.39.73.79	4
128.39.73.80	3
128.39.73.249	3

Again to be able to compare to p0f, we have taken out two of the ip addresses where we have counted the number of operating systems.

Looking at the ip address table, this ip address should contain three operating systems. This is correct according to prads, but according to the knowledge we have, this is wrong. This ip address should only contain two operating systems. One Linux and one Windows.

```
mln6:/home/petter/Tcpdump# cat petter_1504.prads | grep 128.39.73.73
128.39.73.73,0,22,6,SYNACK,[S4:64:1:48:M1460,N,N,S:ZA:Linux:2.6 (Generic 2, SYN from Windows)
128.39.73.73,0,53,17,CLIENT,[unknown:@domain],1,1302877681
128.39.73.73,0,1060,6,SYN,[65535:127:1:48:M1460,N,N,S:.:Windows:2000 SP4, XP SP1+
128.39.73.73,0,80,6,CLIENT,[unknown:@www],1,1302877682
128.39.73.73,0,1062,6,FIN,[65390:127:1:40:.:AFN:unknown:unknown],1,1302877684
128.39.73.73,0,22,6,SYNACK,[5792:64:1:60:M1460,S,T,N,W5:ZAT:Linux:2.6 (newer, 5)
128.39.73.73,0,22,6,SERVER,[ssh:OpenSSH 4.7p1 (Protocol 2.0)],0,1302891973
```

The ip address 128.39.73.74 should have four operating systems according to the table, looking at the raw output from prads we can backup that this is correct. But again according to the knowledge we have about these computers there should only be a Linux and a Windows computer.

```
mln6:/home/petter/Tcpdump# cat petter_1504.prads | grep 128.39.73.74
128.39.73.74,0,53,17,CLIENT,[unknown:@domain],0,1302874529
128.39.73.74,0,55867,6,SYN,[S4:64:1:60:M1460,S,T,N,W5:.:Linux:Ubuntu 8.04
128.39.73.74,0,80,6,CLIENT,[unknown:@www],0,1302874529
128.39.73.74,0,55867,6,FIN,[2003:64:1:52:N,N,T:ATFN:unknown:unknown:uptime:1hrs]
128.39.73.74,0,22,6,SYNACK,[S4:64:1:48:M1460,N,N,S:ZA:Linux:2.6 (Generic 2, SYN from Windows)
128.39.73.74,0,22,6,SYNACK,[5792:64:1:60:M1460,S,T,N,W5:ZAT:Linux:2.6 (newer, 5)
128.39.73.74,0,22,6,SERVER,[ssh:OpenSSH 4.7p1 (Protocol 2.0)],0,1302891972
```

4.1.4 Look at ip address 128.39.73.66

To be able to compare the detection in both p0f and prads, a specific ip address has been chosen. Looking at this specific ip address in both p0f and prads lets us see how the passive detection applications work differently.

Looking at the ip address 128.39.73.66 in p0f

To be able to compare one specific ip address, the following command has been run on the p0f file, that had run for one day.

```
cat petter_1504.p0f | grep 128.39.73.66
```

This command prints every line with the ip address 128.39.73.66 to the screen.

```
petter@mln6:~/Tcpdump$ cat petter_1504.p0f | grep 128.39.73.66
<Fri Apr 15 15:55:19 2011> 128.39.73.66:49295 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 17:37:23 2011> 128.39.73.66:49296 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 19:34:27 2011> 128.39.73.66:49297 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:25:31 2011> 128.39.73.66:49298 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:25:35 2011> 128.39.73.66:49299 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:15 2011> 128.39.73.66:49300 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:23 2011> 128.39.73.66:49301 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:24 2011> 128.39.73.66:49302 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:35 2011> 128.39.73.66:49303 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:37 2011> 128.39.73.66:49304 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:38 2011> 128.39.73.66:49305 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:39 2011> 128.39.73.66:49306 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:40 2011> 128.39.73.66:49307 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:41 2011> 128.39.73.66:49308 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:43 2011> 128.39.73.66:49309 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:46 2011> 128.39.73.66:49310 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:48 2011> 128.39.73.66:49311 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:50 2011> 128.39.73.66:49312 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:52 2011> 128.39.73.66:49313 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:54 2011> 128.39.73.66:49314 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:55 2011> 128.39.73.66:49315 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:27:58 2011> 128.39.73.66:49316 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:28:07 2011> 128.39.73.66:49317 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:28:09 2011> 128.39.73.66:49318 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:28:11 2011> 128.39.73.66:49319 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:28:12 2011> 128.39.73.66:49320 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:28:15 2011> 128.39.73.66:49321 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 21:28:17 2011> 128.39.73.66:49322 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 22:28:33 2011> 128.39.73.66:49323 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Fri Apr 15 23:57:37 2011> 128.39.73.66:49324 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Sat Apr 16 00:08:54 2011> 128.39.73.66:49325 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Sat Apr 16 00:08:54 2011> 128.39.73.66:49326 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Sat Apr 16 00:09:06 2011> 128.39.73.66:49327 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Sat Apr 16 00:09:20 2011> 128.39.73.66:49328 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Sat Apr 16 00:09:24 2011> 128.39.73.66:49329 - Windows XP/2000 (RFC1323+, w+, tstamp-) [GENERIC]
<Sat Apr 16 00:18:03 2011> 128.39.73.66:44476 - Linux 2.6 (newer, 3) (up: 562 hrs)
```

Looking at the ip address 128.39.73.66 in prads

To be able to compare one specific ip address, the following command has been run on the prads file, that had run for one day.

```
cat petter_1504.prads | grep 128.39.73.66
```

```
128.39.73.66,0,22,6,SYNACK,[S4:64:1:48:M1460,N,N,S:ZA:Linux:2.6 (Generic 2, SYN from Windows)
128.39.73.66,0,53,17,CLIENT,[unknown:@domain],0,1302875590
128.39.73.66,0,49295,6,SYN,[8192:127:1:52:M1460,N,W8,N,N,S:.:Windows:XP/2000
128.39.73.66,0,443,6,CLIENT,[ssl:TLS 1.0 Client Hello],1,1302875719
128.39.73.66,0,22,6,SYNACK,[5792:64:1:60:M1460,S,T,N,W6:ZAT:Linux:2.6 (newer, 6)
128.39.73.66,0,22,6,SERVER,[ssh:OpenSSH 5.5p1 (Protocol 2.0)],0,1302891971
128.39.73.66,0,22,6,FIN,[124:64:1:52:N,N,T:ATFN:unknown:unknown:uptime:559hrs],0,1302891977
128.39.73.66,0,80,6,CLIENT,[unknown:@www],1,1302895535
128.39.73.66,0,80,6,FIN,[108:64:1:52:N,N,T:ATFN:Linux:2.6:uptime:562hrs],0,1302904477
128.39.73.66,0,44476,6,SYN,[S4:63:1:60:M1460,S,T,N,W7:.:Linux:2.6 (newer, 3)
```

The interesting difference between the two applications is that p0f collects many more lines than prads does. p0f has many identical lines, that don't need to be presented at all. As for the detection both applications have detected the same operating systems. One Linux 2.6 and a Windows XP/2000 machine. In addition prads has detected two extra Linux machines. This with the SYNACK packet. Even though this SYNACK detection has another signature, it's likely that this is the same operating system that was detected with SYN. With the background knowledge we have, we know that some of the ip addresses contain a Linux machine and a Windows machine. Could this be such a set up? Or could it be that this is a set up of five Linux machines and one Windows XP?

4.2 New signatures

The following results is captured by trying to improve prads further. What have been done is to capture some unknown signatures. Then adding those signatures to the signature file together with the knowledge we have about the operating system. Next time prads detects the same signature, prads will not detect the operating system as an unknown operating system.

To detect an unknown signature with an operating system that is known, collecting packets in the virtual network is done. Below we see the output from prads running on the Tcpdump given from the virtual network. Several unknown signatures are present.

```
asset,vlan,port,proto,service,[service-info],distance,discovered
10.0.0.2,0,22,6,SERVER,[unknown:@ssh],0,1304497807
10.0.0.2,0,55835,6,SYN,[S4:64:1:60:M1460,S,T,N,W5:.:unknown:unknown
10.0.0.1,0,80,6,SYNACK,[5792:64:1:60:M1460,S,T,N,W5:ZAT:unknown:unknown
10.0.0.2,0,80,6,CLIENT,[unknown:@www],0,1304497807
10.0.0.1,0,80,6,SERVER,[unknown:@www],0,1304497807
```

```
10.0.0.1,0,80,6,FIN,[215:64:1:52:N,N,T:ATFN:unknown:unknown:uptime:2512hrs],0,1304497807
10.0.0.2,0,55835,6,FIN,[216:64:1:52:N,N,T:ATFN:Linux:2.6:uptime:2512hrs],0,1304497807
10.0.2.2,0,22,6,SERVER,[unknown:@ssh],1,1304497808
10.0.2.2,0,4548,6,SYN,[S4:63:1:60:M1460,S,T,N,W3:.:unknown:unknown
10.0.2.2,0,80,6,CLIENT,[unknown:@www],1,1304497808
10.0.2.2,0,4548,6,FIN,[864:63:1:52:N,N,T:ATFN:unknown:unknown:uptime:2512hrs],1,1304497808
10.0.0.4,0,22,6,SERVER,[unknown:@ssh],0,1304497810
10.0.0.4,0,34935,6,SYN,[S4:64:1:60:M1460,S,T,N,W4:.:Linux:2.6 (Generic dfrag+)
10.0.0.4,0,80,6,CLIENT,[unknown:@www],0,1304497810
10.0.0.4,0,34935,6,FIN,[432:64:1:52:N,N,T:ATFN:unknown:unknown:uptime:2512hrs],0,1304497810
10.0.1.2,0,22,6,SERVER,[unknown:@ssh],1,1304497813
10.0.1.2,0,1790,6,SYN,[S4:63:1:60:M1460,S,T,N,W3:.:unknown:unknown
10.0.1.2,0,80,6,CLIENT,[unknown:@www],1,1304497814
10.0.1.2,0,1790,6,FIN,[864:63:1:52:N,N,T:ATFN:unknown:unknown:uptime:2512hrs],1,1304497814
```

Looking for an unknown signature we find several. The signatures that will be added is SYN signatures for the ip address 10.0.1.2 and 10.0.2.2, which is the same. And for the ip address 10.0.0.2.

Below is the new signatures made to be able to do the detection in the virtual network. The signatures were taken from the first output from prads. Adding them, together with what is known about the operating system in the virtual network. When collecting a new signature from an unknown list, it's important that not every field can be copied literally when adding the new signature. The initial TTL has to be checked in the documentation of the operating system and is not to be copied directly from the detection application. In the prads signature file it says that you should always wildcard the maximum segment size.

```
# New signatures
S4:63:1:60:M*:.:Linux:2.6.18_6_Xen_amd64
S4:64:1:60:M*,S,T,N,W5:.:Linux:Ubuntu 8.04
```

The signatures in general are explained in detail earlier in the thesis. More specific about these two signatures is that they don't have all the signature fields set. Common for the signatures is that both have windows size, initial TTL, don't fragment bit and the SYN packet size. The first signature has an maximum segment size as a wild card meaning that it can contain any value. The second signature has several options set like maximum segment size, selective ACK OK, NOP option and the window scaling option.

After adding the new signatures, the two operating system added in the signature file is detected. Both the Ubuntu 8.04 and the Xen_amd64.

```
asset,vlan,port,proto,service,[service-info],distance,discovered
10.0.0.2,0,22,6,SERVER,[unknown:@ssh],0,1304497807
10.0.0.2,0,55835,6,SYN,[S4:64:1:60:M1460,S,T,N,W5:.:Linux:Ubuntu 8.04
10.0.0.1,0,80,6,SYNACK,[5792:64:1:60:M1460,S,T,N,W5:ZAT:Linux:2.6 (newer, 5)
```

```
10.0.0.2,0,80,6,CLIENT,[unknown:@www],0,1304497807
10.0.0.1,0,80,6,SERVER,[unknown:@www],0,1304497807
10.0.0.1,0,80,6,FIN,[215:64:1:52:N,N,T:ATFN:unknown:unknown:uptime:2512hrs],0,1304497807
10.0.0.2,0,55835,6,FIN,[216:64:1:52:N,N,T:ATFN:unknown:unknown:uptime:2512hrs],0,1304497807
10.0.2.2,0,22,6,SERVER,[unknown:@ssh],1,1304497808
10.0.2.2,0,4548,6,SYN,[S4:63:1:60:M1460,S,T,N,W3:.:Linux:2.6.18_6_Xen_amd64
10.0.2.2,0,80,6,CLIENT,[unknown:@www],1,1304497808
10.0.2.2,0,4548,6,FIN,[864:63:1:52:N,N,T:ATFN:unknown:unknown:uptime:2512hrs],1,1304497808
10.0.0.4,0,22,6,SERVER,[unknown:@ssh],0,1304497810
10.0.0.4,0,34935,6,SYN,[S4:64:1:60:M1460,S,T,N,W4:.:Linux:2.6 (Generic dfrag+)
10.0.0.4,0,80,6,CLIENT,[unknown:@www],0,1304497810
10.0.0.4,0,34935,6,FIN,[432:64:1:52:N,N,T:ATFN:unknown:unknown:uptime:2512hrs],0,1304497810
10.0.1.2,0,22,6,SERVER,[unknown:@ssh],1,1304497813
10.0.1.2,0,1790,6,SYN,[S4:63:1:60:M1460,S,T,N,W3:.:Linux:2.6.18_6_Xen_amd64
10.0.1.2,0,80,6,CLIENT,[unknown:@www],1,1304497814
10.0.1.2,0,1790,6,FIN,[864:63:1:52:N,N,T:ATFN:unknown:unknown:uptime:2512hrs],1,1304497814
```

When adding signatures, it's important to know that prads checks the signature from the top to the bottom. This means that if a packet could hit two signatures, only the first signature would matter. This is important to know when adding signatures. It could be wise to add the most known signatures in you network, at the top of the signature list.

Adding signatures is straight-forward, but when adding signatures it's important to test the signature afterwards. If using to many wildcards, several different operating systems could hit the same signature.

An improvement to prads would be to create a better way to add signatures. The signatures is the main core of the application, so it's important that these will have quality of assurance. Also having the ability to share signatures among system administrators could help the signature database to expand. Maybe through an Internet community. Looking at p0f it has an Internet site where people can add their signature. But it looks like that this doesn't work as planned. The problem is that the author must add new signature files all the time. This is not being done.

Chapter 5

Discussion and conclusions

In the discussion and conclusion chapter are the results, the process, and the design discussed. In the discussion and conclusion, one tries to look at the approach in a bigger perspective.

5.1 Discussion

Two passive operating system detection tools have been compared. Both applications were compared in the 128.39.73.0 network and the results were observed and run through self-made Perl scripts that collected important statistics about the operating system detection.

The results given from the applications describes the applications and how they work. The results also describes what operating systems that is supposed to exist in the 128.39.73.0 network.

Both p0f and prads is command-line based applications created to be run in a Linux environment. When being run default, p0f prints the detection directly to screen, in opposite to prads that creates a log-file that is written to. Having the possibility to print directly to the screen should have also been included in prads. The statistical output prads gives after each run is a good addition. This helps the user in analysing the detection. This way one knows the amount of packets going through the application.

When investigating the output from the applications one realises that it's difficult to compare the operating system detection. The "problem" with p0f is that it is "greedy". Meaning that it does operating system detection on as many packets as it can. Looking at the p0f output it has as many as 652288 lines in contrast to prads that only has 3296 lines in it's output. Both run on the same Tcpdump! Even though p0f has at most three lines on each detection, p0f has much more lines in it's output. Why is this?

The main reason for this is because prads is "smarter" than p0f in the sense that if one operating system is detected for an ip address, for that ip address, prads does not detect this operating system more than once. This is a truth with modifications. When running the detection over a larger amount of time, prads tends to detect the same operating system over again. Also if the detected operating system has differences in the signature, it tends to be detected

several times. Small differences in the signatures, does not always mean that it is another operating system. In the signature file, this has been handled by adding a wildcard instead of the concrete signature.

At the same time as p0f gives much more lines than prads, p0f is much faster than prads when running the detection. After testing for some time, overall prads uses much more time in it's operating system detection. Testing prads with no asset detection and no other method than SYN, the time decreases, but not significantly. This is proved by looking at results from the prads -XS run.

Another thought to why prads uses more time is that it has a longer signature file. But this is not the case either. To figure out why prads uses much more time than p0f, looking at the source code itself could give some clues, but this is not done due to time constraints.

The testing run in this thesis has always been on a tcpdump. Running it without this tcpdump, and instead in a live network situation could be interesting. This way one could have run a test that checked the cpu, hard drive and memory usage when the applications is running. Comparing the application this way, one could have determined which of the applications that is the most resource hungry. This could be done, expanding the thesis further in the future.

Looking at the detection, prads has detected many more operating systems than p0f has done, even if it is for the same ip address. Why is this? Is this because prads tests more TCP states? When running prads with -XS it looks like the detections is much the same, so the different tests prads do, must have an impact of total detected operating systems. But should running in the different modes affect the detection? Of course, this is why prads test SYN, SYNACK and RST, not only SYN as p0f does.

The most obvious reason is that for every ip address there can be several operating systems. In the 128.39.73.0 network, many of the ip address contain two computers. A Linux and a Windows computer. So this is a obvious reason, but this is not the only reason for detecting many more operating systems than ip addresses.

What if the same computer is detected several times, under different names? This would be a big fault in the application. This leads to the question that is, is the detection correct every time? From the results chapter it's tempting to say that many of the detections is incorrect. Or if we look at the number detected operating systems on each ip address, this is way to many, mostly when running prads. But saying that the detection is wrong, is a bold statement. The operating system in itself is the correct operating system, the problem is that there are to many of each operating system.

Looking closer at the detection and the signature files, it's obvious that the same operating system can be named differently from one signature file to another. This way the script and the detection counts the wrong number of operating systems and lists the same operating system several times for an ip address in the detection. Even if these operating systems is counted wrong and the number is wrong, the detection is correct, and this is maybe what is most important. Knowing which operating systems that is located in your network

is the main priority.

A solution to this problem, could be to do a better job in the quality of assurance of these signature files. New signatures should have some kind of quality checking. Matching the signature files to each other could also be a way to secure that the same operating system is named the same in the different files.

When discussing signatures, another questions comes up. Are the signatures unique? The importance of each signature of being unique is important, if they are not, which operating system will be detected? The answer to this is actually answered earlier in the thesis. The detection has a top-to-bottom approach. The signatures placed early in the signature file will be hit first. But this don't mean that it is not important checking if the signature are unique. An improvement further to the application could be to have a another way of adding signatures. Not being allowed to add signature mere through a signature file, but maybe through a script that checks for different criteria when adding a new signature. This way the script could check for, for example, the uniqueness of the signature. It could also check if the operating system already exist in any of the other signature files.

The importance of data being collected in a thesis, being reliable, is a key element in making a good thesis. Throughout this thesis this has been a goal. The data collected should describe and be able to complement the findings in the thesis. The data collected from the schools network and the data collected in the virtual network are both complementing this thesis the way wanted. Of course doing more tests would further help in comparing the applications.

When it comes to results itself, the results were not surprising. That the applications always didn't do what they were supposed to do, was interesting. Since this has been an investigation starting from scratch, not having had any expectation to the results, no surprises were given.

For another student or researcher to reproduce these results, shouldn't offer any greater problems. Since the data tested in this thesis was collected in a live running network, the exact same data wouldn't be possible to reproduce, but collecting data in another network could do the same job.

The research in this field of computer security is not extensive. Doing research in the start of this thesis, no one else have done something similar to this thesis, and being able to find someone confirming or contradicting these findings is difficult.

When it comes to the process of the thesis some obstacles were met. At first, starting off with trying some alternative approaches on how to solve the thesis. First trying to install several operating systems on a desktop machine. Experiencing that only three operating systems could be installed on the same computer. Secondly trying to experiment with Honeyd as operating systems. Using Honeyd one should be able to simulate different operating systems, but this plan didn't work at all. Making Honeyd work took a long time, and finally getting it to work, trying to do detection on Honeyd didn't work at all.

Designing and performing an approach can be a little bit tricky. Starting off a thesis one has a thought on how to perform the following approach. When designing the approach one can believe that this is the right way to do it. But

when performing the approach one can realise that it does not always work as intended. In retrospect comparing p0f and prads in itself is maybe a wrong approach in investigating passive operating system detection. These two applications do fingerprinting in almost the same way, and will therefore have much of the same results. When comparing the applications, looking deeper at the performance and the usability of the applications could have been done. Also having a closer conversation with the developers of both applications, especially Edward Fjellskål the author of prads, could help me further in investigating and improving the applications. Comparing passive with active operating system detection could also be interesting and should maybe have been done in this thesis. Also looking at asset detection with prads could have been interesting but due to time constrains this wouldn't be possible.

Looking at the big picture, what has this thesis given the system administrators? First off is the knowledge about both p0f and prads and passive operating system detection. Not much is written in this subject and therefore by creating this thesis, it will help system administrators and researches to look at passive operating system detection. Also coming up with ideas and prototypes of how to further improve passive operating system detection has been a running theme through out this thesis.

5.1.1 Future work

When doing a thesis like this, along the way new ideas tend to emerge. Also knowing that there is not time enough to pursue these ideas, they have to been worked with in the future.

These days, virtual computing is more used than ever. Further testing of passive detection in an virtual environment could be a good idea. Testing if the virtual computers have other signatures than the physical ones, or if this actually has no impact on the detection at all.

An idea of developing prads further, is to develop a web interface or a GUI for the application. This way one could click on for example an operating system and the ip addresses having this operating system could pop up. Also using the information to set of alarms. Meaning that if prads finds an unpatched operating system in your network, this would set of an alarm, telling the system administrator to fix it.

An natural development to a passive operating system detection system would be to complement the IDS in the network. Giving the IDS all possible information concerning the network.

Also looking closer at obfuscators and ways to trick the detection could be interesting.

5.2 Conclusions

In this master thesis passive operating system detection was investigated. The main problem statements in this investigation were as follows:

- Investigate which passive operating system detection systems that exist today
- Compare p0f and prads in the 128.39.73.0 network.
- Discover the weaknesses in existing fingerprinting methods.
- Investigate which improvements that can be done to make operating system detection a helpful tool for network administrators.

The main way of answering these problem statements was to investigate passive operating system detection by running and testing some passive operating system detection applications. By testing these applications, one gets the knowledge of how they work and the difference and similarities in the detection applications.

Looking back at the discussion and the result, all this questions have been answered in some way. This master thesis looks at which applications that exist today. It takes a look at many of todays passive operating system detection applications, and goes deep into two of the applications. p0f and prads are tested by running them and also adding new signatures in prads. Both applications are tested by running them in small and a bigger network.

A comparison is done of p0f and prads in the 128.39.73.0 network, by looking at the performance, the output and the way p0f and prads works, in the 128.39.73.0 network. prads and p0f have a lot in common when it comes to detection, but the performance of p0f looks to be extremely faster than prads. prads on the other hand has the ability to run detection not only in SYN mode, but also on SYNACK and RST mode. In addition prads can do asset detection, being able to detect the services run on the operating systems. Even though prads uses a lot more time than p0f when running on the tcpdump collected through a day, we can conclude that since it uses only about five minutes on a file collected through a day the resources it demands running a live detection are minor.

Some weaknesses in the passive operating system detection methods are discovered by running p0f and prads and comparing the output with the knowledge we possess about the 128.39.73.0 network. The most obvious weaknesses are that p0f runs detection on every packet collected. prads does detect too many operating system due to prads ”messy” signature files. Also both applications lacks good ways to present their detection.

Improvements are suggested and tried implemented. Also in future work several more improvements are mentioned for other to work on further. The weaknesses and improvements are directly related to each other. The weakness that deals with the lack of good ways to present the detection has been a focus in improving the application. A prototype script has been made to

present the data in a better manner. Creating new signatures are also a way of improving the application so the application can be a helpful tool for network administrators. A lot of more ideas about improving the applications are also mentioned in the future work and will be interesting working on in the future.

Bibliography

[1] Derek Manky. Top 10 vulnerabilities inside the network. http://www.networkworld.com/news/tech/2010/110810-network-vulnerabilities.html.

[2] Michal Zalewski. *Silence on the Wire : A Field Guide to Passive Reconnaissance and Indirect Attacks*. No Starch Press, Incorporated, 2004.

[3] Neal Krawetz. *Introduction to Network Security*. Course Technolgy, 2006.

[4] Parker et al. *Cyber Adversary Characterization : Auditing the Hacker Mind*. Syngress Publishing, 2004.

[5] Redhat. Red hat documentation, chapter 1. security overview. http://docs.redhat.com/docs.

[6] Bhavya Daya. Network security: History, importance, and future. *University of Florida Department of Electrical and Computer Engineering*, unknown.

[7] V. Preetham, editor. *Internet Security and Firewalls*. Course Technolgy, 2002.

[8] Stanger et al. *Hack Proofing Linux : A Guide to Open Source Security*. Syngress Publishing, 2001.

[9] McClure et al. *Hacking Exposed (5th Edition)*. McGraw-Hill Professional Publishing, 2005.

[10] Security Wizard. Active os fingerprinting tools. http://www.securitywizardry.com/index.php/products/scanning-products/active-fingerprinters.html, 2010.

[11] L. Spitzner. Passive fingerprinting. *May*, 3:1–4, 2003.

[12] Michal Zalewski. P0f. http://lcamtuf.coredump.cx/p0f.shtml.

[13] Rfc: 791. http://www.ietf.org/rfc/rfc0791.txt, 1981.

[14] Matt Shelton. Pads. http://passive.sourceforge.net/about.php.

[15] Edward Fjellskål. Prads homepage. http://gamelinux.github.com/prads/.

[16] Alberto Ornaghi. Ettercap. http://ettercap.sourceforge.net/.

[17] Hjelmvik. Networkminer homepage. http://networkminer.sourceforge.net/.

[18] Gordon "Fyodor" Lyon. Nmap reference guide. http://nmap.org/.

[19] Eric Kollmann. Satori homepage. http://myweb.cableone.net/xnih/.

[20] SinFP. Sinfp - an overview. http://www.gomor.org/bin/view/Sinfp/WebHome.

[21] Gregory Lajon. Intrusion detection faq: What is xprobe? http://www.sans.org/security-resources/idfaq/xprobe.php.

[22] Gaël Roualland and Jean-Marc Saffroy. Ip personality. http://ippersonality.sourceforge.net/doc/ippersonality-en.html.

[23] Pablo Neira Ayuso. Netfilter homepage. http://www.netfilter.org/.

[24] Adrian Crenshaw. Hompage osfuscate. http://www.irongeek.com/i.php?page=security/osfuscate-change-your-windows-os-tcp-ip-fingerprint-to-confuse-p0f-networkminer-ettercap-nmap-and-other-os-detection-tools.

[25] et al. Ken Potter. *OS X for Hackers at Heart*. Syngress Publishing, 2005.

[26] Niels Provos. libevent - an event notification library. http://monkey.org/ provos/libevent/.

[27] Dug Song. libdumbnet homepage. http://code.google.com/p/libdnet/.

[28] Stanislav Malyshev. Libedit. http://libedit.sourceforge.net/.

[29] Greg Roelofs. zlib homepage. http://zlib.net/.

[30] Homepage tcpdump. http://www.tcpdump.org/.

[31] et al. Orebaugh. *Ethereal Packet Sniffing*. Syngress Publishing, 2004.

[32] L. Spitzner et al. Know your enemy. *Parts I, II, III." Available online: www. linuxnewbie. org/nhf/intel/security/enemy. html.(For Parts II and III, replace" enemy" with" enemy2" and" enemy3," respectively.)*, 2001.

[33] G.A. Marin. Network security basics. *Security Privacy, IEEE*, 3(6):68 – 72, 2005.

[34] S. Kalia and M. Singh. Masking approach to secure systems from operating system fingerprinting. In *TENCON 2005 2005 IEEE Region 10*, pages 1 –6, no, November 2005.

[35] G. Taleck. Ambiguity resolution via passive os fingerprinting. In *Recent Advances in Intrusion Detection*, pages 192–206. Springer, 2003.

[36] Tcpdump man page. http://www.tcpdump.org/tcpdump_man.html.

[37] William Stearns. p0f man page. http://linux.die.net/man/1/p0f.

[38] Edward Fjellskål. prads man page. https://github.com/gamelinux/prads/blob/master/doc/prads.man.

[39] David Keppel. Time(1) man page. http://linux.die.net/man/1/time.

[40] grep man page. http://unixhelp.ed.ac.uk/CGI/man-cgi?grep.

List of Figures

Chapter 6

Appendix

6.1 oscounter_p0f.pl

```
#Script that counts the instances of ip addresses and
#operating systems for p0f files.

#! /usr/bin/Perl
use Getopt::Std;
use strict;
use warnings;

my $opt_string = 'f:o:';
getopts("$opt_string",\my %opt) or exit 1;

# Declare variables
my $file = "$opt{'f'}";
my $line1;
my $line2;
my $cnt=0;
my %unique;
my $os = "empty";
# my $file1 = "$opt{'o'}";
my $ip = "empty";
# my $os = "";
my %HoH;
my %iphash;
my %nr;
my $ips;
chomp ($file);


# Open the file
open (FILE, "$file") || die "wrong filename";
```

```
while ($line1 = <FILE>)
{
    if($line1 =~ /^\<.*\> (128\.39\.73\.\d+):\d+ - (.*?\(.*?\))/ )
    {
        $os = "$2";
        $ip = "$1";
    }
     if($line1 =~ /^\<.*\> (128\.39\.73\.\d+):\d+ - (UNKNOWN \[.*\])/ )
      {
          $os = "$2";
          $ip = "$1";
      }

    $HoH{$os}{$ip} = "T";
    $iphash{$ip}{$os} = "T";
    $unique{$os}++;
    $nr{$ip} = "T";
}

# Close file
close FILE;
print "#### Number of detected operating systems on the specific ip address \n";
foreach my $ip (keys %iphash) {
        my $n = 0;

        foreach my $os (keys %{$iphash{$ip}}) {

                        $n++;
                    }
        print "$ip $n\n";
    }
print "########## Number of each operating system ######################\n";
foreach my $os (keys %HoH) {
    my $n = 0;
    foreach my $ip (keys %{$HoH{$os}}) {

        $n++;
    }
    print "$os $n\n";
}

$ips = keys %nr;
```

6.2 oscounter_prads.pl

```
#Script that counts the instances of ip addresses and
#operating systems for p0f files.

#! /usr/bin/Perl
use Getopt::Std;
use strict;
use warnings;

my $opt_string = 'f:o:';
getopts("$opt_string",\my %opt) or exit 1;

# Declare variables
my $file = "$opt{'f'}";
my $line1;
my $line2;
my $cnt=0;
my %unique;
my $os = "empty";
# my $file1 = "$opt{'o'}";
my $ip = "empty";
# my $os = "";
petter@mln6:~/Tcpdump$ cat newoscounter_prads.pl
#! /usr/bin/Perl
use Getopt::Std;
use strict;
use warnings;

my $opt_string = 'f:o:';
getopts("$opt_string",\my %opt) or exit 1;

my $file = "$opt{'f'}";
my $file1 = "$opt{'o'}" || die "Must enter a output file";
my $line1;
my $line2;
my $cnt=0;
my %unique;
my $os = "empty";
my $ip = "empty";
my %HoH;
my %nr;
my %iphash;
my $ips;
chomp ($file);

open (FILE, "$file") || die "wrong filename";

while ($line1 = <FILE>)
{

if($line1 =~ /^(128\.39\.73\.\d+),\d+,\d+,\d+,\w+,\[(.*?:.*?:.*?:.*?:.*?:.*?):(.*?):(.*?):.*\],\d+,\d+/ )
    {


        $os = "$3 $4";
        $ip = "$1";

    }
if($line1 =~ /^(128\.39\.73\.\d+),\d+,\d+,\d+,\w+,\[.*?:(.*?:.*?:.*?:.*?:.*?):(unknown):(unknown):.*\],\d+,\d+/ )
    {


        $os = "$3 $4 $2";
        $ip = "$1";
```

```
    }

    $HoH{$os}{$ip} = "T";
    $iphash{$ip}{$os} = "T";
    $unique{$os}++;
    $nr{$ip} = "T";

}

close FILE;
# open (FILE1, ">$file1");
# Count operatingsystems for ip address
print "########## Number of detected operating systems on the specific ip address ##########\n";
foreach my $ip (keys %iphash) {
    my $n = 0;

    foreach my $os (keys %{$iphash{$ip}}) {

            $n++;
        }
    print "$ip $n\n";
}

# Count instances of operating systems
print "########## Number of each operating system ###########################################\n";
foreach my $os (keys %HoH) {
    my $n = 0;

    foreach my $ip (keys %{$HoH{$os}}) {

            $n++;
        }
    print "$os $n\n";
}

$ips = keys %nr;
```