UNIVERSITY OF OSLO
Department of Informatics

Passive Asset Detection using NetFlow

Master thesis

Mats Erik Klepsland

February 14, 2012

Abstract

Computer networks are growing, making it difficult to keep track of all the hosts and services running on these hosts on the network. Using traditional methods like port scanning to detect hosts and services is cumbersome, host intrusive, slow and has to be performed continuously in order to be sufficiently updated.

In this thesis, we look at implementing a passive asset detection system using NetFlow. This will allow network administrators to detect hosts and services on the network using network traffic data that they already have collected. It also makes it possible to get a quick glimpse of the network state at a specific time that could be months or even years back in time, the only limitation being the amount of NetFlow data collected.

Unlike other passive asset detection systems, like PRADS, using NetFlow makes us able to handle network traffic speeds up to several Gbit/s, or even Tbit/s. This makes a passive asset detection system using NetFlow data highly scalable and because it is capable of processing a lot of data it also has a high detection rate.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisors, Bente Christine Aasgaard, Margrete Raaum and Professor Audun Jøsang, for their invaluable feedback and guidance. Without your help, I would never have been able to finish this work.

I would also like to thank all the people at PING, for providing a friendly working environment, making the process of writing this thesis more enjoyable.

Finally, I would like to thank my girlfriend Lena for her support and encouragement during my work on this thesis.

Thank you all so very, very much.

Mats Erik Klepsland
University of Oslo
February, 2012

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Background and Motivation

With the growth in personal computers, network connected cell phones, and computer network equipment, it is getting more and more difficult to keep track of all the network assets on the company network, which potentially can lead to risk exposure and liability.

A network asset is a host on a network that is identified either by its MAC address or by the IP address it is using to communicate with other hosts on the network. A network asset can have several network services running, like for instance a web service or a domain name service (DNS).

As a network grows and contains thousands of hosts, the task of port scanning the entire network becomes a formidable task which can take days, and even weeks to perform. In addition, the scanning results get outdated fast, due to changes in the network services offered by the hosts. Port filtering is also a barrier to port scanning, and leaves the network administrators potentially blind to network services running on the network.

Passive asset detection solutions often consist of a single host situated centrally on the computer network, so that it sees all the network traffic that passes through. Because of this it can easily be exposed to gigabit network speeds. However, when the traffic amounts starts to exceed several gigabit or even terabit network speeds, it does not scale.

NetFlow, a technology initially developed by Cisco, is able to overcome

this limitation. It is implemented in most Cisco switches and routers which means that the NetFlow data is collected without having to add any additional equipment to the network. There are also other vendors offering NetFlow enabled equipment. However, because of the availability of Cisco equipment, we have chosen to focus on Cisco NetFlow in this thesis.

Since we deal with real traffic data and systems in this thesis, we have chosen to anonymize all the data presented. Therefore, all hosts presented have randomly assigned IP addresses and hostnames.

1.2 Goals and Problem Description

The main goal of this thesis is to develop a passive asset detection system that use data collected by NetFlow to detect assets on a computer network. The system should be scalable enough so it can handle huge computer networks, it must be capable of running in real-time, it must have a high detection rate, and the rate of false positives must be low.

In addition to detecting the hosts on the network, it must also detect the network services that each of the hosts are hosting, and the operating system that the host uses.

The system itself should be able to run on a NetFlow collector running Nfdump tools, an open source NetFlow collector. The whole system consists of four components:

- the main component parsing the NetFlow data to collect assets,
- a component to remove duplicates and store the assets in a database,
- a database to store the assets in, and
- a component to display the collected assets to the user in an orderly way.

These four components could be merged into one. However, the Unix philosophy states that all programs should only do one thing and do it well [1]. We choose to comply with these principles.

1.3 Outline

This thesis is divided into six chapters. Chapter 2 consists of background material relevant to this thesis. Chapter 3 presents a detailed description of the implementation we have done and the related challenges we have encountered. The chapter is divided into subsections for each of the components in our system. Chapter 4 presents an evaluation of our implementation. It shows how the system performs over time. Chapter 5 contains discussion of concepts relevant to this thesis. It also contains legal concerns and a discussion of advantages and disadvantages that we have encountered while implementing our system. Chapter 6 concludes the thesis and suggests future work. Source code of the system implemented for the thesis and extended technical documentation is provided in the Appendix.

Chapter 2

Background

In this chapter we provide the necessary background for the tools used in this thesis. Section 2.1 gives an overview of what NetFlow is, what it can be used for, and basic functionality. Section 2.2 through section 2.4 discuss different techniques for detecting network assets. The last part of the chapter discusses two tools used for asset detection, Nmap in section 2.5 and PRADS in section 2.6.

2.1 NetFlow

NetFlow is a technology initially created by Cisco for use in switches and routers running Cisco IOS, but it is now supported by a range of vendors [2] [3]. NetFlow provides the network administrators with the tools to understand how, why and where the traffic is flowing on their network. Seeing these flows makes it easier to troubleshoot network problems and gives the administrator an audit trail for each IP packet on the network.

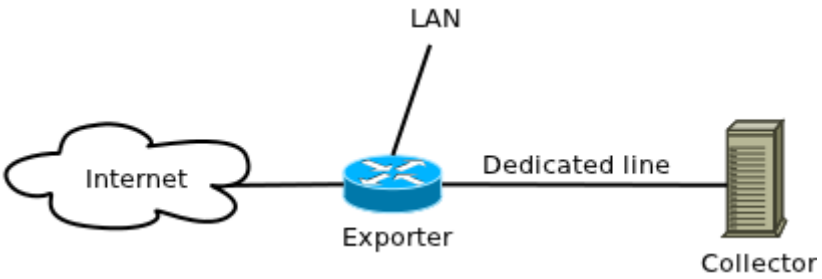

Figure 2.1: NetFlow overview

NetFlow can be used to track application and network usage and is often used by Internet service providers (ISP) to calculate how much network resources each customer is using. It can be used to get a quick glimpse of how changes made to the network infrastructure impact the network. In addition to this, NetFlow can also be used to detect network anomalies and security vulnerabilities. If an Internet worm is communicating on a specific port or is communicating in a certain pattern, one can often with high certainty detect the machines infected with the worm. This makes NetFlow a powerful tool when protecting the local network.

NetFlow consists of two main components, an exporter and a collector. The exporter can be a NetFlow enabled Cisco switch or router. This means that we do not have to add additional equipment to start collecting NetFlow data if the core network already consist of Cisco equipment. There are also several other vendors making NetFlow exporters, so the exporter do not necessarily have to be a Cisco switch or router. However, in this thesis we have mainly focused on Cisco NetFlow when mentioning equipment and technology regarding NetFlow, even though it would not have made much difference looking at other solutions.

The collector is a centralized host on the network that receives NetFlow data from all the exporters. Using a collector and thereby centralizing all the NetFlow data collected, eases the administration and gives a complete picture of all the traffic going through the network. It is possible to have both several exporters collecting and exporting data, and several collectors receiving data from the exporters on the network.

It is important to plan ahead when choosing where to place the collector on the network. The reason for this is that 1 - 5 % of the total network traffic flowing through the network is used to export NetFlow data to the collector. Therefore, it is best to place the exporters as close to the collector as possible to avoid utilizing too much bandwidth. This is also important to minimizing packet loss between the exporters and the collector, since this leads to gaps in the collected flow data.

2.1.1 IP Flows

Each packet that is forwarded within a NetFlow enabled switch or router is examined for a set of IP packet attributes. These attributes are used as the IP packet's identity, or fingerprint of the packet, to determine if the packet

is unique or similar to other packets. This enables recognition of duplicate packets. The IP packet attributes that are inspected by NetFlow can be seen in table 2.1. The attributes in the table are IPv4 attributes. IPv6 is only supported by NetFlow version 9.

Attribute	**Description**
IP source address	IP address of the sending host
IP destination address	IP address of the receiving host
Source port	Port used by the sending host
Destination port	Port used by the receiving host
Layer 3 protocol type	Protocol used (TCP/UDP/ICMP)
Class of service	Priority value that can be used by QOS
Router or switch interface	Interface used on the device

Table 2.1: IP packet attributes used by NetFlow

All packets with the same source and destination IP address, source and destination ports, protocol interface, and class of service are grouped into a flow. The number of packets and bytes are then added together to display the total amount of traffic in that specific flow. Grouping IP packets together into flows makes NetFlow highly scalable, since it greatly reduces the amount of traffic data that needs to be stored. However, it also leaves the network administrators with less information to work with. All the flows are stored in the NetFlow cache on the exporter before being transferred to the collector.

Additional information are also added to the flows, like flow timestamps. These timestamps are useful for calculating the number of packets and bytes per second. Next hop IP addresses and subnet mask for the source and destination addresses are added. The subnet masks are added to calculate prefixes. TCP flags are also added. It is important to remember that because all the network traffic is grouped together, so are the TCP flags, which means that it is not possible to see the TCP flags for individual packets.

All flows are unidirectional, meaning that a flow only consists of packets flowing in one direction. Because of this a TCP connection between Bob and Alice would create two flows, one flow flowing from Bob to Alice, and another flow flowing from Alice to Bob. As illustrated in figure 2.2 network traffic can consist of several IP packets being sent back and forth between two hosts, but NetFlow groups all the IP packets together, creating one flow per direction. The same would apply for a UDP connection.

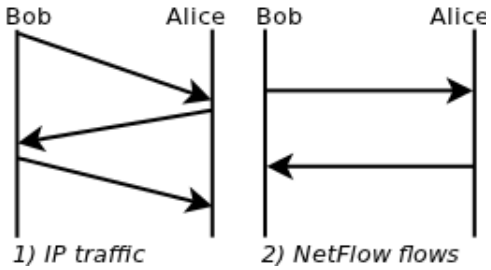

Figure 2.2: IP traffic versus NetFlow flows

2.1.2 Exporter

The exporter can be a Cisco device running Cisco IOS with NetFlow enabled. This is typically a Cisco switch or router. Most new Cisco devices are capable of capturing NetFlow data. See table 2.2 for an overview of Cisco devices supporting NetFlow. The exporter examines all IP packets being forwarded through the device and store the NetFlow data in the NetFlow cache.

Device	Supported
Cisco 800, 1700, 2600	Yes
Cisco 1800, 2800, 3800	Yes
Cisco 4500	Yes
Cisco 6500	Yes
Cisco 7200, 7300, 7500	Yes
Cisco 7600	Yes
Cisco 10000, 12000, CRS-1	Yes
Cisco 2900, 3500, 3660, 3750	No

Table 2.2: Cisco switches and routers supporting NetFlow

It is possible to examine the NetFlow data directly on the exporter, using the Cisco device's command line interface (CLI). This is useful for getting an immediate view of what is happening in the network. It is also very useful for troubleshooting. However, using a collector is vital for getting a better overview, and facilitate in-depth analysis.

To be able to use a centralized collector the flows must be exported from the device collecting the flows. The device determines which flows to export to the NetFlow collector by looking at the state of the flows. If a flow has been inactive for 15 seconds, it is exported. A flow is also exported if the flow duration exceeds the active timer. The default active timer is 30 minutes,

meaning that a flow is split up and exported if it is active longer than this. Large file transfers will typically exceed the active timer. It may be broken into multiple flows by the exporter and combined again by the collector. A flow is also exported at termination (e.g. FIN or RST flag). When exporting flows approximately 30 to 50 flows are bundled together and transported using UDP to the NetFlow collector. Because UDP is used for this purpose, loosing a packet would leave a gap in the collected NetFlow data.

2.1.3 Collector

The NetFlow collector receives flows from one or more NetFlow exporters. It processes the flows by parsing and storing them. The collector provides the network administrators with a centralized overview of all the flows on the network. This increases network visibility and facilitates troubleshooting.

There exists a range of different NetFlow collectors. These include Cisco, open source solutions and third party products that collect and present NetFlow data. It is important to choose a solution that is suitable for the network that is being monitored. Some systems even offer a two-tier architecture, where collectors are placed in key points in the network as probes and data is forwarded to a main reporting server. Other systems consists of only a single server for collecting and reporting.

The operating system used by the various collectors varies. Everything from Linux and BSD to Windows is used. Price is also an important factor. The different solutions vary greatly in price, while some of the solutions are free, the expensive solutions cost more than USD 25,000.

2.1.4 Export Formats

NetFlow can use different export formats when exporting packets from the exporter to the collector. These are commonly called the export version. The export versions include version 5, 7 and 9. Version 5 is the most common format used, while version 9 is the latest format and has advantages when it comes to traffic analysis, security, support for IPv6 and multicast.

Bytes	Content	Description
0 - 1	version	NetFlow export format version number
2 - 3	count	Number of flows exported in this packet (1-30)
4 - 7	SysUptime	Current time in milliseconds since router booted
8 - 11	unix_secs	Current seconds since 0000 UTC 1970
12 - 15	unix_nsecs	Residual nanoseconds since 0000 UTC 1970
16 - 19	flow_sequence	Sequence counter of total flows seen
20 - 21	engine_type	Type of flow switching engine
21 - 23	engine_id	Slot number of the flow switching engine

Table 2.3: NetFlow version 5 header format

The NetFlow flows consists of a flow header format and a flow record format [4]. The version 5 header format consists of a version number, number of flows exported, system uptime of the exporter, the current time in UNIX time (both seconds and nanoseconds), and information about the flow engine. See table 2.3 for the entire version 5 header format.

The version 5 and 7 header formats are quite similar to each other. The only difference is that the version 7 header format does not include information about the flow switching engine and it has three unused bytes (reserved).

While the flow header format contains information about the entire bundle of flows, the flow record format contains the individual flows. The version 5 flow record format contains information like source IP address, destination IP address, transport protocol used (e.g. TCP or UDP), source port number, destination port number, and TCP flags. See table 2.4 for the entire version 5 flow record format.

The version 5 and 7 flow record formats are almost identical to each other. One of the few differences is that the version 7 format includes a source router field. The version 7 record format is only used on Cisco Catalyst switches, but it has very limited support and few switches actually support this export format. It is therefore safer to use the version 5 export format instead.

The NetFlow version 9 export format differentiates from version 5 and 7 by being template based [2]. By using templates it allows future enhancements to NetFlow services, without requiring changes to the flow record format. This makes it simple to add new features to NetFlow more quickly without breaking current implementations.

Bytes	Content	Description
0 - 3	srcaddr	Source IP address
4 - 7	dstaddr	Destination IP address
8 - 11	nexthop	Next hop router's IP address
12 - 13	input	Ingress interface SNMP ifIndex
14 - 15	output	Egress interface SNMP ifIndex
16 - 19	dPkts	Packets in the flow
20 - 23	dOctects	Octets (bytes) in the flow
24 - 27	first	SysUptime at the start of the flow (ms)
28 - 31	last	SysUptime at the time of the last packet of the flow was received (ms)
32 - 33	srcport	Layer 4 source port number or equivalent
34 - 35	dstport	Layer 4 destination port number or equivalent
36	pad1	Unused (zero) byte
37	tcp_flags	Cumulative OR of TCP flags
38	prot	Layer 4 protocol (for example, 6=TCP, 17=UDP)
39	tos	IP type-of-service byte
40 - 41	src_as	Autonomous system number of the source, either origin or peer
42 - 43	dst_as	Autonomous system number of the destination, either origin or peer
44 - 45	src_mask	Source address prefix mask bits
46 - 47	dst_mask	Destination address prefix mask bits
48	pad2	Pad 2

Table 2.4: NetFlow version 5 flow record format

The version 9 export format consists of a flow header format followed by at least one or more template or data FlowSets. A template FlowSet provides a description of the fields that will occur in future data FlowSets. A FlowSet template must be sent before the FlowSet data for the collector to be able to understand which data fields the FlowSet data consists of. The FlowSet data could either be sent within the same export packet as the FlowSet template, or it could be sent later on. The collector must always cache any received templates, and examine the template cache to figure out which template a data set belongs to.

The version 9 NetFlow packet header is based on the version 5 NetFlow packet header, thus being almost identical to its predecessor. However, fields like sampling interval and aggregation scheme has been left out. These are instead sent in another data record, called the option template. The option template is used to supply meta data about the NetFlow process, like information about IP flows.

To export NetFlow data using the version 9 export format FlowSet templates are used. FlowSet templates consists of a FlowSet ID, length, template ID, field count, field type, and field length. The FlowSet ID is used to separate a template from a data record by setting the FlowSet ID to zero. The length refers to the total length of the FlowSet. The template ID is a unique ID given to each FlowSet template. The field count is the number of fields in this template. The field type and field length refers to one of the fields in the FlowSet template. See table 2.5 for all the fields used in the FlowSet template.

Field Name	**Value**
FlowSet ID	The FlowSet ID is used to distinguish template records from data records. A template record has a FlowSet ID of zero.
Length	Length refers to the total length of this FlowSet.
Template ID	Template ID is a unique value given to each template FlowSet to match the type of NetFlow data it will be exporting.
Field Count	This field gives the number of fields in this template record.
Field Type	The NetFlow data field to be exported.
Field Length	The length of the field defined above.

Table 2.5: NetFlow version 9 FlowSet Template

The possible data fields that can be exported using NetFlow version 9 are almost unlimited. New data fields can easily be added. This makes NetFlow version 9 more versatile than version 5. See appendix D for a list of data fields implemented for NetFlow version 9.

2.1.5 Security Concerns

It is important to not only survey the benefits of a technology, but also the drawbacks, when deploying it on your network. NetFlow was designed with the expectation that the exporter and collector would reside within a single private network [3]. However, this is often not the case. In many cases NetFlow data is sent over the Internet.

NetFlow does not deploy any form of cryptography when sending data between the exporter and the collector. Because of this, all the NetFlow data is sent in clear text, opening up for several attack vectors.

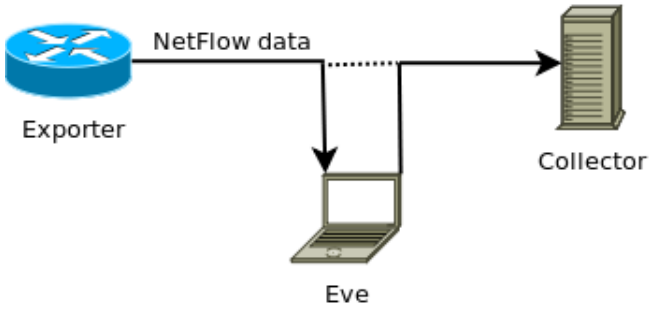

Figure 2.3: Eavesdropping on NetFlow traffic

An attacker may eavesdrop on the traffic, if she can access the NetFlow export network. This can give the attacker information about the active flows in the network, traffic patterns and communication endpoints. This information could be used to plan further attacks and to spy on user behaviour. The effectiveness of this attack depends on the kinds of data that is being reported. If a flow record contains both source and destination IP addresses it might reveal sensitive information about the user activity. However, if it only contains the source and destination IP network it would be less revealing about user activity, but it could reveal sensitive or classified information about business relationships or communication partners.

NetFlow does not deploy any form of integrity checking. Because of this, it is possible to forge exported flow records. This could be used to prevent the detection of an attack, by altering the flow records on the path between the exporter and the collector, or it could be done by injecting forged flow records that pretend to be coming from the exporter. By doing this flow records for traffic that has not actually occurred on the network could be added to the collector.

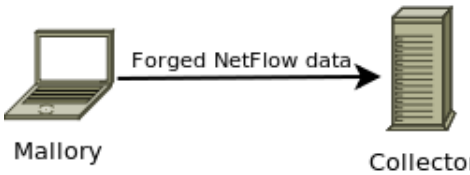

Figure 2.4: Injecting forged NetFlow traffic

If NetFlow version 9 is used then both templates, option templates, and FlowSet data could be forged and injected using this method.

It is also possible to launch a Denial of Service (DOS) attack against the collector, rendering it unable to capture NetFlow export packets. However, this is not a security threat that is specific to NetFlow.

2.1.6 Nfdump Tools

Nfdump tools is an open source NetFlow collector distributed under the BSD license [5]. It is designed to be able to analyze NetFlow data from the past as well as to continuously track interesting traffic patterns in real-time. The amount of historic NetFlow data it can store is only limited by disk space. Because of this, months or even years worth of flows can easily be stored. Nfdump support NetFlow version 5, 7 and 9 and consists of the following tools:

nfcapd NetFlow capture daemon. Reads the NetFlow data sent from the exporters and stores the data into files. Automatically rotates files every 5 minutes (default).

nfdump NetFlow dump. Reads NetFlow data from the files stored by nfcapd and displays it to the user. Filters can be used to limit the data.

nfprofile NetFlow profiler. Reads NetFlow data from the files stored by nfcapd and stores it in files based on specified filter sets.

nfreplay NetFlow replay. Reads the NetFlow data from the files stored by nfcapd and sends it over the network to another host. Filters can be applied to only send matching flows.

nfclean.pl Cleans up old data. Sample script to cleanup old data.

ft2nfdump Read and convert flow-tools data. Convert data from flow-tools format to nfdump format to be processed by nfdump. Flow-tools is a collection of programs used to collect, send and process NetFlow data [6].

Nfdump is able to read and store NetFlow data from several exporters at the same time. However, one nfcapd process is needed for each NetFlow stream. All the data stored by nfcapd is organized in a time based fashion. The output file is by default rotated and renamed every five minutes with the time stamp *nfcapd.YYYYMMddhhmm*. The file *nfcapd.201111201605* contains data from November 20th 2011 16:05. The total amount of nfcapd files per day are 288, one for each five minutes interval. All the data is stored to disk, before being analyzed. This separates the process of storing from the process of analyzing the data.

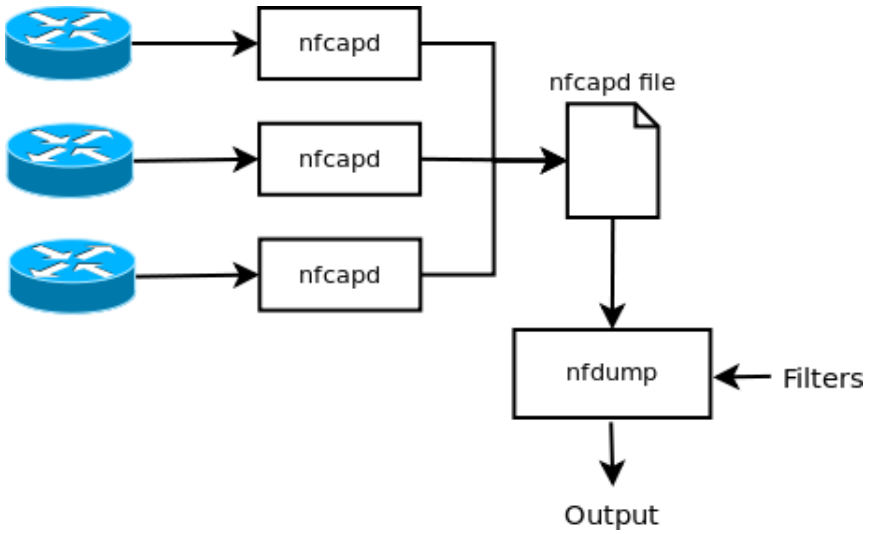

Figure 2.5: Nfdump overview

Nfdump tools deploy a filtering scheme with syntax similar to TCPdump. Because of this, it is simple for someone that already knows the filter syntax for TCPdump to use Nfdump. The tools are optimized for speed for efficient filtering. Processed flows can either be printed in text to the standard output stream (STDOUT) or written to a file. Example output from Nfdump can be seen in listing 2.1.

Listing 2.1: Example output from Nfdump

```
$ nfdump -r nfcapd.201111201605 'src ip 143.220.5.201 and dst port 53'

Date flow start  Duration Prot  Src IP Addr:Port  Dst IP Addr:Port  Packets  Bytes  Flows
2011-06-18 20:56:58.515  0.000 UDP  143.220.5.201:47349 -> 143.220.1.30:53  1  58  1
2011-06-18 20:56:58.515  0.000 UDP  143.220.5.201:39466 -> 143.220.1.30:53  1  58  1
2011-06-18 20:56:58.515  0.000 UDP  143.220.5.201:37828 -> 143.220.1.30:53  1  58  1
2011-06-18 20:56:58.515  0.000 UDP  143.220.5.201:59128 -> 143.220.1.30:53  1  58  1
2011-06-18 20:56:58.515  0.000 UDP  143.220.5.201:58275 -> 143.220.1.30:53  1  58  1
2011-06-18 20:56:58.515  0.000 UDP  143.220.5.201:44873 -> 143.220.1.30:53  1  58  1
2011-06-18 20:56:58.514  0.000 UDP  143.220.5.201:52625 -> 143.220.1.30:53  1  58  1
2011-06-18 20:56:58.518  0.000 UDP  143.220.5.201:58926 -> 143.220.1.30:53  1  58  1
[..]
2011-06-18 20:56:58.518  0.000 UDP  143.220.5.201:36574 -> 143.220.1.30:53  1  58  1
2011-06-18 20:56:58.454  0.000 UDP  143.220.5.201:39233 -> 143.220.1.30:53  1  58  1

Summary: total flows: 264, total bytes: 15520, total packets: 265, avg bps: 343,
avg pps: 0, avg bpp: 58
Time window: 2011-06-18 20:27:53 - 2011-06-18 21:04:55
Total flows processed: 213402, Blocks skipped: 0, Bytes read: 11097068
Sys: 0.035s flows/second: 5928986.2  Wall: 0.033s flows/second: 6278745.4
```

The example shows the use of Nfdump filters. In this case flows with the source IP address *143.220.5.201* and destination port 53 is filtered out and printed to the screen. This is the default output format used by Nfdump. It is also possible to use the output format *long* to get additional information about the flows such as TCP flags and Type of Service (TOS). If this is not enough output it is possible to use the output format *extended*. It provides even more information like packets per second, bits per second and bytes per packet.

Tag	Description	Tag	Description
%ts	Start time - first seen	%in	Input interface number
%te	End time - last seen	%out	Output interface number
%td	Duration	%pkt	Packets
%pr	Protocol	%byt	Bytes
%sa	Source address	%fl	Flows
%da	Destination address	%flg	TCP flags
%sap	Source address:port	%tos	Type of service
%dap	Destination address:port	%bps	Bits per second
%sp	Source port	%pps	Packets per second
%dp	Destination port	%bpp	Bytes per second
%sas	Source AS	%das	Destination AS

Table 2.6: Nfdump custom output pre defined element tags

Nfdump also supports the use of a custom output format allowing to specify what the output should look like. This is done using pre-defined element tags. See table 2.6 for a list of pre-defined element tags allowed by Nfdump.

By using the custom output format we can limit the output to only the fields we need. This makes it simple to use the output from Nfdump in other applications.

Listing 2.2: Custom output in Nfdump

```
$ nfdump -r nfcapd.201111201605 -o fmt:%sa%sp%da%dp 'src ip 143.220.5.201 and dst port 53'

Src IP          AddrSrc Pt  Dst IP         AddrDst Pt
143.220.5.201   47349       143.220.1.30   53
143.220.5.201   39466       143.220.1.30   53
143.220.5.201   37828       143.220.1.30   53
143.220.5.201   59128       143.220.1.30   53
143.220.5.201   58275       143.220.1.30   53
143.220.5.201   44873       143.220.1.30   53
143.220.5.201   52625       143.220.1.30   53
[..]
143.220.5.201   58926       143.220.1.30   53
143.220.5.201   36574       143.220.1.30   53

Summary: total flows: 264, total bytes: 15520, total packets: 265, avg bps: 343,
avg pps: 0, avg bpp: 58
Time window: 2011-06-18 20:27:53 - 2011-06-18 21:04:55
Total flows processed: 213402, Blocks skipped: 0, Bytes read: 11097068
Sys: 0.035s flows/second: 5928986.2  Wall: 0.033s flows/second: 6278745.4
```

Listing 2.2 shows the usage of the custom output format. It is specified using *-o fmt:* and then the elements we want to display. In this case the elements source IP address, source port, destination IP address and destination port are selected.

In addition to being able to specify output mode and filter flows, Nfdump is capable of printing flow statistics. It does this by aggregating flows based on the fields we want statistics for. This makes it possible to print the amount of traffic generated by each host on the network, or even the amount of traffic generated on each port. This makes it easy to see which hosts that generate the most traffic on the network, or to recognize the most common services on the network.

2.2 Host Detection Techniques

In the following sections we are going to explore several methods that are used for detecting hosts on a network. These methods are used by common tools like Nmap (mentioned in section 2.5), Nessus, PRADS (mentioned in section 2.6), OpenVAS as well as many others.

2.2.1 Ping Sweep

A ping sweep is a kind of network probe, where the intruder or network auditor sends a set of ICMP (Internet Control Message Protocol) echo packets to a range of IP addresses [7]. The goal of this is to see which hosts that respond to the probes sent. By doing this it is simple to determine the hosts that are alive and the hosts that are not. This could be compared to knocking on people's doors to look for indication of the presence of life.

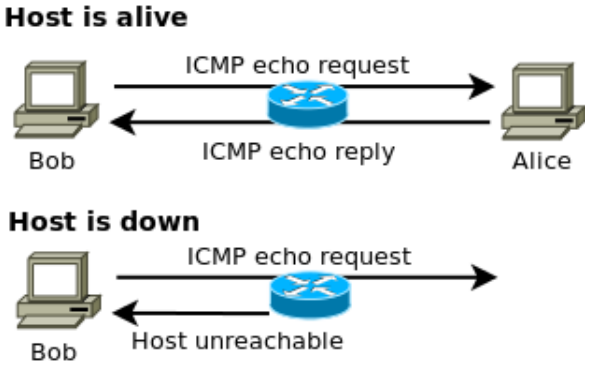

Figure 2.6: Ping Sweep

This is done by sending ICMP echo request (ICMP type 8, code 0) to a set of hosts [8]. The hosts that sends an ICMP echo reply back (ICMP type 0, code 0) are available on the network. Unfortunately, there are a lot of network administrators that do not comply with the ICMP RFC (RFC 792) by turning off ICMP echo packets on their networks, to make their machines less visible. This is a basic method to thwart host detection, but it makes it cumbersome to use Ping sweeps for host detection.

2.2.2 TCP SYN Ping

TCP SYN Ping is done by sending an empty TCP packet with the SYN flag set to a host to check if it is alive [9]. The destination port could be any port, but it is recommended to use port 80, since this is one of the most common ports used on the Internet and it is therefore more likely that this port is not blocked. This method picks up hosts not detected by ping sweep (section 2.2.1), where ICMP is either turned off or blocked.

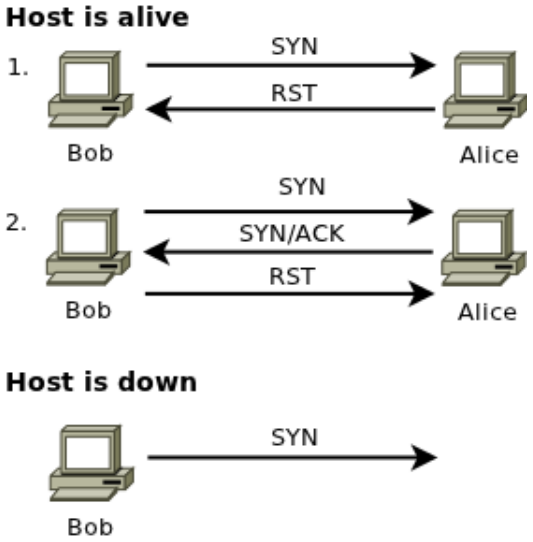

Figure 2.7: TCP SYN Ping

By setting the SYN flag we indicate that we want to establish a connection to the remote host. The remote host then responds with a RST (reset) packet if the destination port is closed. This happens in most cases. Otherwise, if the port is open the host responds with continuing the three way handshake by sending a SYN/ACK packet back. If this is the case one would send a RST packet to the host closing the connection rather than completing the TCP handshake by sending an ACK packet back. The reason to do this is to tell the remote host that the TCP connection should be terminated, to avoid it believing that the packet has been dropped and continue resending the packet.

As long as we get a response back from the host, it does not really matter what kind of answer, since our main objective is to check if the host is alive or not.

2.2.3 TCP ACK Ping

This technique is similar to the SYN ping (section 2.2.2). However, in this case a TCP ACK flag is set instead of a SYN flag. This makes it look like we are trying to acknowledge data over an already established TCP connection, even though such a connection does not exist. Because of the lack of an existing session, remote hosts will always respond with a RST packet, hence revealing their mere existence.

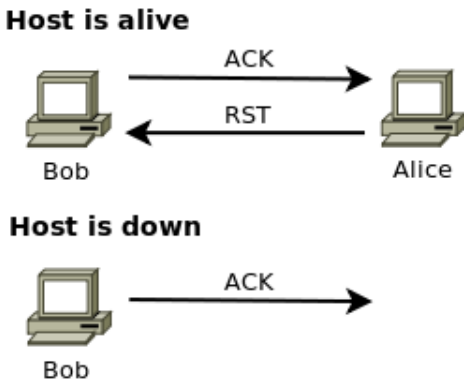

Figure 2.8: TCP ACK Ping

The reason for using this technique in combination with TCP SYN ping is to increase the possibility to bypass firewalls. Many network administrators configure their filtering routers or firewalls to block incoming SYN packets destined for anything but public services. By using the ACK probe, this kind of filtering can be circumvented. However, some firewalls keep track of the state of all connections and blocks all packets that are not part of established TCP connections. This method is therefore not sufficient to circumvent rules in stateful firewalls.

2.2.4 ARP Ping

Address Resolution Protocol (ARP) is used to glue together the IP and Ethernet networking layers [10]. It is used to locate the Ethernet address associated with a desired IP address. When a host has a packet destined to another IP address it will broadcast an ARP request asking who has that IP address. The host with the requested IP address will then respond by sending a reply back telling the requesting host its Ethernet address. See figure 2.9 for an example of an ARP conversation.

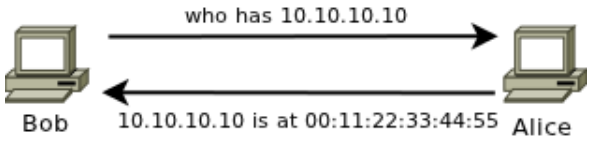

Figure 2.9: ARP Ping

The ARP protocol is fast and because of this, ARP replies usually comes within a couple of milliseconds [9]. Using IP scans like in the techniques mentioned above it could take as long as two seconds to scan each host. This is not a problem when only scanning a few hosts, but it becomes a huge problem when scanning large subnets such as a 16-bit subnet (class B) or even as big as a 8-bit subnet (class A). This makes ARP scanning a valuable scanning method when scanning large computer networks.

It is also possible to detect hosts by passively listen for ARP packets on the network since they are broadcasted to all hosts on the network. We do not detect hosts as quickly as when scanning when doing this, but it is a lot less intrusive. However, ARP packets does not pass through a router unless the router is set up with proxy ARP, hence this is useful on local networks only.

2.3 Service Detection Techniques

In the following sections we will present a set of techniques for detecting services running on hosts on the network.

2.3.1 TCP SYN Scan

TCP SYN scan is often called stealth scan, the reason being that it is not easy to detect since it never completes TCP connections [9]. TCP SYN scan is popular since it is quick, unobtrusive and stealthy. It can scan thousands of ports per second on a fast network.

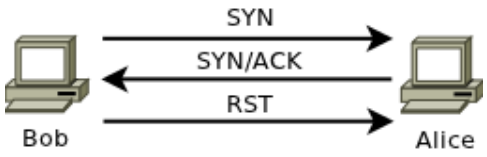

Figure 2.10: TCP SYN Scan

TCP SYN scan works similar to TCP SYN ping (mentioned in section 2.2.2) by sending a TCP packet with the SYN flag set. However, since we want to check if a port is open rather than if the host is alive, the port number selected have to be the port we want to check. If the port is open the remote host sends a SYN/ACK packet back, otherwise a RST packet is sent back. On receiving a SYN/ACK, the scanning host sends a RST packet to the remote host instead of completing the TCP handshake. If the scanner does not send a RST packet the remote host would assume the packet it sent was dropped and then just keep re-sending its SYN/ACK.

2.3.2 TCP Connect Scan

TCP connect scan is a scanning method that does not send raw packets, and therefore does not need as high system privileges as the previous scanning methods [9]. Raw packets bypass layers in the TCP/IP stack, and therefore needs high privileges to be sent [11]. TCP connect scan asks the operating system to establish a connection with the target host and port by issuing the *connect* system call. This system call is also used by web browsers and most other network-enabled applications when establishing a connection.

Instead of only opening TCP connections half way through, like the previous methods mentioned, this method completes the connections. Since it completes the connections it will take longer to perform this scan then the TCP SYN scan, and Unix systems will log the connection to syslog as suspicious when a connection is made and no data is sent. Because of this TCP connect scan is not considered a stealthy scanning method.

Figure 2.11: TCP Connect Scan

As shown in figure 2.11, this scan starts by initiating a TCP connection with the remote host. After a connection is established, the remote host sends the service banner of the service running on the specified port. Instead of sending data traffic, the connection is then closed down by sending an RST

packet to the host. In the figure the port is open. In cases were the port is closed, the remote host would not respond with a SYN/ACK packet and the TCP connection would not be established.

By grabbing the banner string of the service one can see not only what service is running, but also frequently the service version. This is useful when looking for hosts running vulnerable software versions.

2.3.3 UDP Scan

The scanning methods previously mentioned in section 2.3 are only capable of detecting TCP services [9]. Even though the most popular services on the Internet run over TCP, it is important to be able to also detect UDP services. Some UDP services are widely deployed, like DNS, SNMP, NFS and DHCP. Many security auditors ignore scanning UDP ports since UDP scanning is generally slower and more difficult to perform than TCP, and most commonly used tools do not provide efficient UDP scanning.

UDP scan works by sending an empty UDP header to each port on a remote host. A port is considered open if any UDP response is sent back from the target port. However, this is highly unlikely, since the listening application usually discards the probe as invalid, because it does not match the traffic it expects to get. If no response is returned the port is either open or filtered. If an ICMP port unreachable packet is returned the port is closed.

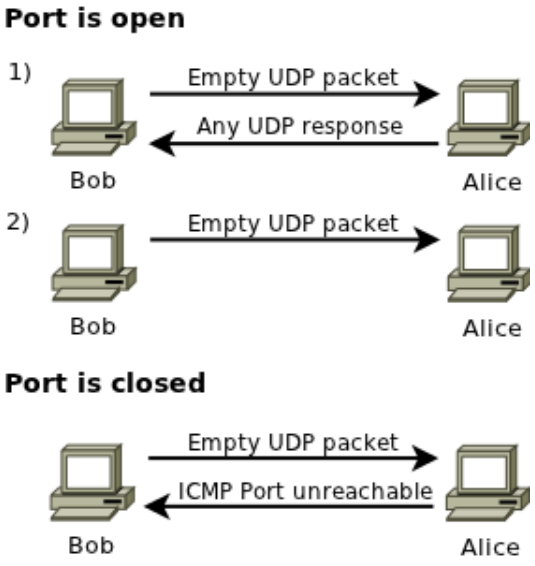

Figure 2.12: UDP Scan

Speed is a concern when performing a UDP scan. The reason being that open and filtered ports rarely send any response, leaving the scanner to time out. Another reason is that many operating systems limit the amount of ICMP port unreachable messages that can be sent as response. In GNU/Linux this is often limited to as little as one per second by default. Because of this it can take lots of time to scan a single host using this method.

2.3.4 TCP FIN Scan

TCP FIN scan uses a loophole in the TCP RFC (RFC 793) to detect whether a port is open or closed [9]. What the RFC states is that if the destination port is closed and the packet does not contain a RST packet, then a RST packet should be returned. Hence if a host is compliant with the TCP RFC, any packet that does not have the SYN, RST or ACK bits set will result in a returned RST packet if the port is closed and no response if the port is open.

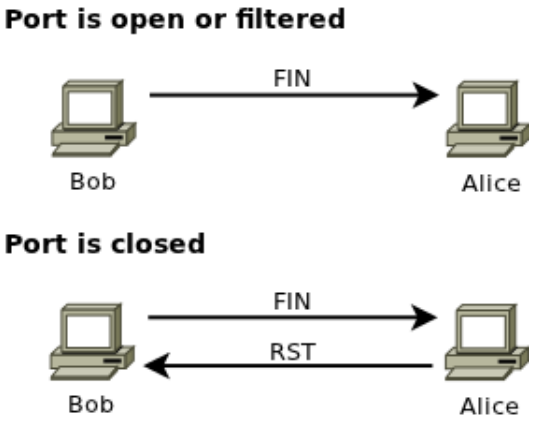

Figure 2.13: TCP FIN Scan

The advantage of using this method is that it can easily bypass filtering firewalls that block TCP packets with the SYN flag set. Unfortunately, not all systems follow the TCP RFC, in addition it can not differentiate between open and filtered ports, rendering this method ineffective in some cases.

2.3.5 Determine Traffic Direction

When trying to detect assets based on looking at network traffic, it is important to determine the direction of the traffic, which host is the server and which host is the client. Unlike port scanning where we are actively trying to find open ports on a remote host, passively looking for ports is difficult, because it can be difficult to figure out the roles of the hosts involved.

One way of detecting the traffic direction is by looking at the TCP handshake that occur every time a TCP connection is established [12].

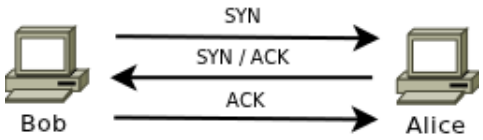

Figure 2.14: Establishing a TCP connection

When a client wants to connect, it sends a SYN packet to the server. The server then responds with sending a SYN/ACK packet back. Because of this, it is simple to see who is acting as a server in a connection, exemplified by Alice acting as the server in figure 2.14, since she is sending a SYN/ACK packet to Bob. The method described in this section is used by PRADS (section 2.6) when passively detecting network assets.

2.4 Remote Operating System Detection

Remote operating system detection is the process of determining the identity of a remote host's operating system [13]. This is done by sending packets to a remote host and analyzing the response. The response is compared to a database that contains fingerprints for the different operating systems.

Remote operating system detection was originally done by using a technique called banner grabbing. This was done by looking at the service banner displayed when trying to connect to a service like ftp or similar services.

Newer forms of remote operating system detection is based on a finger-

printing approach. What it does is that it looks for operating system specific traits in the answer it receives from packets sent to remote hosts.

Xprobe2 is a tool that uses fingerprinting to determine the operating system of remote hosts. It needs at least one closed UDP port to work and it relies primarily on the use of the ICMP protocol.

It consists of several modules, or tests that are run against the target machine. The first two modules are reachability tests, that try to determine whether the target machine is alive or not. This is done by sending an ICMP echo request to the target. If the host is alive, it sends an ICMP echo reply back. The other test is a distance test, where a TCP packet with the SYN flag is sent to the target. The goal here is to get either a TCP packet with the SYN/ACK flag enabled meaning that the port is open, or a TCP packet with the RST flag enabled meaning that the port is closed. If no response is received another TCP packet is sent to a different port, with the same goal.

The rest of the modules are fingerprinting tests, that try to determine the operating system running on the target. These consist of five modules in total:

- Module A sends an ICMP echo request to the target.
- Module B sends an ICMP timestamp request message to the target.
- Module C sends an ICMP address mask request message to the target.
- Module D sends an ICMP information request message to the target.
- Module E sends an UDP packet acting as a DNS request to the target, trying to get an ICMP port unreachable message back.

After all the responses has been received from the modules, the scores are calculated and compared to a fingerprint database. This database contains fingerprints of known operating systems. Xprobe2 then returns an estimate of which operating system that is the most probable match for the target host. Listing 2.3 shows a partial Xprobe2 fingerprint for Microsoft Windows NT 4 server with service pack 4.

Listing 2.3: Partial Xprobe2 fingerprint

```
fingerprint {
        OS_ID = "Microsoft Windows NT 4 Server Service Pack 4"
        #Entry inserted to the database by: Ofir Arkin (ofir@sys-security.com)
        #Entry contributed by: Ofir Arkin (ofir@sys-security.com)
        #Date: 30 July 2002
        #Modified: 11 July 2003

        #Module A
        icmp_echo_reply = y

        icmp_echo_code = 0
        icmp_echo_ip_id = !0
        icmp_echo_tos_bits = !0
        icmp_echo_df_bit = 1
        icmp_echo_reply_ttl = <128

        [..]

        #Module F [TCP SYN | ACK Module]
        #IP header of the TCP SYN ACK
        tcp_syn_ack_tos = 0
        tcp_syn_ack_df = 1
        tcp_syn_ack_ip_id = !0
        tcp_syn_ack_ttl = <128
}
```

The fingerprint database is distributed with the source code and consists of fingerprints for operating systems ranging from old and new versions of Microsoft Windows to Sun Solaris and NetBSD [14]. The current version is able to uniquely identify 226 different operating systems.

2.5 Nmap

Nmap ("Network Mapper") is an open source tool for exploring networks and security auditing. It was designed to scan large networks, but it also works against single hosts. Even though its main usage is for security audits, many network administrators use it for keeping track of the assets available on their network. It is useful for routine tasks like network inventory management, managing service upgrade schedules, and monitoring host and service uptime.

Nmap is a port scanner and can be used to establish which hosts are available on the network, which services that are available on these hosts, and which operating system the different hosts run. Nmap uses several of

the methods described in the previous sections, and has several additional features. Mentioning all the methods used by Nmap, are out of the scope of this thesis. When a network administrator or auditor wants to get an overview of the network equipment and machines that are on the network, it is useful to do what Nmap calls a "list scan". What Nmap does when a list scan is performed, is that it simply lists all the targets in the specified IP range. See figure 2.4 for an example of a Nmap list scan.

Listing 2.4: Nmap list scan

```
root@rosa:/home/matsekl\# nmap -sL 143.220.114.20-30

Starting Nmap 5.21 ( http://nmap.org ) at 2012-01-02 11:24 CET
Nmap scan report for 143.220.114.20
Nmap scan report for 143.220.114.21
Nmap scan report for leda.uio.no (143.220.114.22)
Nmap scan report for astro.uio.no (143.220.114.23)
Nmap scan report for duiker.uio.no (143.220.114.24)
Nmap scan report for paradox.uio.no (143.220.114.25)
Nmap scan report for bane.uio.no (143.220.114.26)
Nmap scan report for alchemist.uio.no (143.220.114.27)
Nmap scan report for trinity.uio.no (143.220.114.28)
Nmap scan report for pippin.uio.no (143.220.114.29)
Nmap scan report for brick.uio.no (143.220.114.30)

Nmap done: 11 IP addresses (0 hosts up) scanned in 0.01 seconds
```

The list scan is useful when a simple overview of all the machines on the network is needed. However, Nmap is able to do more than this. It is also capable of providing information on which network services each host is running. Several methods for scanning hosts for services are implemented in Nmap, as different scanning methods have their strengths and weaknesses. Nmap tries to bypass the weaknesses by letting the user combine several scanning methods.

Listing 2.5 shows an example of a TCP SYN scan performed with Nmap. In this example Nmap scans through the thousand most popular ports and prints the results. In this case Nmap reports that four services are running on the host. This corresponds to the services actually running on the host.

Listing 2.5: Nmap TCP SYN scan

```
root@rosa:/home/matsekl\# nmap -sS 143.220.114.30

Starting Nmap 5.21 ( http://nmap.org ) at 2012-01-02 11:48 CET
Nmap scan report for brick.uio.no (143.220.114.30)
Host is up (0.000097s latency).
Not shown: 996 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
25/tcp    open  smtp
111/tcp   open  rpcbind
5000/tcp  open  upnp
MAC Address: 70:75:BC:B7:71:14 (Unknown)

Nmap done: 1 IP address (1 host up) scanned in 0.07 seconds
```

In addition to doing host discovery and service detection, Nmap is also able to guess the operating system running on the hosts. This is done by using a similar method to the fingerprinting method described in the section about remote OS detection (section 2.4). See Listing 2.6 for an example of remote operating system detection using Nmap.

Listing 2.6: Nmap remote operating system detection

```
root@rosa:/home/matsekl\# nmap -O waldo.uio.no

Starting Nmap 5.21 ( http://nmap.org ) at 2012-01-02 12:15 CET
Nmap scan report for waldo (143.220.114.196)
Host is up (0.00026s latency).
rDNS record for 143.220.114.196: waldo.uio.no
Not shown: 984 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
Device type: general purpose
Running: Linux 2.6.X
OS details: Linux 2.6.13 - 2.6.28
Network Distance: 4 hops

Nmap done: 1 IP address (1 host up) scanned in 1.60 seconds
```

The examples shown here are just a few of the many available options for Nmap. Nmap can provide further information on targets, including reverse DNS names, traceroute and version scanning. Version scanning makes network administrators and auditors able to detect services that are old and vulnerable. Nmap does version scanning by grabbing the banners provided by each service. However, it is possible to fool Nmap by changing the services so they either do not provide the service version or provide

a fake version. Nmap include a service database of 2.200 well-known services, making it easier to understand the results provided by Nmap.

Nmap has several possible output types. It can print the output directly to the terminal, or it can print the output in XML to a file. The latter output type is important because it can easily be converted to HTML, and parsed by other programs or imported into databases.

2.6 PRADS

PRADS (Passive Real-time Asset Detection System) is a program that passively listens to network traffic, and uses the information gathered to map the network [15]. It is able to tell which services the hosts are running and which hosts that are available on the network.

PRADS use both TCP, UDP and ICMP traffic to detect hosts, meaning that as long as a host is communicating it is detected. To be able to do this PRADS must be situated so that it listens in on all the traffic on the network. This could either be done by connecting the host running PRADS to a span port on the switch and mirroring all the traffic to it, or by using a network tap to copy all the traffic running through it (see figure 2.15). PRADS is also able to detect hosts based on ARP packets (see section 2.2.4).

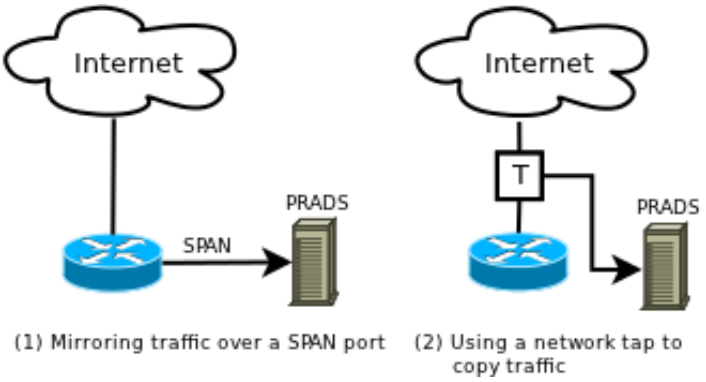

Figure 2.15: Placing the host running PRADS in the network

PRADS can be used to detect services running on the hosts. To do this, it first has to determine which of the hosts communicating that is acting as the server. It does this by using the method described in determine traffic direction (section 2.3.5). After it has decided which host is the server, it

grabs the service banner from the traffic. This way PRADS is able to detect which service it is, even if a service is running on an nontraditional port. PRADS is mostly limited to detecting TCP services. However, a few UDP services has also been included, like DNS. UDP services are only detected based on the ports they are running on.

PRADS is able to detect operating systems based on a method similar to the one described in the section about remote operating system detection (section 2.4).

PRADS has two supported output types, it can either print the results to a file or it can use a FIFO[1] device to send the results to another application. When the former output type is selected, it saves the output to */var/log/prads-asset.log* by default. Listing 2.7 shows an example of what output generated by PRADS looks like.

Listing 2.7: Example output from PRADS

```
143.220.114.42,0,443,6,CLIENT,[ssl:TLS 1.0 Client Hello],0,1313415545
143.220.114.42,0,744,6,ACK,[24565:64:1:0:N,N,T:AT:Linux:2.4(newer)/2.6:uptime:295hrs],0,1313415472
143.220.114.196,0,111,6,ACK,[91:60:1:0:N,N,T:AT:Linux:2.4(newer)/2.6:uptime:8698hrs],4,1313415601
143.220.114.22,0,22,6,CLIENT,[ssh:OpenSSH 5.8p1 (Protocol 2.0)],0,1313415735
74.125.43.19,0,443,6,ACK,[948:48:1:0:N,N,T:AT:Linux:2.4(newer)/2.6:uptime:3452hrs],16,1313418521
143.220.114.42,0,514,17,CLIENT,[unknown:@syslog],0,1313418601
143.220.114.200,0,2049,6,SERVER,[unknown:@nfs],4,1313418654
76.73.76.2,0,80,6,SERVER,[http:Apache],12,1313418752
143.220.114.200,0,2049,6,ACK,[501:60:1:0:N,N,T:ZAT:Linux:2.6:uptime:932hrs],4,1313418871
```

The output from PRADS can be incomprehensible at first since it provides a lot of information, making it unreadable for most people. However, it provides lots of useful information. From the example in listing 2.7 we can easily determine that the host with the IP address *143.220.114.42* is running a flavour of Linux that has a kernel version of 2.4 or newer. It also has a uptime of 295 hours (12 days) and has a syslog client installed. We can also see that the host with the IP address *143.220.114.200* is running a NFS server, running Linux 2.6, and has a uptime of 932 hours (38 days).

Another feature of PRADS, shown in the example, is that it shows the version of the services running on the hosts. In addition to displaying the version of network services running on servers, it also display versions of the software used by clients when connecting to the servers. Because of this, PRADS can detect if a client with a vulnerable web browser, has

[1] First in, first out

connected to a website serving malware. This feature in PRADS could be useful when used together with an intrusion detection system (IDS).

In addition to listening to traffic in real-time, PRADS also has the option to read files with already captured network traffic in pcap format. This is the same format used by TCPdump when dumping network traffic. PRADS can therefore also be used to map a network based on historic data as long as a dump of the network traffic for that period exist.

PRADS is a useful tool when network administrators want to find out what is running on their network. What makes PRADS especially useful is that all the asset information is captured passively. This means that unlike Nmap, where the network administrators would have to scan each host separately, PRADS captures all the information by just listening to the network traffic. This makes PRADS much less intrusive than active network fingerprinting tools like Nmap, which is a good thing because it does not stress the scanned systems and it does not set off alarms in intrusion detection systems.

Chapter 3

Design

In this chapter we describe the research method chosen for this thesis. We also present the requirements for our passive asset detection system. Finally we give an overview of the system design and the design considerations made before implementing the system.

3.1 Research Method

The research method we choose for this thesis is to design and implement a passive asset detection system that uses data collected by NetFlow to detect assets on a computer network. The reason why we choose to do this is because there is a need for a solution that is able to collect assets on computer networks with high network throughput, and as far as we know this has not been done before.

3.2 Requirements

In the following sections we describe the requirements that the system implemented in this thesis should fulfill.

3.2.1 Real-time

The system should be able to run and detect new assets at all times. This prevents the collected data from being outdated by constantly renewing it. At the same time it gives the network administrators an overview of the new assets that have been collected by the system. Because of this requirement the processing time of the system must be as low as possible.

3.2.2 Scalable

The system must be scalable enough to not only handle todays network throughput, but also handle the network throughput that is expected in a few years time. It must also be able to handle new functionality that is added in the future. The system should be able to handle large scale networks, with at least Gbit/s of network traffic.

3.2.3 Detection Rate

The detection rate of the system must be high. Since the system sees all the traffic flowing through the network, it should be able to detect most of the assets in a short amount of time.

3.2.4 False Positives

The system should have as few false positives as possible. The assets it detects should exist on the network.

3.3 System Design

In the following sections we give an overview of the bits and pieces that the system should consist of, the design considerations that have been made, and what the system actually has to do.

3.3.1 System Overview

The tasks that the system implemented for this thesis have to perform can be abstracted into three parts. Figure 3.1 shows how the different parts interact with each other.

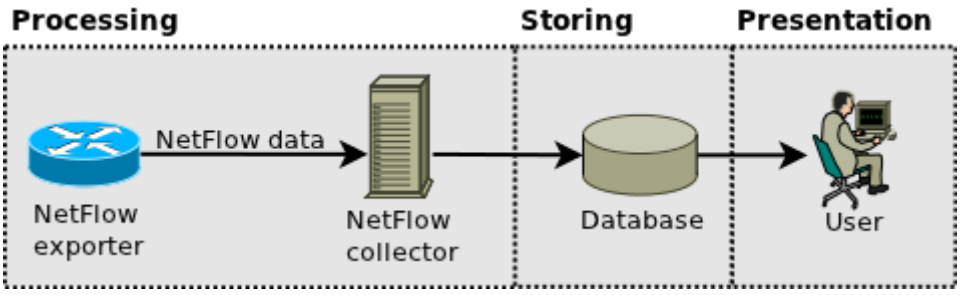

Figure 3.1: Design overview

The system has to process all the NetFlow data that is captured by the NetFlow exporter and exported to the NetFlow collector. Based on this data the system must detect all involved assets on the network based on specific rules. The system must store information about each detected asset in a database. Finally, the stored asset information must be read from the database and presented to the user in an appropriate format.

3.3.2 Processing

Processing data in our system could be done in three steps:

1. Read the NetFlow data collected.
2. Process the data and detect assets based on rules implemented.
3. Write the assets to a file or pass them on to another process.

The first thing that has to be done when processing the data, is to read the NetFlow data. It varies greatly from collector to collector how this can be done. In our system we have chosen to use Nfdump tools (section 2.1.6) as the collector software. Because of this we could either read the binary files generated by Nfdump tools' capture daemon (nfcapd), or we could wrap our system around Nfdump tools. Because the system

we are implementing are meant as a *proof-of-concept* and because of time constraints, we have chosen to do the latter.

After the data has been read, it has to be processed. In this thesis we have chosen to do a rule based approach where assets are detected when they trigger rules that has been implemented.

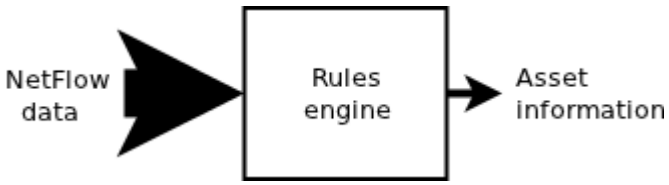

Figure 3.2: Detecting assets using a set of rules

As seen in figure 3.2 all the NetFlow data are processed by passing it through a set of rules. It is expected that the amount of NetFlow data that is being processed is much higher than the amount of assets that are detected. The reason for this is that one single host can send and receive lots of network traffic, but not all the flows contains information useful for detecting assets.

When an asset is detected it must be written to a file or passed on to an other application for further processing. This could be done by writing it to a comma-separated values (CSV) file.

To fulfill the requirement defined earlier in this chapter that the system must be able to run in real-time, processing the data must be quicker than the time interval between each time nfcapd rotates the NetFlow data file. It does this by default each five minutes. Therefore, run-time of the processing component must be lower than five minutes.

3.3.3 Storing

Storing the collected asset information can be done in three steps:

1. Read the asset information collected when processing the data.
2. Check for duplicates.
3. Write the asset information to the database.

Firstly, we have to read the asset information collected when processing the NetFlow data. This is done by reading the CSV-file generated in the previous part.

Then each of the asset information in the CSV-file must be checked for duplicates. By duplicates we mean asset information that are exactly the same. It is worthless for us to store the same information twice. Besides, database operations are expensive and we do not want to store more asset information than necessary in the database. Therefore, duplicates should be removed.

The last step involves storing the asset information in the database. The most important thing to remember here is to minimize the amount of database operations that has to be done for each asset. We have to remember that for each extra database operation we have to do, it ends up slowing down the system, since we are probably going to store information about thousands of assets in each run.

3.3.4 Presentation

Presenting the asset information stored in the database to the user can be done in two steps:

1. Retrieve the asset information from the database.
2. Present the asset information to the user.

The first thing that has to be done is to retrieve the asset information from the database. This is done by connecting to the database and retrieving the asset information that the user wants to view.

The user should be able to search for assets based on host name, IP address and subnet. It should also be possible to see statistics on the asset information stored in the database.

The amount of information that is presented should not overwhelm the user with information, but it should provide enough information for the user to understand what the assets are. For instance the system should present a description of services that are detected for an asset, not just present port numbers.

Chapter 4

Implementation

In this chapter, we take a closer look at implementing a passive asset detection system based on NetFlow data. We look at how the components are implemented and how they work together, how the rules to process asset data are implemented and how the database is structured.

4.1 Implementation Overview

The system is divided into four components, each responsible for performing a specific task. Figure 4.1 shows an abstract overview of the implementation and how the components interact with each other.

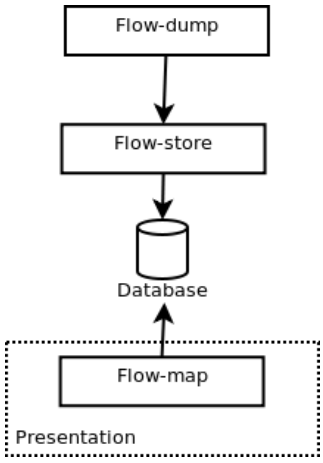

Figure 4.1: Implementation overview

4.2 Flow-dump

Flow-dump is the core component in the system. It reads NetFlow data from the nfcapd files generated by the NetFlow capture daemon (nfcapd) and processes the NetFlow data based on a set of pre defined rules. The asset data collected is then sent to the output module that outputs the data based on the output mode selected.

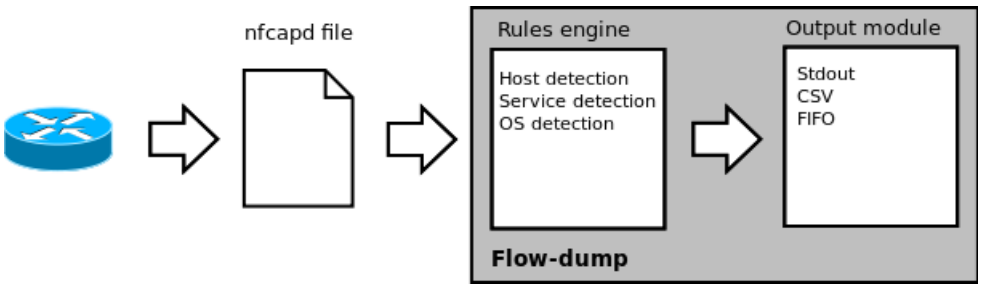

Figure 4.2: Overview of Flow-dump

Nfdump has several ways to output data to the screen. One of these is to output the data as comma-separated values, which is easy to process by other applications. It was possible to increase the processing speed of Flow-dump considerably when only the data fields needed for asset detection was read from the nfcapd files. These data fields are presented in table 4.1.

Data field	Description
%ts	Start time - first seen
%sa	Source IP address
%da	Destination IP address
%sp	Source port number
%dp	Destination port number
%pr	Protocol (TCP/UDP/ICMP)
%flg	TCP flags
%ipkt	Input packets
%ibyt	Input bytes

Table 4.1: Nfdump data fields used by Flow-dump

Flow-dump can either be run in real-time mode or it can be used to read a selected nfcapd file. If it is run in real-time mode, it waits for the next nfcapd file to be written, and then reads and processes it. When it completes processing the file, it waits for the next file and so on. Flow-

dump can also be used to read selected nfcapd files. Thus, enabling us to read and process historic NetFlow data. Because of this, Flow-dump can be used to get a complete view of a network even after a security breach has occurred, if the NetFlow data for this period is stored securely. This gives the flexibility to process historic network traffic as long as the data is available, which could be months or even years back in time.

We created a shell script to be able to process a whole day worth of flows (see Appendix C.2). Flow-dump itself only utilizes one CPU core at a time regardless of how many cores are available. This script utilizes several cores simultaneously by starting several Flow-dump processes (11 processes in our case). By running several Flow-dump processes, the total time to process one day of flows is reduced considerably.

When collecting asset data we are only interested in what is happening on our own network. To avoid processing and storing unnecessary data, it is possible to specify which subnets to monitor in Flow-dump's configuration file. The format used is subnet/netmask (e.g. 143.220.0.0/16 or 143.220.12.0/24). It is important to remember that each subnet that is specified will add substantial processing time to the component, because the component will have to look for occurrences of the subnets in all the flows processed. Therefore, it is more efficient to define one large subnet rather than several smaller subnets. See table 4.2 for a complete list of options available in the configuration file.

Option	**Type**	**Description**
debug	Variable	Turn debugging on or off (1/0)
filepath	Variable	The path to the nfcapd files
flowsources	Array	The NetFlow streams to read
networks	Array	Define the subnets to monitor
outputmode	Variable	Select output mode (CSV/FIFO/stdout)
host_blacklist	Array	List of blacklisted hosts
port_blacklist	Array	List of blacklisted services
services	Array	List of pre-defined services
updateservers	Array	List of update servers

Table 4.2: Options in Flow-dump configuration file

In contrast to Nfdump where one nfcapd process is needed for each NetFlow stream, Flow-dump processes can handle several NetFlow streams at once. NetFlow streams can easily be defined in the configuration file. This

enables Flow-dump to read streams stored by all nfcapd processes in one go, and we no longer have to start several processes to cover all NetFlow streams.

When someone is actively scanning the network with tools like nmap, Flow-dump receives data about hosts and services that does not exist. This creates data that is worthless to the user and that contributes to polluting the collected data. However, this can easily be fixed by adding a blacklist. Flow-dump blacklist feature allows blacklisting known network scanners by adding them to the blacklist in the configuration file. This means that if a host on our network is contacted by a blacklisted host, these flows will not get stored. A consequence of this is that we would not see attacks from these hosts, so there are both advantages and disadvantages of doing this.

Several of the options in the configuration file can be overwritten at run time by command line options. One example of this is the output mode option. It can easily be overwritten by specifying the command line option *–output* at run time. We can also select which nfcapd file to read from with *–read* and which file to save the output in with *–write*. The complete usage information for Flow-dump can be seen in listing 4.1.

Listing 4.1: Usage information for Flow-dump

```
Usage: ./flow-dump [OPTION]...
Passive Asset Detection using NetFlow.

  -r, --read <file>   : nfcapd file to read
  -o, --output <type> : overwrite output mode (csv/fifo/stdout)
  -w, --write <file>  : write to file
  -h, --help          : display this help and exit

Write option only works for CSV and FIFO output modes.
```

When Flow-dump triggers on any of the pre defined rules, it outputs the asset data collected using the selected output mode. The output modes supported by Flow-dump are stdout (standard output stream), CSV (comma-separated values) and FIFO (named pipes). The reason why several output modes was implemented was to make it easier for other developers to use the output from the component as input for their own applications.

The stdout output mode was mainly implemented for testing purposes, as it prints the output directly to the screen. By doing this, we can easily see if rules are working as they should. This can also be used to pipe the

output to another programs using Unix pipes. However, this is not the most suited output mode if the output is going to be used as input for an other application. See listing 4.2 for an example of how output from the stdout output mode looks like.

Listing 4.2: Example output from the stdout output mode

```
2011-11-13 16:29:36 - Service detected - 143.220.5.27,53/UDP
2011-11-13 16:29:56 - Service detected - 143.220.11.62,80/TCP
2011-11-13 16:29:36 - Host detected - 143.220.9.216
2011-11-13 16:29:36 - OS detected - 143.220.1.9,742,Windows
2011-11-13 16:29:36 - Service detected - 143.220.61.131,22/TCP
2011-11-13 16:29:36 - Service detected - 143.220.4.40,53/UDP
2011-11-13 16:29:36 - Host detected - 143.220.15.7
2011-11-13 16:29:36 - OS detected - 143.220.3.40,RedHat
2011-11-13 16:29:36 - Service detected - 143.220.1.9,25/TCP
2011-11-13 16:29:56 - Service detected - 143.220.8.39,21/TCP
```

To make it easier for external components to use the output from Flow-dump, the CSV output mode was implemented. This output mode stores assets detected by Flow-dump in a file as comma-separated values. This makes using the output from Flow-dump as simple as just reading the file. Basic UNIX tools like *cat* could also be used to print the file to stdout and piping it into the application. See listing 4.3 for example output using this output mode.

Listing 4.3: Example output from the CSV output mode

```
Service,143.220.7.45,445/TCP,2011-11-13 16:29:08
Service,143.220.3.3,53/UDP,2011-11-13 16:29:28
Host,143.220.13.7,2011-11-13 16:29:28
Host,143.220.240.101,2011-11-13 16:29:32
Service,143.220.4.40,53/UDP,2011-11-13 16:29:28
OS,143.220.4.5,RedHat,2011-11-13 16:29:35
Service,143.220.15.7,80/TCP,2011-11-13 16:29:48
Service,143.220.12.40,445/TCP,2011-11-13 16:29:2
Service,143.220.65.3,123/UDP,2011-11-13 16:29:40
Host,143.220.250.211,2011-11-13 16:29:32
```

The last output method implemented is the FIFO output method. A FIFO is similar to a pipe, except that it is accessed as part of the file system [16]. It can be opened simultaneously by multiple processes for reading and writing. When a FIFO is used the kernel passes all data internally without writing it to the file system. Data is sent through first-in first-out, meaning that the first data that is written on one end is the first data that is read on the other end. The FIFO implemented in Flow-dump is implemented

as a blocking FIFO. This means that the FIFO must be opened on both ends (reading and writing) before data can be passed. Flow-dump opens a FIFO, then it blocks until an other process starts reading from the FIFO. The data sent through is on the same format as the data from the CSV output method, making it easy to read by other processes.

4.3 Flow-dump Rules

To collect assets using Flow-dump we have implemented several rules. These rules make Flow-dump able to detect hosts and services on the network. In this section we are going to describe the different rules that are implemented in this thesis.

4.3.1 Host Detection Based on ICMP

A simple way to detect hosts on a network is to look for ICMP echo packets sent by network administration utilities, like Ping. What we are interested in are the ICMP echo replies that are sent back from a machine, indicating that the machine is on the network [8].

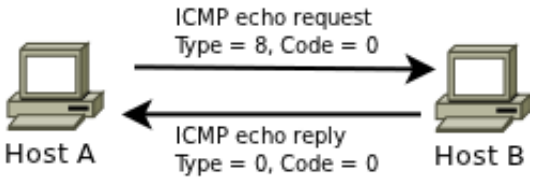

Figure 4.3: ICMP echo reply

Figure 4.3 shows a scenario where host A is sending an ICMP echo request to host B. In this case host B sends an ICMP echo reply back to host A saying that it is present on the network. The most likely alternative would be to get a host unreachable reply back indicating that the host is not available.

The reason why this rule is set up to trigger on only ICMP echo replies and not ICMP echo requests as well, is because that would have led to many false positives. A host would be "detected" every time a machine on the network tries to ping another machine, even if there is no echo reply.

Type	Code	Description
0	0	Echo reply
3	1	Host unreachable
3	3	Port unreachable
3	10	Destination host administratively prohibited
3	13	Communication administratively prohibited by filtering
8	0	Echo request
11	0	TTL equals 0 during transit

Table 4.3: Common ICMP types

In Nfdump the destination port indicates what type of ICMP packet is being sent. This makes it easy to pick out the flows containing ICMP echo replies with the correct response code by looking for flows with protocol ICMP and destination port *0.0*. See table 4.3 for an overview of the most used ICMP types.

This rule is especially useful when management tools like Nagios are used on the network. Nagios, like many other management tools uses ICMP actively to check if hosts they are monitoring are responsive.

A flow chart of the implementation of the host detection rule can be seen in figure 4.4.

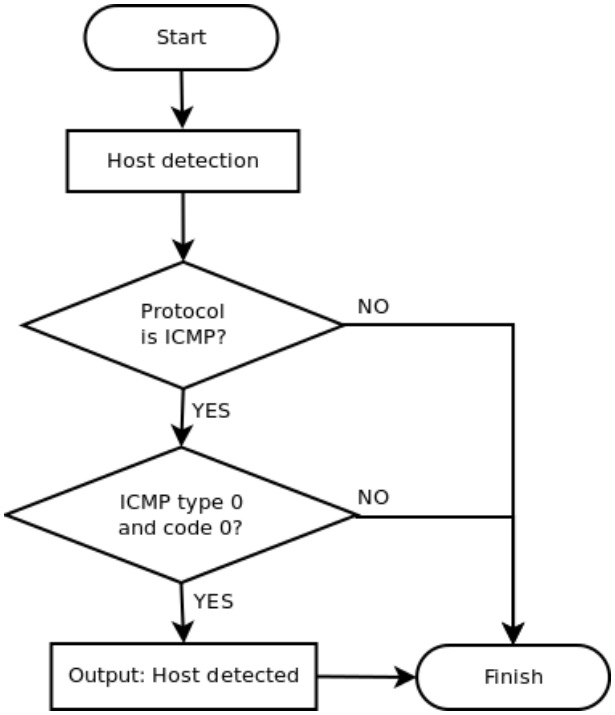

Figure 4.4: Flow chart of host detection based on ICMP

When this rule triggers, it uses Flow-dump's output module to output that a host has been detected, the IP address of the host, and a timestamp of when the host was detected.

4.3.2 Service Detection Based on Ports

One limitation of using NetFlow data instead of network traffic data is that we can not rely on TCP flags to detect which host is acting as the server. This is important to determine to detect which services are running on which host. The reason why TCP flags can not be used for this purpose is because TCP flags in NetFlow are aggregated. This can be explained by looking at the TCP handshake that occurs every time a TCP connection is initiated (see figure 4.5).

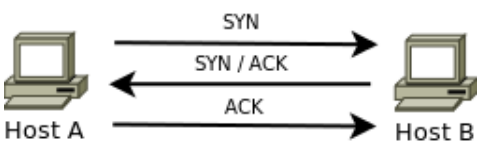

Figure 4.5: TCP handshake

The client sends a SYN packet to the server indicating that it wants to connect to a port. The server answers with a SYN/ACK packet. When the client sends the last ACK packet, the TCP connection is established. In a normal setting it would be easy to see that host B is the server in this example because of the SYN/ACK packet sent. However, since we are using NetFlow this TCP handshake would be separated over two flows (remember that NetFlow is unidirectional), one flow from the client to the server, and one flow from the server to the client. Both of these flows would have the TCP flags SYN and ACK set. It would therefore be impossible to make a decision on the roles of the hosts based on TCP flags when using NetFlow data.

There are some Cisco switches and routers that do not fully support sending TCP flags with NetFlow data [17]. We have therefore decided to not rely on TCP flags in our implementation.

The method we have chosen to determine the direction of the traffic is by looking at which ports the hosts are communicating on. Most services listen on ports under 1024. When a client connects, it usually communicates from a port above 1024. To detect services, the rule looks for hosts communicating on a port under 1024 with another host on a port above 1024. This means that we do not detect services communication on ports above 1024. The next rule tries to solve this problem (see section 4.3.3).

There are some pitfalls to using this method. Apart from not detecting services on ports above 1024 some services generate false positives by communicating from a port above 1024 with ports under 1024. One service that in our experience causes lots of false positives is NFS (Network File System). The reason for this is because NFS uses a vast amount of ports, alternating. We solve this by creating a port blacklist. Entries to this blacklist can be added to the Flow-dump configuration file and are on the format *direction,port*. To blacklist services where portmap (used by NFS) is contacting another host, we can add *dst,2049* to the blacklist. By using the blacklist actively the number of false positives could be reduced considerably.

Figure 4.6 shows a flow chart of the implementation of the service detection based on ports rule.

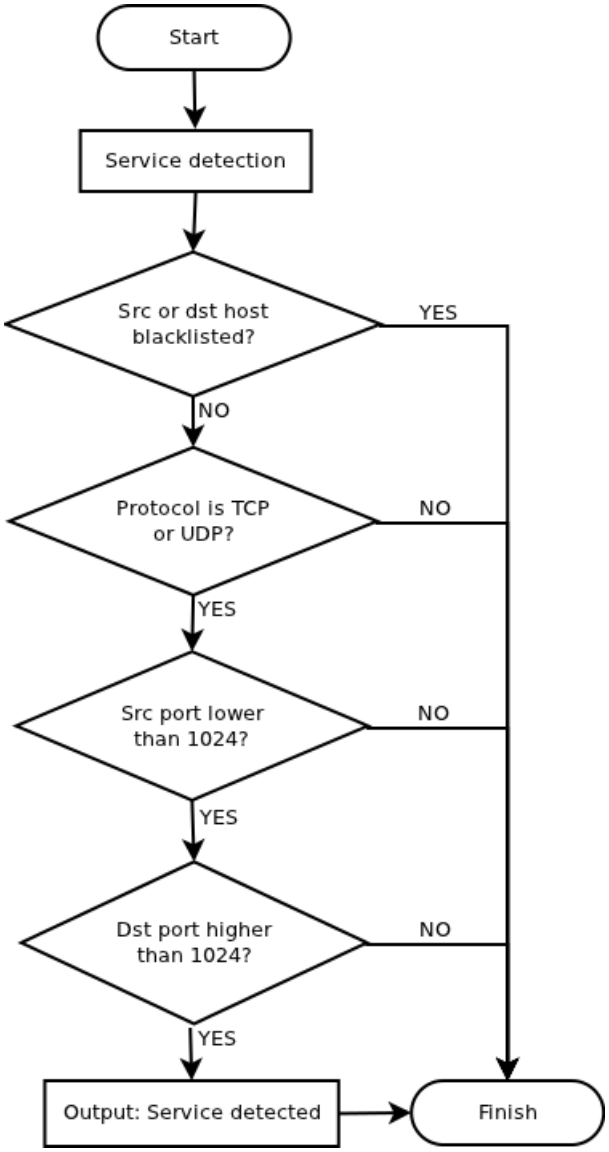

Figure 4.6: Flow chart of service detection based on ports

When this rule triggers it uses Flow-dump's output module to output that a service has been detected, the IP address of the host, what protocol the service is communicating with (TCP or UDP), the port number of the service, and a time stamp of when the service was detected.

4.3.3 Service Detection Based on Pre defined Services

As mentioned in section 4.3.2, we can not rely on TCP flags when using NetFlow data for asset detection, which limits us when detecting services. By using the rule in the previous section we only detect services running on ports beneath 1024.

The next rule tries to make the results a little more accurate by looking for known services. What it does is to look for hosts communicating on ports specified in the Flow-dump configuration file. Similar to the rule in section 4.3.2, the port number used by the other host must be above 1024 for the rule to trigger. The reason for this is to avoid false positives. One example of this is that Host B is running a web server on port 80. Host A connects to Host B and is assigned a random port number to communicate on. If this random port number is the same as one of the pre defined services used by this rule, then it would look like both Host A and Host B is running a service even if only one of them really are. It is far more unlikely to encounter this if we limit the ports of Host A to port numbers above 1024.

Services can easily be added to the *services* array in the Flow-dump configuration file. The format used is *port,service*. Services can be fetched from */etc/services* or from Nmap's *nmap-services* file. An example can be seen in listing 4.4.

Listing 4.4: Array of pre-defined services

```
@services = [
              '2049,TCP',  # portmap
              '3306,TCP',  # mysql
              '5432,TCP',  # postgresql
              '4045,TCP'   # lockd used by NFS
            ];
```

This rule also uses the host blacklist to avoid network scanners and hosts that we consider uninteresting in a service respect. However, it does not use the port blacklist used by the rule in section 4.3.2, because the port blacklist could easily contain services that generate false positives when used with detection in the previous section, but that we want to look closer at using this new rule.

This rule only detects known and pre defined services. It falls short when trying to detect unknown services, like backdoors. It is primarily meant as a supplement to service detection based on ports beneath 1024.

Figure 4.7 shows a flow chart of the implementation of the service detection based on pre defined services rule.

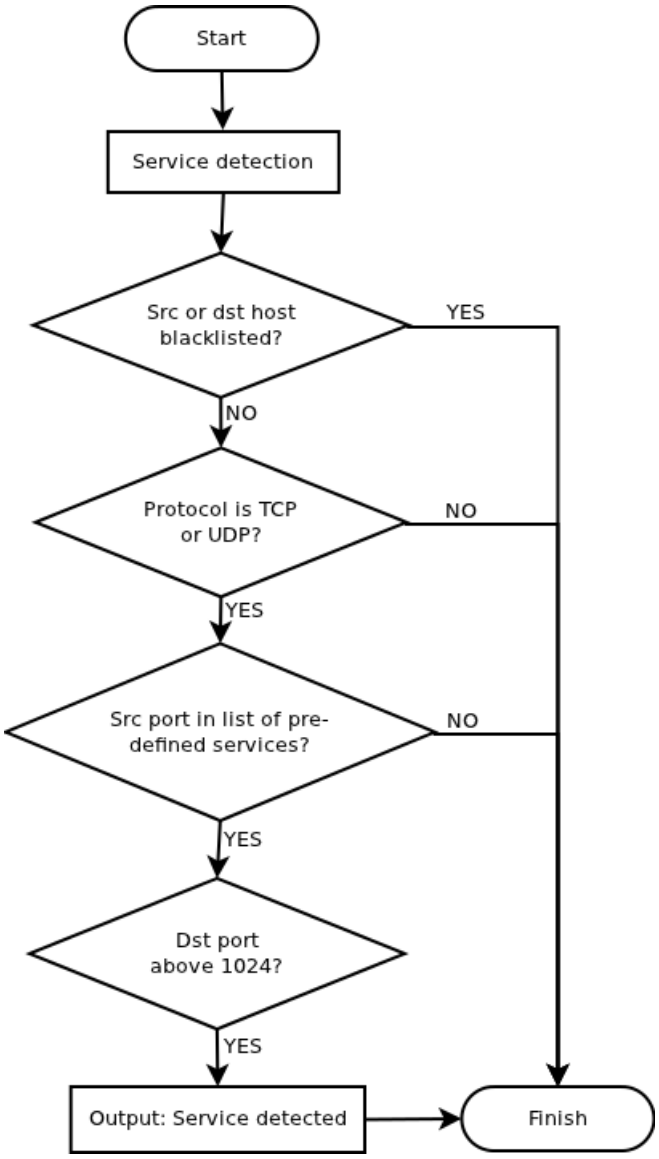

Figure 4.7: Flow chart of service detection based on pre-defined services

When this rule triggers it uses Flow-dump's output module to output that a service has been detected, the IP address of the host, what protocol the service is communicating with (TCP or UDP), the port number of the service, and a time stamp of when the service was detected.

4.3.4 Operating System Detection Based on Update Servers

For an asset detection system to be valuable, it should provide as much information about the target hosts as possible. Knowing the operating system of a host can be useful when trying to determine if it is vulnerable to new exploits. It is also useful to be able to easily see what kind of operating systems hosts on the network use.

The method this rule uses to detect operating systems is to look for connections made to update servers. Based on observations, this usually seems to be done using TCP on port 80. The reason probably being to avoid firewalls, since it is uncommon to filter this port, as clients use this port to browse the web.

The update servers for the different operating systems can be specified in the Flow-dump configuration file and is on the format *IP address,Operating System,port*. Adding the port number is not mandatory, and it defaults to port 80 if not specified.

Listing 4.5: Update servers used in this thesis

```
@updateservers = [
                '143.220.2.25,RedHat',    # yum.uio.no
                '65.55.0.0/16,Windows',   # update.microsoft.com
                '143.220.12.27,Windows',  # wsus.uio.no
                '17.250.248.95,Darwin',   # swscan.apple.com
                '129.241.93.37,Ubuntu'    # no.archive.ubuntu.com
                ];
```

Update servers for the operating systems shown in listing 4.5 was added, because these are the most common operating systems on the specific network that was used to test our implementation. However, additional update servers could easily be added to the configuration file.

It is also possible to add entire subnets to the configuration file, instead of just single IP addresses. This is mainly to detect hosts running Windows, as Microsoft uses load balancing on their update servers, and therefore changes the IP address that *update.microsoft.com* resolves to once every three hundred seconds. We originally made a script to keep a list of update servers used by Microsoft (see Appendix C.3). This worked when feeding Flow-dump with real-time data, but it proved inaccurate when running Flow-dump with historic data. Therefore, we added the possibility to add entire subnets instead.

A flow chart of the implementation of the operating system detection based on update servers can be seen in figure 4.8.

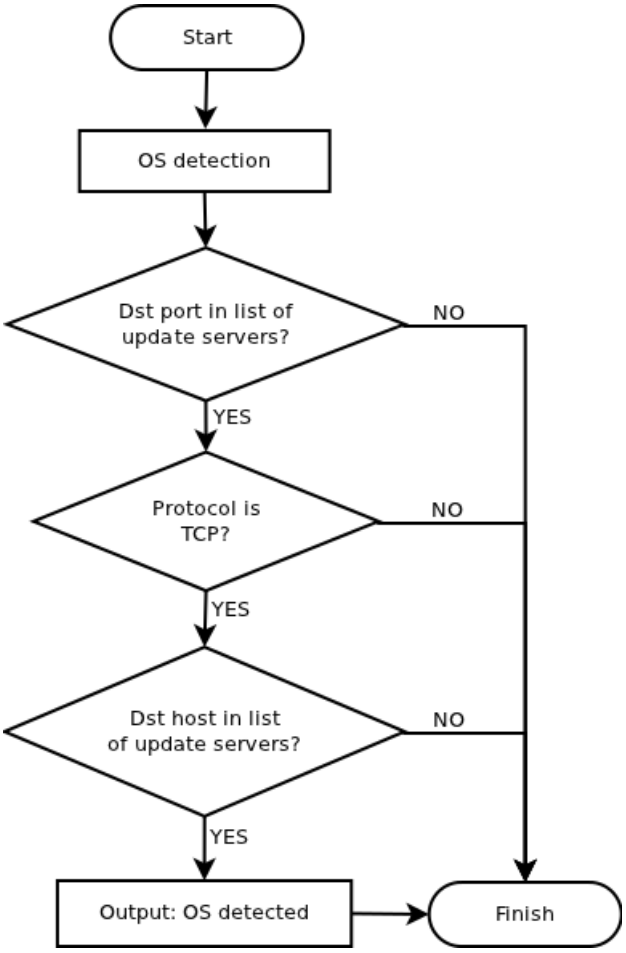

Figure 4.8: Flow chart of operating system detection based on update servers

The order of the checks has been carefully selected to save processing time. It was possible to increase the processing speed by checking for port numbers used by the update servers before comparing the destination IP address with the IP addresses in the list of update servers. This is because comparing port numbers is faster than comparing IP addresses.

When this rule triggers it uses the Flow-dump output module to output that an operating system has been detected, the IP address of the host, the operating system that has been detected, and a time stamp.

4.4 Flow-store

Flow-store is responsible for reading the asset data provided by Flow-dump and storing it in a database for later usage. The reason why Flow-dump does not handle this task itself is because database operations take longer time to process than the implemented output modes, and time is sparse when we are handling real time NetFlow data. Separating components also makes it easier to share the workload between several servers, making it possible to use the implemented system on low end equipment.

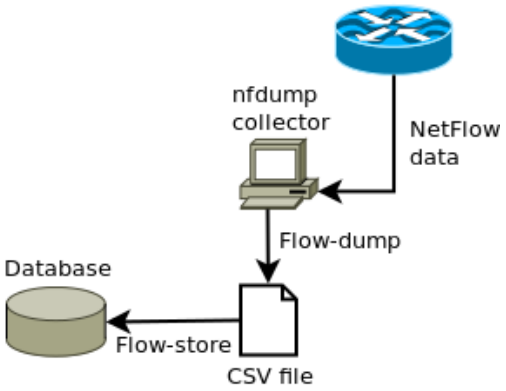

Figure 4.9: Overview of Flow-store

When handling around half a million entries once every five minutes the need for speed is a priority. Therefore, measures have been taken to make Flow-store as fast as possible. Speed improvement measures include both on the database queries and on pre processing of data.

Instead of searching for similar entries when trying to store assets in the database, the new assets are blindly inserted into the database using a *INSERT or IGNORE* query. If the asset already exist the *INSERT* query fails and an *UPDATE* query is run instead to update the last seen time stamp of the asset. Doing this saves one or two *SELECT* queries depending on the asset that is being inserted, thus saving processing time.

However, the biggest time saver came from pre processing the asset data before storing it in the database. Flow-dump produces a lot of similar entries, since each connection made to a web server would make Flow-dump detect that a web server is running and writing this to the asset file. Traversing through the asset data and removing similar entries makes the

amount of data considerably smaller. This is done by taking a backup of the asset file, reading through it line by line, and storing the unique entries in a hash (the time stamp is removed before comparing). The original asset file is then replaced by the new one only containing unique entries. Testing shows that this reduces the amount of data that Flow-dump produced by approximately twenty times.

By default, Flow-store reads from *assets.csv*. However, this can be overwritten by the command line option *–read <file>*. The data is read from the file, processed and then stored in a database. The default database is *assets.db*. This can also be overwritten by a command line option, by *–write <database>*. The complete usage information for Flow-store can be seen in listing 4.6.

Listing 4.6: Usage information for Flow-store

```
Usage: ./flow-store [OPTION]..
Store assets detected by Flow-dump in a database

  -r, --read <file>      : read from specified CSV file
  -w, --write <database> : write to specified database
  -h, --help             : display this help and exit
```

There are three types of asset entries that can be stored by Flow-store. These are host entries, operating system entries, and service entries. When storing host entries in the database a blind insertion is first attempted. If a host with the same IP address already exist in the database, then only the last seen time stamp of the host is updated. Otherwise, the new host is added to the database. The same goes for storing operating system entries. The only difference is that instead of just updating the last seen field in the database, the operating system field is also updated. When storing service entries, a host is added in the same way as when storing host entries. In addition to this, there is also added a service that is linked to the host entry by IP address.

4.5 Database

The database is used to store the asset data collected. Because of the vast amount of data that needs to be processed and stored, the database is a vital component in the system. The database is used by Flow-store to store assets detected by Flow-dump. It is also used by Flow-map when presenting the collected data to the user.

Because of this there are certain criteria that needs to be fulfilled:

Lightweight The database must consume a small amount of the systems total processing resources.

Portable It must be easy to create new databases. It must also be easy to move the databases between systems and to backup the database.

Speed The database must be fast enough to handle the amount of data provided by Flow-store.

Programming language bindings There must exist libraries for interacting with the database from the selected programming language.

After evaluating different databases using the criteria above, the decision finally landed on using a SQLite database for the system. The main reason for this is because it is self contained, meaning that it requires very minimal support from external libraries or from the operating system [18]. This makes it highly portable from system to system. This also makes it easy to write programming language bindings for SQLite, which has already been done for a lot of programming languages, including Perl used for the system in this project. In addition to this, the entire SQLite database is contained in a single file, making it easy to move around and back up if necessary. Testing, using a SQLite database with Flow-store shows that it is more than fast enough.

The database schema used by the system is quite simple (see figure 4.10). It consists of two tables, one for hosts and one for services.

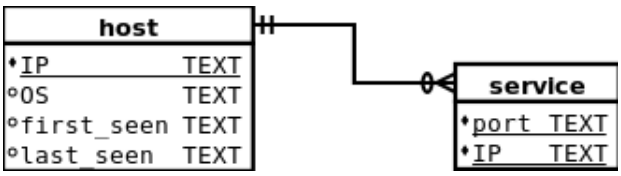

Figure 4.10: E-R diagram of the database used by the system

The host table is used for storing hosts. A host is identified in the table by the *IP* field. This field is the primary key in the host table, meaning that there can not exist two hosts with the same IP address in the database. This makes it simple to look up hosts based on IP address. Other fields in the host table are *OS* used to store the operating system of the host, and *first and last seen* used to store time stamps of when the host was detected. The last seen time stamp can be used to remove hosts that have not been seen in a while from the database.

The service table is used for storing services in the database. It consists of only two fields, a *port* field for storing the port number of the service and a *IP* field for storing the IP address the service belongs to. A host in the database is allowed to have several services connected to it while a service must belong to a single host.

A shell script was added to set up and create the database. This script is added in Appendix C.1.

4.6 Flow-map

Flow-map is the component that presents the asset data stored in the database to the user. Flow-map is intended to look like the active fingerprinting tool, Nmap. It has a command line interface and it allows the user to search for assets by specifying a target. It can also display statistics of assets discovered.

When using Flow-map to display asset data, a target must be specified. This can be done by using the command line option *–target <target>*. The target could be either a single IP address, a host name or an entire IP range. If the target is a single IP address, Flow-map only shows data for that IP address, the same thing applies when a hostname is selected as the target. In this case, the hostname is resolved and the asset data for that IP address

is displayed. When an entire IP range is selected, then asset data for each host in that IP range is displayed.

Listing 4.7: Example output from Flow-map

```
$ ./flow-map -t 143.220.8.
Starting Flow-map 1.0 at Mon Nov 28 10:50:49 2011

Host: 143.220.8.40 (fuzzy.uio.no)
First seen: 2011-11-12 01:04:26
Last seen:  2011-11-15 16:00:18
PORT       SERVICE             DESCRIPTION
111/TCP    rpcbind             portmapper, rpcbind
445/TCP    microsoft-ds        SMB directly over IP

Host: 143.220.8.45 (ryder.uio.no)
First seen: 2011-11-12 03:16:12
Last seen:  2011-11-15 16:00:03
OS: RedHat
PORT       SERVICE             DESCRIPTION
111/TCP    rpcbind             portmapper, rpcbind
445/TCP    microsoft-ds        SMB directly over IP
80/TCP     http                World Wide Web HTTP
139/TCP    netbios-ssn         NETBIOS Session Service

Host: 143.220.8.46 (spectre.uio.no)
First seen: 2011-11-12 12:00:51
Last seen:  2011-11-15 15:59:42
PORT       SERVICE             DESCRIPTION
445/TCP    microsoft-ds        SMB directly over IP
139/TCP    netbios-ssn         NETBIOS Session Service
80/TCP     http                World Wide Web HTTP
137/UDP    netbios-ns          NETBIOS Name Service

Flow-map done. Found 3 hosts.
```

Flow-map displays the IP address and host name of the selected targets. It also displays the time stamp of when the host was first detected and when it was last seen. In addition to this, it displays the operating system of the targets and which services they are running. In the listing 4.7 the operating system is only shown for one of the hosts. The reason being that the operating systems for the other two hosts has not yet been detected.

When displaying services, Flow-map uses the nmap service file *nmap-services* to map port numbers to known services. This provides the user with the name of the service and a description, making it easier to get a quick glimpse of what services that are running on a target.

The amount of data displayed as default can be quite overwhelming when displaying data for many hosts simultaneously. Therefore, it is also

possible to limit the amount of data to only a list of hosts. This is done by specifying the command line option *–list*.

Listing 4.8: Flow-map list view

```
$ ./flow-map -t 143.220.1. --list
Starting Flow-map 1.0 at Mon Nov 28 11:07:01 2011
Host: 143.220.1.1 (elixir.uio.no)
Host: 143.220.1.10 (arcadia.uio.no)
Host: 143.220.1.107 (eris.uio.no)
Host: 143.220.1.108 (ceres.uio.no)
Host: 143.220.1.117 (eve.uio.no)
Host: 143.220.1.119 (caliban.uio.no)
Host: 143.220.1.12 (cosmo.uio.no)
Host: 143.220.1.15 (bane.uio.no)
Host: 143.220.1.17 (oort.uio.no)
Host: 143.220.1.18 (oryx.uio.no)
Flow-map done. Found 10 hosts.
```

When this is done, it only displays the IP address and host name of the targets. This is useful when the user only wants a quick overview of the hosts on the network. This resembles using Nmap to display a list of hosts in an IP range.

Flow-map can also be used to display statistics of the assets in the database. Flow-map can display the amount of hosts and services in the database and a list of the most common services.

Listing 4.9: Flow-map statistics view

```
$ ./flow-map --stats
--- Flow-map 1.0 statistics ---
Hosts in database:     18505
Services in database:  56223

------ Top Services ------

AMOUNT   PORT       SERVICE          DESCRIPTION
6197     22/TCP     ssh              Secure Shell Login
6088     80/TCP     http             World Wide Web HTTP
2551     25/TCP     smtp             Simple Mail Transfer
2381     443/TCP    https            secure http (SSL)
1650     81/TCP     hosts2-ns        HOSTS2 Name Server
1058     161/UDP    snmp             Simple Net Mgmt Proto
909      135/TCP    msrpc            Microsoft RPC services
595      113/TCP    auth             ident, tap, Authentication Service
441      445/TCP    microsoft-ds     SMB directly over IP
298      23/TCP     telnet
```

By default the ten most frequent services are displayed. The size of the list can be changed by specifying the command line option *–num <number>*. It is also possible to specify which IP range to show statistics for. This can be done by specifying a target with the target command line option. The complete usage information for Flow-map can be seen in listing 4.10.

Listing 4.10: Usage information for Flow-map

```
Usage: ./flow-map [OPTION]..
Present asset data for selected target.

  -t, --target <target> : target to display
  -s, --stats           : summary of assets in database
  -n, --num <number>    : number of top services to show in summary
  -l, --list            : simple list of targets in given IP range
  -r, --read <database> : read from specified database
  -h, --help            : display this help and exit

Target can be either a hostname, a single IP or an entire IP range.
```

Chapter 5

Evaluation

In this chapter we evaluate the implementation of our passive asset detection system. We begin by describing the goals for our evaluation (section 5.1), then we describe our test setup (section 5.2). Finally we present the results (section 5.3).

5.1 Goals

The objectives in this evaluation is to verify that the system implemented fulfills the requirements it was designed for:

- It must be fast enough to run in real-time
- It must be scalable, to handle high amounts of data
- It must have a high asset detection rate
- It must have a low rate of false positives

5.2 Test Setup

The system was tested on USIT's (The University Center for Information Technology) NetFlow collector. This server collects NetFlow data from several Cisco switches and routers on the University of Oslo's computer network.

The server used as the collector has 16 Intel Xeon 2.27 GHz CPU cores, 74 GiB [19] RAM and 3 TiB storage space. Almost all the storage space is used to store NetFlow data. The collector heaps flow-records for approximately three months at a time.

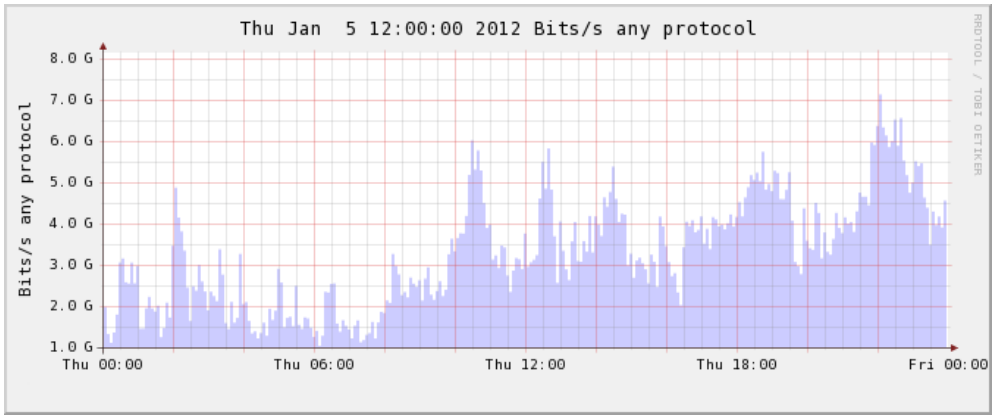

Figure 5.1: Traffic flowing through one of the University's gateways at 5 January 2012

All the tests were performed using data collected from one of the border gateways on the University network, at 5 January 2012. Figure 5.1 shows the amount of traffic that went through the selected gateway on that day.

5.3 Results

In the following sections we verify that the implemented system behaves as expected. Since the system relies on being able to run in real-time, be scalable, have a decent detection rate, and a low rate of false positives, we have decided to show the results as graphs.

5.3.1 Run-time

The system should be able to run in real-time to be able to constantly pick up new network assets. To be able to do this, the total run-time of the components must be lower than five minutes, because Nfdump tools rotates the files containing NetFlow data every five minutes by default.

Figure 5.2 (page 64) shows the run-time of the Flow-dump component throughout the day. When compared to the network traffic graph (figure

5.1) we see that the run-time does not seem to be affected much by the amount of network traffic. This is mostly due to NetFlow's ability to group the traffic into flows, so that even though the traffic amount is high, the amount of flows is much lower.

To get the full picture, we also have to look at the run-time of the Flow-store component. All the data collected by Flow-dump has to be parsed and stored in the database by Flow-store, and this also has to happen within the five minute period. The run-time for Flow-store is shown in figure 5.3. Instead of showing the run-time per hour, we show how long it takes to process and store different amounts of data. When looking at the graph, we see that the amount of time it uses does not have a linear growth. The reasons why it behaves the way it does is because of the pre processing of data that Flow-store does, and that the amount of database operations has been minimized. This makes the system highly scalable and able to handle large amounts of data.

We can conclude that the system is fast enough to handle real-time data, and that we have plenty of room for adding more rules if we find it necessary.

In addition to being able to run the system in real-time it is also interesting to see how well the system handles historic data. To be capable of doing this, it should be fast enough to process several days of data within a reasonable amount of time. To test this we checked how much time the script mentioned in section 4.2 uses when processing a whole days worth of flows. Table 5.1 shows the amount of time used by the script when processing five different days.

Date	Run-time (minutes)	Data Amount (TB)
2011-01-01	28	25.9
2011-01-02	27	29.7
2011-01-03	26	34.4
2011-01-04	26	30.4
2011-01-05	26	35.3

Table 5.1: Run-time of script when processing five whole days of data

If we look at the average processing time it is close to half an hour, so processing a week worth of flows would take a couple of hours. This makes the system capable of getting a detailed view of how a network looked at any point in time, as long as NetFlow data for that period is stored.

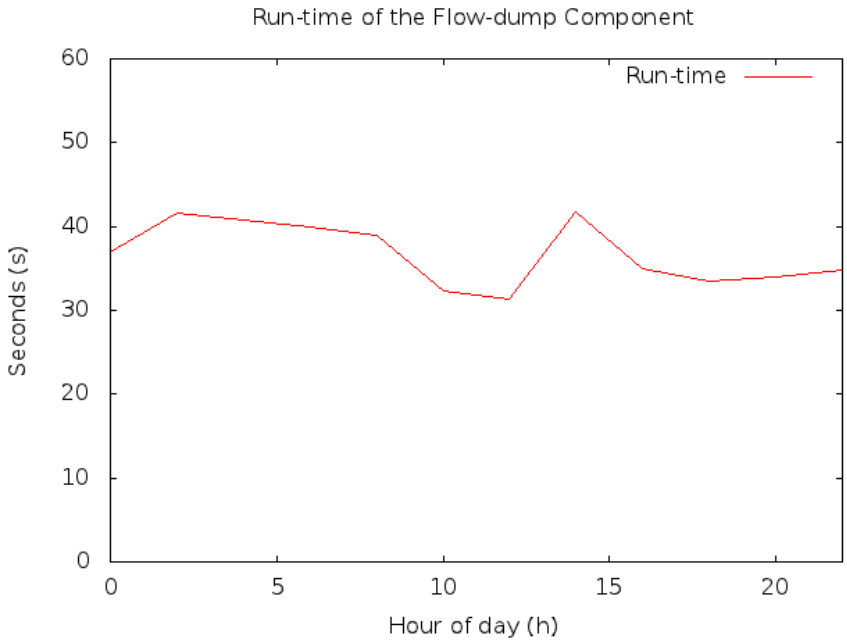

Figure 5.2: Run-time of the Flow-dump component

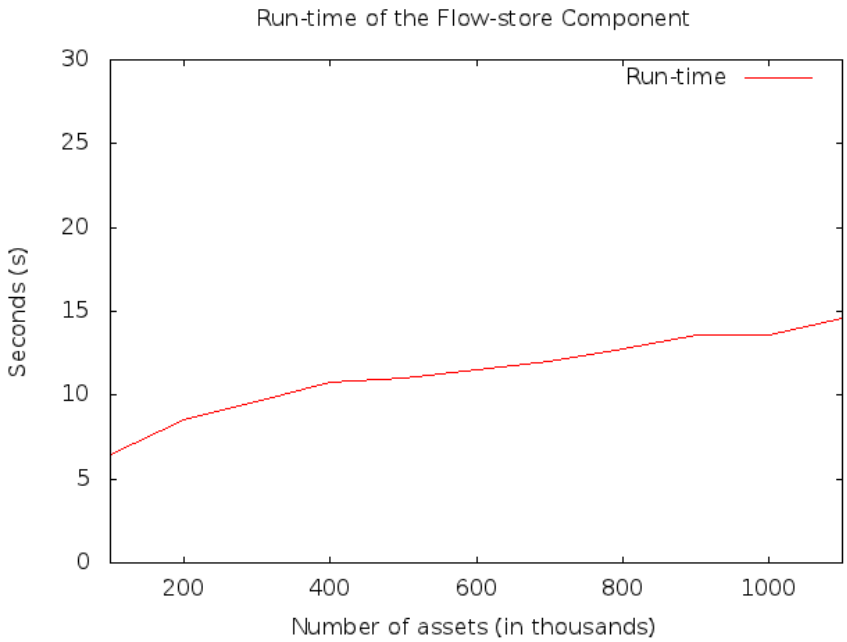

Figure 5.3: Run-time of the Flow-store component

5.3.2 Host Detection Rate

For the system to actually be of any use it has to detect network assets. This is even more important than being able to run the system in real-time. If the system does not detect assets, it is of no use to us or anyone else.

Figure 5.4 (page 66) shows the detection rate of hosts every fifteen minutes over a period of three hours. As we see in the graph there is a peak at the beginning, which means that the detection rate is at its highest when the system is started. The reason for this is that at that time the system does not know of any hosts, a "learning phase" and therefore most of the hosts are detected as new by the system. As more and more hosts are discovered, there is a longer interval of time between each time a new host is discovered.

In addition to discovering hosts using ICMP, hosts are also detected each time a service is detected and each time a operating system is detected. Basically, we can say that as long as a host is communicating with other hosts on the network it will be detected by the system.

The total amount of hosts discovered the day the graphs were made (5 January 2012) was 17 643 hosts. This means that within the first fifteen minutes of running the system we had already discovered a quarter of the hosts. After the first time period, the amount of hosts discovered each period was steadily declining.

Figure 5.5 shows an overview of the host detection rate for the entire day (the two graphs are independent of each other). The measurements are from every run made by Flow-store. This is done once every two hours and forty minutes and only unique hosts are shown. This graph shows that around half the hosts were detected within the first couple of hours. It also shows that after a little while very few hosts were detected. At 10:00 p.m. only four new hosts were detected.

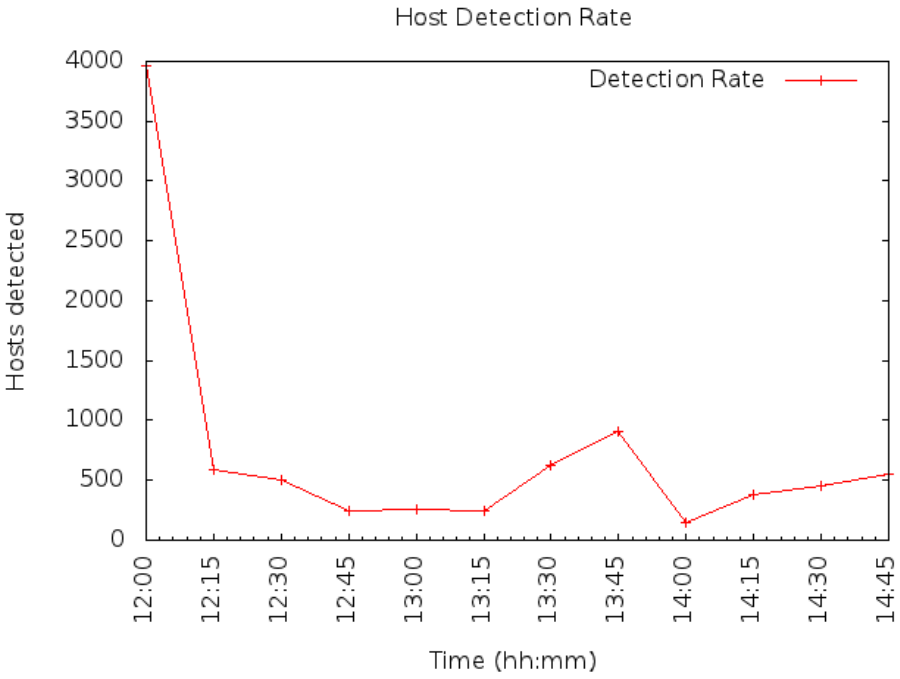

Figure 5.4: Host detection rate (short time span)

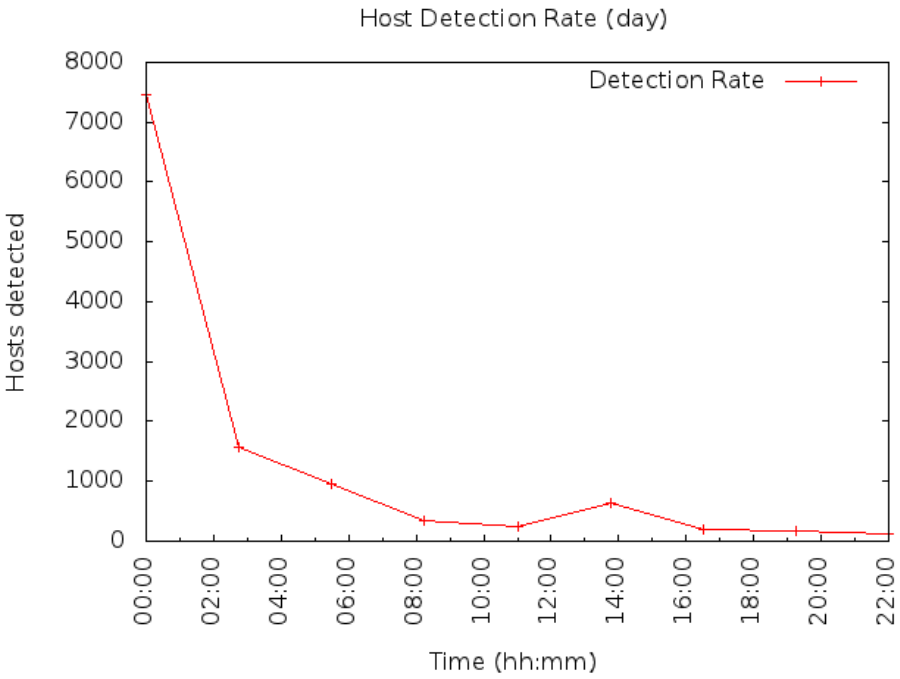

Figure 5.5: Host detection rate (day)

5.3.3 Service Detection Rate

Each host on the network can have several network services running, and not all of these services are used at all times. Therefore we can expect to find more services than hosts, and at different times of the day. The total amount of services discovered in one day (5 January 2012) were 88 595 services

Figure 5.6 (page 68) shows the detection rate of services every fifteen minutes over a period of three hours. As with the host detection the largest spike of services detected were right after the system had started. One difference is that we do not detect as high percentage at the start. We detect a little less than ten percent of the total services detected that day within the first fifteen minutes, compared to twenty-five percent of the hosts. On the other hand we detect around five hundred services every fifteen minutes after the first period, which is promising.

Figure 5.7 shows the detection rate of services for an entire day. As with figure 5.5 it shows the services detected and inserted into the database each time Flow-dump runs. It shows that we detect around a quarter of the services detected that day within the first couple of hours. The curve for the service detection is not as steep as for the host detection. There are still services to detect at the end of the day.

Amount	Port	Service	Description
5868	22/TCP	ssh	Secure Shell Login
4127	80/TCP	http	World Wide Web HTTP
3659	23/TCP	telnet	Telnet
3280	21/TCP	ftp	File Transfer [Control]
3228	443/TCP	https	secure http (SSL)
1978	81/TCP	hosts2-ns	HOSTS2 Name Server
1807	135/TCP	msrpc	Microsoft RPC services
1407	445/TCP	microsoft-ds	SMB directly over IP
1151	161/UDP	snmp	Simple Net Mgmt Proto
984	25/TCP	smtp	Simple Mail Transfer

Table 5.2: Top ten services detected

Table 5.2 shows the top ten services detected by the system. Around thirty percent of the service instances detected are among the top ten services.

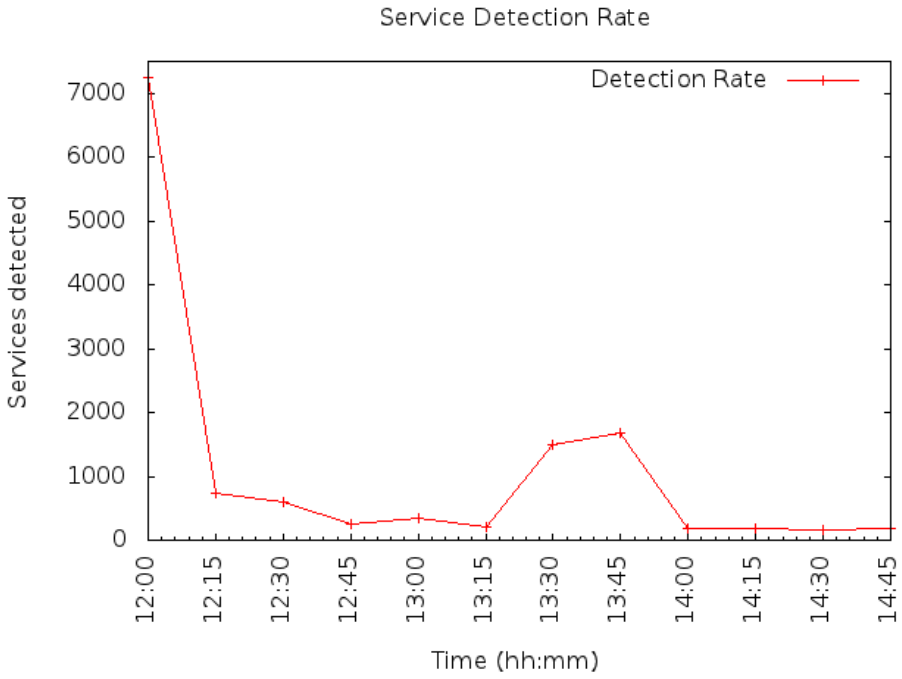

Figure 5.6: Service detection rate (short time span)

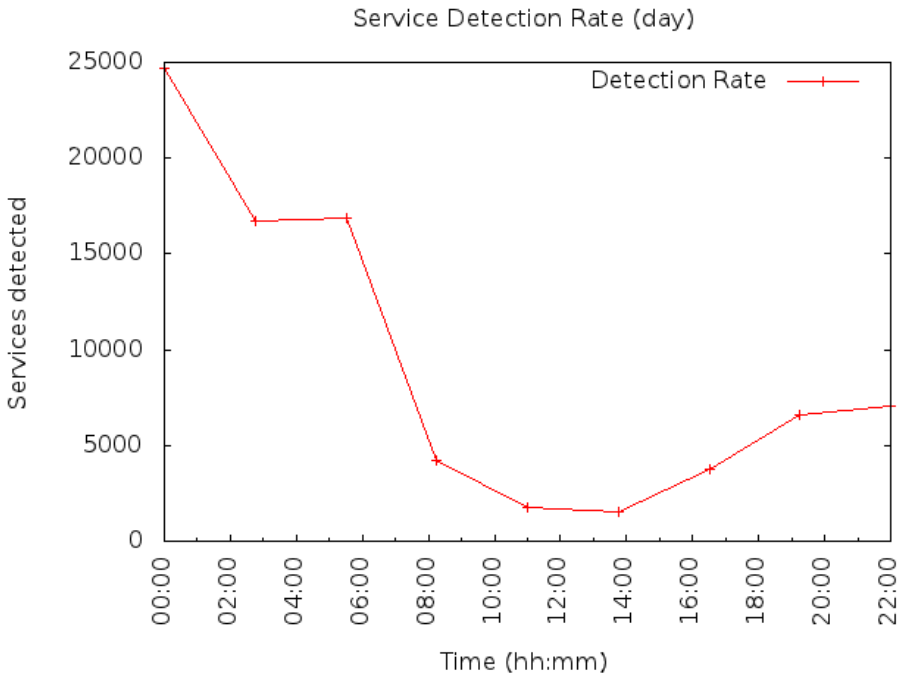

Figure 5.7: Service detection rate (day)

5.3.4 Operating System Detection Rate

Each host will only be registered with one operating system in the system created for this thesis, even though systems may for instance be running virtual operating systems that are using network address translation (NAT) to share one IP address. This limitation is probably not going to distort the results much, since the University has a policy against using NAT.

Figure 5.8 (page 70) shows the detection rate of operating systems every fifteen minutes over a period of three hours. The curve is even less steep in this case than in the host and service detection cases. This is because we have to wait for the hosts to connect to the update servers they are using to update their operating system. Only about fifteen percent of the operating systems detected that day, was detected within the first fifteen minutes.

Figure 5.9 shows the operating system detection rate for a full day. The measurements are made for every run made by Flow-store. It shows that around eighty percent of the operating systems detected that day was detected within the first couple of hours.

Amount	Operating System
13059	Unknown
4362	Windows
181	RedHat
35	Darwin
6	Ubuntu

Table 5.3: Distribution of detected operating systems

Table 5.3 shows the distribution of the different operating systems detected. Only twenty-six percent of the operating systems for the hosts was detected within the first day. When looking at the graphs it does not look like this will improve much by processing more data. The detection rate can probably be increased by adding more update servers to the configuration file used by Flow-dump. The list of update servers used in this thesis is only a subset of all the update servers that exist.

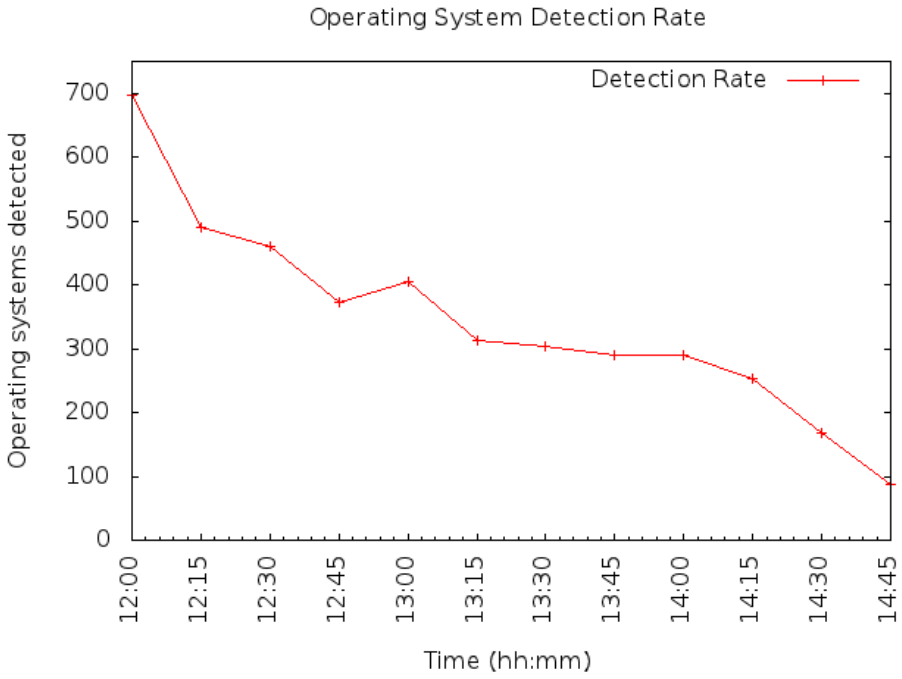

Figure 5.8: Operating system detection rate (short time span)

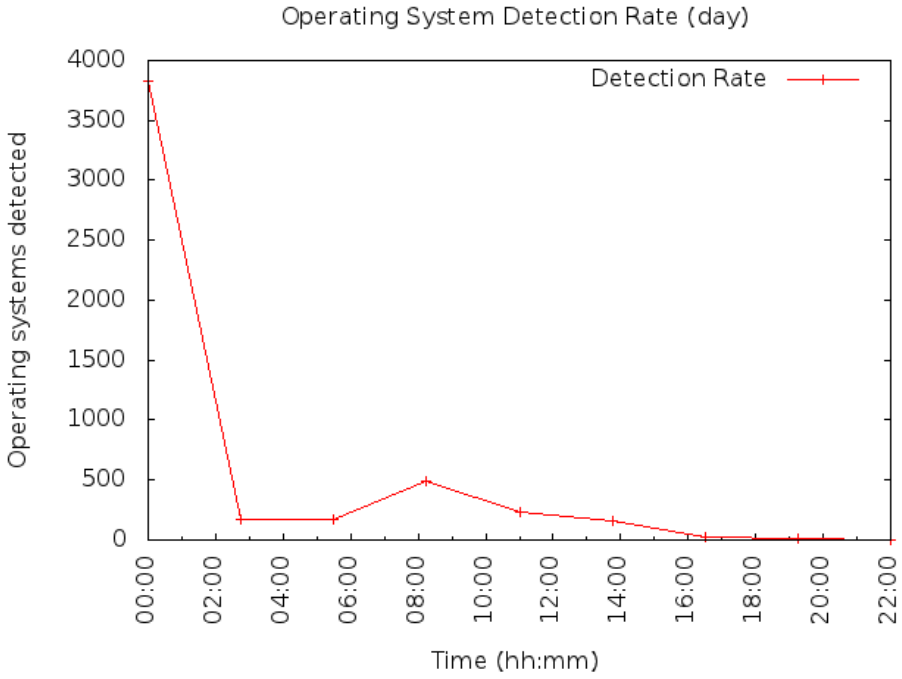

Figure 5.9: Operating system detection rate (day)

5.3.5 Success Rate

In addition to having a good detection rate, it is also important to verify that the assets detected by the implemented system are correct. Verifying the hosts and services detected would require us to have direct access to all the machines. Because of lack of access and time constraints, we chose to not do this. Verifying the operating systems detected are a whole lot easier, since the University is running Active Directory (AD), which contains records for almost all the machines on the University network running Windows.

Manually looking up machines in AD is laborious and it consumes lots of time. Therefore, we ended up writing *ad-check-os.pl*, a Perl program that goes through a list of all the hosts running Windows, search through AD, and checks if the hosts are in fact running Windows. The source code is available in Appendix B.5.

Total	Correct	False
4305	4220	85

Table 5.4: Results from running ad-check-os.pl

Table 5.4 shows the results from running *ad-check-os.pl* on the hosts detected by Flow-dump that claims to be running Windows. These are the same hosts used in 5.3.4, but the number of total hosts differ a little, since only hostnames are used in AD. Hosts were the hostname did not resolve was excluded from the results.

Of the 4305 hosts checked there are only 85 that according to AD is not running Windows. This gives us a success rate of 98.00 %.

Total	Correct	False
4305	4259	46

Table 5.5: Results after manually looking up hosts in Cerebrum

Luckily for us, the University of Oslo has records of all the hosts they manage in their administration system called Cerebrum [20]. This is mainly a system that is used for user management, but they also keep records of hosts they manage there. By manually looking up the 85 hosts that according to AD allegedly were not running Windows, we found out that about half of them (39 hosts) actually were (see table 5.5). Because of this,

the success rate of Flow-dump's Windows operating system detection is 98.93 %. This gives us a fault rate close to one percent, which means that the operating system detection has a low enough rate of false positives.

The first half of the false positives mainly consisted of network equipment and servers, like for instance gateways and PXE servers. The other half consisted of hosts running Darwin (Mac OS and iPhone OS). The most probable reason why so many hosts running Darwin was detected as hosts running Windows, is because they have Microsoft Office installed. Microsoft office uses the same address range as Windows update for their update servers, and it is therefore difficult to differentiate between them. It is also worth mentioning that Mac's are capable of running Windows, so some of these host may in fact be running Windows, but this is a more unlikely scenario.

Chapter 6

Discussion

In this chapter we discuss the benefits and drawbacks of using active and passive asset detection (section 6.1), and on-line and off-line asset detection (section 6.2). Then we discuss the advantages and disadvantages of using NetFlow for asset detection (section 6.3). Finally, we discuss legal concerns connected to asset detection (section 6.4).

6.1 Active or Passive Asset Detection

Passive asset detection tools like PRADS (section 2.6) and the system implemented in this thesis (chapter 4) sees all the traffic that passes through the network, if placed correctly, hence a host will be detected as long as it communicates with other hosts on the network.

Active asset detection tools like Nmap (section 2.5), on the other hand, only detects hosts and services they actively scan. To save time and resources, they also often only scan the most common ports, so services running on non standard ports are usually not detected by active scanning tools. Scanning all 65 535 ports for both TCP and UDP services takes too much time, especially if this has to be done on thousands of hosts.

Another drawback of using active asset detection tools is that they do not detect services that are filtered, either by the application or by a filtering gateway. Passive tools detect filtered services as long as these services are used and data is transmitted. Active tools often show that the service is filtered, but it is almost impossible to know if there is actually a service

running on the filtered port or not.

Active scanning is resource intensive, both for the host scanning the network and for the network, since it generates lots of network traffic. In addition to this, it is host intrusive. It tries to open up many TCP and UDP connections to the target host, making the target host use more resources than it usually does. It also triggers a lot of alarms in other systems, like intrusion detection systems.

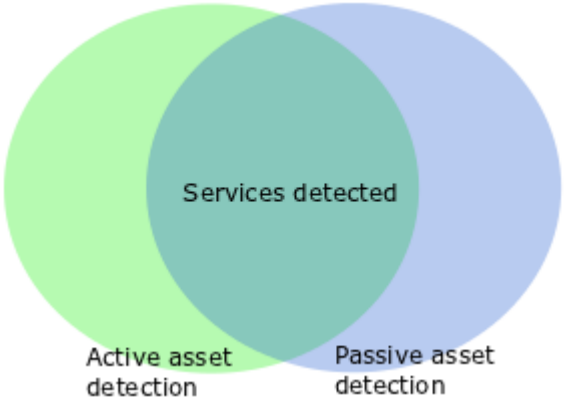

Figure 6.1: Venn diagram of service detection

Another disadvantage of using active scanning tools is that the result of the scan gets outdated very quickly. A network service could be started on the target host only minutes after the scan took place. By using a passive asset detection tool, the service would be detected as soon as the service starts sending data.

One downside to passive asset detection tools is that they only detect services that are sending data, hence idle network services will not be detected.

Because of the different advantages and disadvantages of using either active or passive asset detection, the detection result will vary. Figure 6.1 illustrates that active and passive asset detection detects mostly the same services, but active tools picks up a few services passive tools misses, and the other way around.

6.2 On-line or Off-line Asset Detection

By on-line asset detection, we mean passive tools that process the network traffic on the fly, thus not storing the network traffic before processing it. Off-line asset detection, on the other hand, is done by processing already stored data. The system implemented for this thesis does off-line asset detection. PRADS has the capability to do both on-line and off-line asset detection as long as the network traffic data is stored.

Off-line asset detection using *full* network traffic data is difficult to do, due to the amount of traffic data collected. This is possible to do on networks with low network throughput, but it does not scale well on high traffic networks. At one point in the network we saw 4.6 Tbit/s network throughput, which means we would have to store 575 GB per second. This is not possible to do with the hard disk technology that exists today.

If we use a more modest example, like a network with a throughput of 1 Gbit/s, we would have to store 125 MB per second. This is still a lot, but at least it is feasible to do. In this case we would have to store 450 GB per hour. This is possible but it would require lots of disc capacity and cost lots of money.

Compression could be used to save hard disk space. However, it is difficult to find a compression tool with a good compression ratio and low enough time usage. Table 6.1 shows a test we did with common Unix compression tools. It shows the compression ratio and time used from a compression test done on a 252 MB network dump file using several available tools.

Tool	Time	Ratio	Speed
gzip (fast)	5.2s	3.15	48.46 MB/s
gzip (best)	14.6s	3.36	17.26 MB/s
bzip2 (fast)	22.6s	3.32	11.15 MB/s
bzip2 (best)	22.9s	3.67	11.00 MB/s
lrzip (LZO compression)	14.3s	6.33	17.62 MB/s
lrzip (level 1)	20.2s	7.49	12.48 MB/s
lrzip (level 7)	37.2s	7.64	6.77 MB/s

Table 6.1: Compression ratio, time and speed of common compression tools

As seen in table 6.1, the compression tool with the best time versus compression ratio is lrzip with LZO compression. However, it is not fast enough to use for any of the examples mentioned above. It can only

compress 17.62 MB per second, meaning that it can handle network traffic speeds up to 141 Mbits per second. Because of the high compression ratio the 252 MB file is compressed down to 40 MB.

The fastest compression tool tested is gzip with the *–fast* flags set. It does not have as high compression ratio as lrzip (LZO compression) but it several times faster. Gzip (fast) is able to compress 48.46 MB per second. Thus, being able to handle network speeds up to 387.7 Mbits per second. This is not fast enough to handle the traffic amounts in the examples above, but it is fast enough to handle the network throughput for a reasonably large computer network.

Off-line asset detection is easier to do if only the necessary traffic information is stored. NetFlow does this by only storing IP header information, and grouping the traffic together into flows. This, combined with Nfdump tools binary format for storing NetFlow data, makes us able to store months and even years of network traffic data. However, this comes at the cost of not having the entire IP packet available for analysis.

6.3 Using NetFlow for Asset Detection

In the following sections we discuss the advantages and disadvantages of using NetFlow for asset detection.

6.3.1 Advantages

One of the biggest advantages of using NetFlow data for asset detection is that many companies already collect NetFlow data for other purposes. This makes it easy to get more value out of already collected data. An other advantage is that most Cisco equipment in addition to several other vendors support collecting NetFlow data (see section 2.1.2). Therefore, it is simple for companies with Cisco switches and routers to start collecting NetFlow data.

The system implemented for this thesis is a passive asset detection system so it process all the traffic that flows through the network. NetFlow has the benefit of storing all the flow data collected on the network centrally, and at the same time collecting the flows on de-centralized nodes spread

around on the network. NetFlow lets the switches and routers collect the NetFlow data, which is a huge benefit since no extra equipment than the core network equipment is needed. This makes NetFlow a cost effective solution compared to other alternatives.

Because NetFlow collects flow data in a de-centralized manor and aggregate the network traffic, it is capable of handling large amounts of network traffic, and we saw in the experiments that NetFlow scales up to even terabits of network traffic per second. Other passive asset detection systems like PRADS would have a hard time analyzing high traffic networks like this, since it would have to be able to read network traffic at this pace. Since no network cards support network speeds this high, the traffic would have to be divided amongst several servers running PRADS, making this an expensive alternative to NetFlow, and even impossible in some cases.

Source	**Number of Flows**	**Traffic Amount**
uio-gateway1	2487715	147.4 TB
uio-gateway2	1521585	156.2 GB
uio-gateway3	1324694	44.4 GB

Table 6.2: Statistics gathered by Nfdump tools for a five minute interval

Another advantage of NetFlow, is that the amount of network traffic flowing through a switch does not affect the amount of flows in a linear way. Table 6.2 show that even though almost more than a thousand times more traffic flows through *uio-gateway1* than *uio-gateway2*, the amount of flows is not that much higher. The amount of flows for the selected period is only almost twice as large. Because of this, the system implemented for the thesis have no problems handling traffic speeds that exceed terabits per second.

The traffic stored to disk by Nfdump, makes it possible to easily go back and check how the network looked at a certain point in time, due to the low disk usage by Nfdump tools. Even though 147.4 TB of data flowed through *uio-gateway1* over a five minute time period, the captured amount of NetFlow data that were stored to disk was only 52 MB. Hence, by using NetFlow we do not require much disk space to store months, or even years of NetFlow data.

The high detection rate shown in the evaluation (section 5) is also an advantage. With a little tweaking, by adding more services and operating

system update servers, we could probably get the detection rate even higher. The system implemented detects both TCP and UDP services. PRADS only detect one UDP service, DNS running on port 53.

6.3.2 Disadvantages

One disadvantage with NetFlow is that it stores filtered traffic, even though it is dropped by the switch or router. We have solved this issue in the system implemented by only looking at the flows going from the server to the client, so this is not a problem in our case. By doing this, we avoid the problem with clients generating false positives by trying to contact services that does not exist.

The largest disadvantage of using NetFlow data for asset detection is that we do not have access to the raw network traffic data, but only a limited subset of the total data. This limits what we are able to detect and the methods we can use to detect network assets.

Because of the limited amount of data fields available in NetFlow version 5, we do not get the MAC address of the hosts in the flows. Therefore, we have to use IP addresses to identify hosts on the network. This is far from optimal, since one host can have one IP address one day and another next day. It would be better to use MAC addresses as a unique identifier for the hosts, since MAC addresses are supposed to be unique. MAC spoofing is possible, but it is rare that someone does it, and it is therefore not important to us. NetFlow version 9 has the capability to add a source and destination MAC address data fields to the flows.

Another limitation to NetFlow in asset detection, is that TCP flags are grouped together. Because of this, most of the service detection methods mentioned in section 2.3 can not be used, thus forcing us to use port numbers to detect services instead. This is not easy to do and the amount of false positives are higher than if we could use TCP flags.

Because we do not have access to the raw network traffic data, we can not inspect the traffic to identify which services that are actually running on the hosts. In the system implemented for this thesis we have to rely on ports. Therefore, if the service runs on a non standard port, we will identify the service as unknown or as the service that runs on that port by default. We would detect the service, but the service name and description would

potentially be wrong. In addition to this we would not be able to detect the version of the service running, unlike PRADS, which is capable of doing this.

6.4 Legal Concerns

The legal concerns connected to asset detection methods like port scanning, and even to asset detection in general are complicated. The laws vary from country to country and there are few legal cases available to set legal precedent for port scanning.

One example of a case is a 17 year old Finnish boy who was convicted of attempted computer intrusion for port scanning a bank [9]. The supreme court stated in its ruling that the defendant had systematically carried out port scanning operations to gather information for the purpose of unauthorised break in to the bank's computer network. This was in the verdict defined as an attempted computer break in. The defendant had to pay a fine for telecommunications interference and attempted computer break in. He also had to compensate the bank for the amount the investigation had cost.

Here in Norway we had a case in 1998, where the University of Oslo (UiO) accused Norman Data Defence Systems (NDDS) of trying to break into their systems [21]. NDDS had port scanned UiO's computer network without permission and tried to log in to one of UiO's machines using telnet with a guest account. The reason supplied by NDDS for doing this was that they were going to show on a TV documentary aired by NRK (the Norwegian state television channel) that the Internet is a dangerous place if precautions are not made.

NDDS was charged with breaking the Norwegian penal code paragraph 145, that states that it is illegal to break protections to get access to data that are stored or that can be transfered electronically. UiO stated that they used 25-30 working hours on investigating the security breach.

The Norwegian supreme court concluded that none of the programs that were used against UiO's computer network would have given access to protected data. One of the judges stated that by connecting a computer to the Internet the owner of the computer has accepted that others ask the

computer which services it has to offer, and is therefore not looked upon as unsolicited use of the machine. NDDS ended up being acquitted for all the accusations.

In Norway IP addresses are considered "personal" information, hence is governed by privacy laws. This may be changing as a EU court now deemed IP addresses "non personal" which could become a precedence. Some voice concern about the visibility of services at each host, exposing personal data usage habits. However, most operational and security factors overrule this concern.

When performing a port scan on someone else's computer network it is best to always get a written consent beforehand. If this is done it is easier to avoid ending up spending all your money and time in court.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The main goal of this project was to implement a passive asset detection system that detects assets on a computer network using NetFlow data. We have successfully accomplished this goal.

The evaluation shows that the implemented system is able to run in real-time, even with good margins opening for adding additional rules and functionality.

The system has a good detection rate and is therefore able to detect most of the hosts on the network, and the network services they are running within a short period of time. Given enough time, the system should be able to detect all the services running on all the hosts on the network it is monitoring. In addition to this we verified the results from the operating system detection rule used by Flow-dump, and it had just above one percent false positives.

As discussed in section 6.3.2 there are disadvantages to using NetFlow for this purpose. By having direct access to the network traffic as PRADS do, we are able to detect versions of the services running. It is also easier to get a more correct guess of which operating system that are running on the hosts. These are things that are not so easy to do when only NetFlow data are available. However, we have tried to minimize these limitations by guessing operating systems based on connections made to operating system update servers. The results from this looks promising, but it needs

more adjustments to be of any real use.

We have also tested the scalability of using NetFlow for asset detection. The system implemented had no problems running on a network with a throughput of 4.6 Tbit/s. Because of this, we can conclude that the system is scalable enough to use on high speed networks. A system like PRADS with direct access to the network traffic would not have been able to process traffic from networks with as high throughput as in this case.

We have also tested running the system on historic data, which only takes a little less than thirty minutes when distributed over several CPU cores. Therefore, it is possible to process weeks or even months of data without much hassle.

Flow-map can provide an easy and reasonable system for asset detection for network administrators. It can give a quick overview and an early warning about rouge services and hosts.

7.2 Future Work

Due to some time limitations, there are still some improvements that can be made to our system. In this section we list some of the improvements and future research that can be made to make asset detection using NetFlow even better.

7.2.1 IDS Correlation

It would be useful to test how well the system performs when the data produced is correlated with an Intrusion Detection System like Snort. It could be used to escalate the degree of seriousness on alerts that relates to services that are actually running on hosts on the network. If a snort alarm regarding a web vulnerability hits a host running a web server, then it is worth investigating more closely.

7.2.2 Detect Software Running on Hosts

In this thesis we tried to guess which operating systems the different hosts were using based on the update servers they contacted. This could also

be used to detect other software running on the different hosts, if a host is contacting a Java update server, it is probably running Java.

7.2.3 Operating System Detection Based on Services

In addition to detecting operating systems based on connections made to update servers, we could also use services running on hosts to guess which operating system they are running. For instance if a host is running the service *Microsoft RPC services* on port 135, then it is probably running some version of Microsoft Windows.

7.2.4 Rules Engine

By writing a rules engine for the system, it would be easier to implement new asset detection rules. The rules engine could use an approach similar to the one used in snort rules. By using a rules engine, the host detection rule implemented in this thesis could have looked something like this:

host icmp $HOME_NET 0 -> any 0.0

7.2.5 NetFlow version 9

In this thesis we did not get the possibility to test the full extents of the benefits that using NetFlow version 9 would bring. NetFlow version 9 adds more possible data fields, and it includes IP version 6 traffic. It is worth investigating this closer.

7.2.6 Evaluation

Because of time constraints we did not get to fully test how many false positives and false negatives the system has when detecting assets. This is a cumbersome task because it requires us to manually check which services that are running on the hosts, and this requires access to all the systems. The alternative is to do a port scan of the hosts, which may not be accurate because of filtered ports.

Bibliography

[1] Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley Professional, 2003.

[2] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004.

[3] Introduction to cisco ios netflow. `http://www.cisco.com/go/netflow`. Accessed: 2011-10-25.

[4] Netflow data export. `http://www.cisco.com/en/US/docs/switches/lan/catalyst6500/ios/12.2SXF/native/configuration/guide/nde.pdf`. Accessed: 2011-12-22.

[5] Nfdump tools. `http://nfdump.sourceforge.net`. Accessed: 2011-11-10.

[6] Flow-tools. `http://www.splintered.net/sw/flow-tools`. Accessed: 2011-12-14.

[7] Lawrence Teo. Port scans and ping sweeps explained. *Linux J.*, 2000, November 2000.

[8] Internet control message protocol rfc. `http://tools.ietf.org/html/rfc792`. Accessed: 2011-11-21.

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

[10] Address resolution protocol (arp). `http://linux-ip.net/html/ether-arp.html`. Accessed: 2011-12-28.

[11] Raw ip networking faq. `http://www.ntua.gr/rin/rawfaq.html`. Accessed: 2012-10-28.

[12] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093.

[13] Ryan Spangler. Analysis of remote active operating system fingerprinting tools. http://www.packetwatch.net>. http://www.packetwatch.net/documents/papers/osdetection.pdf, 2003.

[14] Xprobe - active os fingerprinting tool. http://sourceforge.net/projects/xprobe. Accessed: 2011-01-03.

[15] Passive real-time asset detection system. http://gamelinux.github.com/prads/. Accessed: 2012-01-02.

[16] Linux programmer's manual, december 2008. FIFO(7).

[17] Cisco catalyst 6500 series flaws. http://www.cisco.com/en/US/products/hw/switches/ps708/products_configuration_example09186a0080721701.shtml#switched. Accessed: 2011-11-05.

[18] Sqlite, sqlite is self-contained. http://www.sqlite.org/selfcontained.html. Accessed: 2011-11-28.

[19] Nist - prefixes for binary multiples. http://physics.nist.gov/cuu/Units/binary.html. Accessed: 2011-12-19.

[20] Cerebrum. http://sourceforge.net/projects/cerebrum/. Accessed: 2012-01-28.

[21] Norwegian supreme court, norman vs uio. http://heim.ifi.uio.no/gisle/local/law/norman2_hak_hr.html. 1998-12-15.

Appendix A

Abbreviations

ACK	Acknowledgement
ARP	Address Resolution Protocol
BSD	Berkeley Software Distribution
CLI	Command Line Interface
CSV	Comma-Sepparated Values
DOS	Denial of Service
DNS	Domain Name Service
FIFO	First in, first out
Gb	Gigabit
GB	Gigabyte
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
ISP	Internet Service Provider
LAN	Local Area Network
Mb	Megabit
MB	Megabyte
NAT	Network Address Translation
NDDS	Norman Data Defence Systems
P.M.	Post Meridiem
PRADS	Passive Real-time Asset Detection System
RST	Reset
STDOUT	Standard Output Stream
SYN	Synchronize
Tb	Terabit
TB	Terabyte
TOS	Type of Service
UiO	University of Oslo
USIT	University Center for Information Technology

Appendix B

Source Code

This chapter contains the source code for the passive asset detection system implemented in this thesis. To run it a Perl interpreter and Nfdump tools must be installed.

B.1 flow-dump

```
1  #!/usr/bin/perl -w
2  #
3  # Flow-dump - Passive Asset Detection using Netflow
4  # Copyright (C) 2011-2012  Mats Klepsland <matsekl@ifi.uio.no>
5  #
6
7  use strict;
8  use Getopt::Long;
9  use Time::Local;
10 use NetAddr::IP;
11 require 'config.pl';
12
13 # Command line options
14 my $help = '';
15 my $read = '';
16 my $output = '';
17 my $write = '';
18 GetOptions ('usage|help|h|?' => \$help, 'read|r=s' => \$read,
19         'write|w=s' => \$write, 'output|o=s' => \$output);
20
21 # Declare variables from config file
22 our $debug;           # enable/disable debugging
23 our $filepath;        # path to netflow files
24 our @flowsources;     # flow sources to monitor
25 our @networks;        # networks to monitor
26 our $outputmode;      # program output mode
27 our @blacklist;       # blacklist known network scanners
28 our $windowsupdate;   # path to file containing Windows Update servers
29 our @updateservers;   # hash containing OS update servers
30 our @port_blacklist; # blacklist ports generating false positives
31
32 # Prepare array containing networks to monitor
33 my @home_net = ();
34
```

```
35  foreach (@networks)
36  {
37    push (@home_net, NetAddr::IP->new($_));
38  }
39
40  # Prepare array containing OS update servers
41  my @os_update_servers = ();
42
43  foreach (@updateservers)
44  {
45    my @server = split(',', $_);
46    $server[0] = NetAddr::IP->new($server[0]);
47    push @os_update_servers, [@server];
48  }
49
50  # Prepare list of update servers to use for OS detection
51  my @windowsupdateservers = ();
52  fetch_windows_update_server_list();
53
54  # Overwrite output mode if --output is specified
55  if ($output) {
56    $outputmode = $output;
57  }
58
59  # Print usage
60  if ($help) {
61    usage();
62    exit;
63  }
64
65  # Initialize array to store NetFlow data in
66  my @data = ();
67
68  # Read file and exit if --read is specified
69  if ($read) {
70    if (!exist($read)) {
71      die "Could not read file: $!\n";
72    }
73    @data = readfile($read);
74    parse(@data);
75    exit;
76  }
77
78  # Otherwise loop through all Nfdump files starting from now
79  my ($min, $hour, $day, $month, $year) = (localtime)[1,2,3,4,5];
80  $min -= ($min % 5);  # round time to closest five minutes
81
82  # Loop forever
83  while (1)
84  {
85    foreach (@flowsources)
86    {
87      my $file = get_file_path($_);
88
89      while (!exist($file))
90      {
91        print "Waiting for file $file. Sleeping.\n" unless !$debug;
92        sleep 5;  # sleep five seconds
93      }
94
95      print "Parsing file $file\n" unless !$debug;
96      @data = readfile($file);
97      parse(@data);
98      @data = ();
99    }
100
101   fetch_windows_update_server_list();
102   increment_five_minutes();
103 }
104
105
```

```
106  # Fetch list of Windows Update servers for OS detection
107  # Used by old OS detection.
108  sub fetch_windows_update_server_list
109  {
110    @windowsupdateservers = ();
111
112    open (WINDOWS, $windowsupdate) or die "Could not read file: $!\n";
113    while (<WINDOWS>)
114    {
115      # Convert IP address to number to make it easier to compare
116      chomp (my $ip = $_);
117      my @octets = split(/\./, $ip);
118      my $ip_address = ($octets[0]*1<<24)+($octets[1]*1<<16)+($octets[2]*1<<8)+($octets[3]);
119
120      push (@windowsupdateservers, $ip_address);
121    }
122    close (WINDOWS);
123  }
124
125
126  # Get file path of next Nfdump file to read
127  sub get_file_path
128  {
129    my ($src) = @_;  # name of flow source
130    my ($mi, $ho, $da, $mo, $ye, $file);
131    $mi = pad_number($min);
132    $ho = pad_number($hour);
133    $da = pad_number($day);
134    $mo = pad_number($month + 1);
135    $ye = $year + 1900;
136
137    $file = "$filepath/$src/$ye/$mo/$da/nfcapd.$ye$mo$da$ho$mi";
138    return $file;
139  }
140
141
142  # Pad numbers smaller than ten with zero
143  sub pad_number
144  {
145    my ($num) = @_;
146    if ($num < 10) {
147      $num = 0 . $num;
148    }
149    return $num;
150  }
151
152
153  # Increment time five minutes
154  sub increment_five_minutes
155  {
156    my $next = timelocal(0, $min, $hour, $day, $month, $year);
157    $next += 320;  # 5 * 60 sec
158    ($min, $hour, $day, $month, $year) = (localtime($next))[1,2,3,4,5];
159  }
160
161
162  # Check if file exist
163  sub exist
164  {
165    my ($file) = @_;
166    if (!-e $file) {
167      return 0;
168    }
169    return 1;
170  }
171
172
173  # Read Nfdump file
174  sub readfile
175  {
176    my ($file) = @_;
```

```
177
178   # About 300 percent faster to select the fields we need instead
179   # of using the csv output format
180   my $output = 'fmt:%ts,%sa,%da,%sp,%dp,%pr,%flg,%ipkt,%ibyt';
181   my @data = qx/nfdump -r $file -o $output 'src net $networks[0]'/;
182   return @data;
183 }
184
185
186 # Parse array containing netflow data
187 sub parse
188 {
189   my (@data) = @_;
190   shift(@data);      # remove first line containing meta data from result
191   splice @data, -4; # remove last four lines containing meta data from result
192
193   foreach (@data)
194   {
195     # Trim whitespace added to make things look nice
196     my $entry = join ' ', split ' ', $_;
197     $entry =~ s/,\s+/,/g;
198     $entry =~ s/\s+,/,/g;
199
200     # Split comma-separated values into variables
201     my ($time, $srcip, $dstip, $srcport, $dstport, $protocol, $tcpflags,
202         $packets, $bytes) = split(',', $entry);
203
204     # Host detection based on ICMP
205     host_detection($time, $srcip, $dstport, $protocol);
206
207     # Service detection based on ports
208     service_detection_ports($time, $srcip, $dstip, $srcport, $dstport, $protocol);
209
210     # Service detection based on TCP flags
211     #service_detection_flags($time, $srcip, $dstip, $srcport, $dstport, $tcpflags, $protocol);
212
213     # OS detection
214     #windows_os_detection($time, $srcip, $dstip);
215     os_detection($time, $srcip, $dstip, $dstport);
216   }
217 }
218
219
220 # Check if IP address is on one of the subnets we are monitoring
221 sub within_home_net
222 {
223   my $ip = NetAddr::IP->new($_[0]);
224
225   foreach (@home_net)
226   {
227     if ($ip->within($_)) {
228       return 1;
229     }
230   }
231   return 0;
232 }
233
234
235 # Host detection
236 sub host_detection
237 {
238   my ($time, $srcip, $dstport, $protocol) = @_;
239
240   # Detect hosts based on ICMP echo reply
241   if ($protocol eq 'ICMP' and $dstport eq '0.0' and within_home_net($srcip)) {
242     output('Host', $srcip, $time);
243   }
244 }
245
246
247 # Service detection based on ports under 1024. Port scanning makes it
```

```
248 # hard to rely on this one. We only need to check one way, since flows are
249 # unidirectional. To avoid false positives from port scanning, we check the
250 # traffic coming from the server.
251 sub service_detection_ports
252 {
253   my ($time, $srcip, $dstip, $srcport, $dstport, $protocol) = @_;
254
255   # Check if client IP address is blacklisted as a network scanner
256   foreach (@blacklist)
257   {
258     if ($dstip eq $_) {
259       return; # return if contacted by blacklisted host
260     }
261   }
262
263   # No services is using ICMP, so return if this protocol is detected
264   if ($protocol eq 'ICMP') {
265     return;
266   }
267
268   # Detect ports under 1024
269   if ($srcport < 1024 and $dstport > 1024 and within_home_net($srcip)) {
270
271     # Check if port is blacklisted
272     foreach (@port_blacklist)
273     {
274       my ($dir, $port) = split(',', $_);
275       if (($dir eq 'src' and $srcport eq $port) or ($dir eq 'dst' and $dstport eq $port)) {
276         # We can still use this result for host detection
277         output('Host', $srcip, $time);
278         return;
279       }
280     }
281
282     output('Service', "$srcip,$srcport/$protocol", $time);
283   }
284 }
285
286
287 # Service detection based on TCP flags
288 # This won't work with Cisco HW that don't export TCP flags in netflow data,
289 # like for instance Catalyst 6500.
290 sub service_detection_flags
291 {
292   my ($time, $srcip, $dstip, $srcport, $dstport, $tcpflags, $protocol) = @_;
293
294   # Detect service based on SYN-ACK from server from TCP three-way-handshake
295   if ($protocol eq 'TCP' and $tcpflags eq '.A..S.' and within_home_net($srcip)) {
296     output('Service', "$srcip,$srcport/$protocol", $time);
297   }
298
299   # Detect service based on ACK from client from TCP three-way-handshake
300   if ($protocol eq 'TCP' and $tcpflags eq '....S.' and within_home_net($dstip)) {
301     output('Service', "$dstip,$dstport/$protocol", $time);
302   }
303 }
304
305
306 # Detect hosts using Windows operating system based on connections made
307 # to update servers. Old OS detection.
308 sub windows_os_detection
309 {
310   my ($time, $srcip, $dstip) = @_;
311
312   # Convert IP addresses to numbers to make them easier to compare
313   my @octets = split(/\./, $_[1]);
314   my $src = ($octets[0]*1<<24)+($octets[1]*1<<16)+($octets[2]*1<<8)+($octets[3]);
315   @octets = split(/\./, $_[2]);
316   my $dst = ($octets[0]*1<<24)+($octets[1]*1<<16)+($octets[2]*1<<8)+($octets[3]);
317
318   foreach (@windowsupdateservers)
```

```
319     {
320       if ($src == $_ and within_home_net($dstip)) {
321         output('OS', "$dstip,windows", $time);
322         return;
323       }
324       if ($dst == $_ and within_home_net($srcip)) {
325         output('OS', "$srcip,windows", $time);
326         return;
327       }
328     }
329   }
330
331
332   # Detect what OS hosts are using based on connections made to update
333   # servers.
334   sub os_detection
335   {
336     my ($time, $srcip, $dstip, $dstport) = @_;
337
338     # Saves a lot of time by only continuing if srcport is 80. This also helps us
339     # avoid a lot of false positives from name servers, mail servers, and similar
340     if ($dstport != 80) {
341       return;
342     }
343
344     # Prepare IP address for comparing
345     my $dst = NetAddr::IP->new($dstip);
346
347     foreach my $row (@os_update_servers)
348     {
349       # We only have to check one direction since there should be two flows
350       # per session anyway
351       if ($dst->within(@$row[0])) {
352         output('OS', "$srcip,@$row[1]", $time);
353       }
354     }
355   }
356
357
358   # Direct output to selected output source; FIFO, stdout or CSV
359   sub output
360   {
361     my ($type, $output, $time) = @_;
362     $time = substr($time, 0, -4);  # we don't need milliseconds
363
364     # Write to STDOUT
365     if (!$outputmode or uc $outputmode eq 'STDOUT') {
366       print "$time - $type detected - $output\n";
367     }
368
369     # Write to CSV file
370     elsif (uc $outputmode eq 'CSV') {
371       $write = 'assets.csv' unless $write;
372
373       open (CSV, '>>', $write) or die "Could not open file: $!\n";
374       print CSV "$type,$output,$time\n";
375     }
376
377     # Write to FIFO
378     elsif (uc $outputmode eq 'FIFO') {
379       $write = 'assets' unless $write;
380
381       # Create FIFO unless it already exist
382       unless (-p $write) {
383         unlink $write;
384         system ('mknod', $write, 'p') and die "Can't mknod $write: $!";
385       }
386
387       # Next line blocks until there is a reader
388       open (FIFO, ">", $write) or die "Can't write $write: $!";
389       print FIFO "$type,$output,$time";
```

```
390       close FIFO;
391       select(undef, undef, undef, 0.005);  # wait to avoid dup signals
392     }
393
394     else {
395       die 'Output mode not supported. Choose \'CSV\', \'FIFO\' or \'STDOUT\' instead.';
396     }
397   }
398
399
400   # Print usage
401   sub usage
402   {
403     print "Usage: $0 [OPTION]...\n";
404     print "Passive Asset Detection using Netflow.\n\n";
405     print "  -r, --read <file>    : netdump file to read\n";
406     print "  -o, --output <type> : overwrite output mode (csv/fifo/stdout)\n";
407     print "  -w, --write <file>   : write to file\n";
408     print "  -h, --help           : display this help and exit\n\n";
409     print "Write option only works for CSV and FIFO output modes.\n";
410   }
```

B.2 flow-store

```
1  #!/usr/bin/perl -w
2  #
3  # Flow-store - Store assets detected by Flow-dump in database
4  # Copyright (C) 2011-2012  Mats Klepsland <matsekl@ifi.uio.no>
5  #
6
7  use DBI;
8  use strict;
9  use Getopt::Long;
10
11 # Command line options
12 my $help  = '';
13 my $read  = '';
14 my $write = '';
15 GetOptions ('usage|help|h|?' => \$help, 'read|r=s' => \$read,
16             'write|w=s' => \$write);
17
18 # Variables used for statistics
19 my $num_host = 0;
20 my $num_service = 0;
21 my $num_os = 0;
22
23 # Turn debug messages on/off
24 my $debug = 0;
25
26 # Print usage
27 if ($help) {
28   usage();
29   exit;
30 }
31
32 # Read from asssets.csv if --read is not specified
33 if (!$read) {
34   $read = "assets.csv";
35 }
36
37 # Write to assets.db if --write is not specified
38 if (!$write) {
39   $write = "assets.db";
40 }
41
42 # Connect to database
43 my $db = DBI->connect("dbi:SQLite:$write", "", "",
44 {RaiseError => 1, AutoCommit => 1});
45
46
47 # Remove duplicate entries from CSV file
48 my %seen = ();
49 {
50     local @ARGV = ($read);
51     local $^I = '.bac';
52     while(<>){
53        my @str = split(' ', $_);
54        $seen{$str[0]}++;
55        next if $seen{$str[0]} > 1;
56        print;
57     }
58 }
59
60
61 # Read CSV file and process content
62 open (ASSETS, $read) or die $!;
63 while (<ASSETS>)
64 {
65   chomp;
66   my @data = split(',', $_);
67
68   # Handle Host entries
```

```
69    if ($data[0] eq 'Host') {
70      add_host($data[1], '', $data[2]);
71    }
72
73    # Handle OS entries
74    if ($data[0] eq 'OS') {
75      add_host($data[1], $data[2], $data[3]);
76    }
77
78    # Handle Service entries
79    if ($data[0] eq 'Service') {
80      add_host($data[1], '', $data[3]);
81      add_service($data[1], $data[2]);
82    }
83  }
84  close (ASSETS);
85
86
87  # Print statistics
88  if ($debug) {
89    print "Database Changes\n----------------\n";
90    print "Host: " . $num_host . "\n" . "OS: " . $num_os . "\n";
91    print "Service: " . $num_service . "\n";
92  }
93
94
95  # Add host to database
96  sub add_host
97  {
98    my ($ip, $os, $date) = @_;
99    my $sql;
100   my $rows_affected;
101
102   if (!$os) {
103     $sql = <<SQL;
104 INSERT OR IGNORE INTO host (IP, first_seen, last_seen)
105 VALUES ('$ip', '$date', '$date');
106 SQL
107   } else {
108     $sql = <<SQL;
109 INSERT OR IGNORE INTO host (IP, OS, first_seen, last_seen)
110 VALUES ('$ip', '$os', '$date', '$date');
111 SQL
112
113   }
114
115   $rows_affected = $db->do($sql);
116
117   # Update if insert failed
118   if ($rows_affected != 1) {
119     if ($os) {
120       $sql = "UPDATE host SET last_seen = '$date' AND OS = '$os' WHERE IP = '$ip'";
121     } else {
122       $sql = "UPDATE host SET last_seen = '$date' WHERE IP = '$ip'";
123     }
124     $rows_affected = $db->do($sql);
125   }
126
127   # Collect statistics
128   if ($rows_affected == 1) {
129     if ($os) {
130       ++$num_os;
131     } else {
132       ++$num_host;
133     }
134   }
135 }
136
137
138 # Add service to a host in the database
139 sub add_service
```

```
140 {
141   my ($ip, $port) = @_;
142   my $rows_affected;
143   my $sql = <<SQL;
144 INSERT OR IGNORE INTO service (port, ip)
145 VALUES ('$port', '$ip')
146 SQL
147
148   $rows_affected = $db->do($sql);
149
150   # Collect statistics
151   if ($rows_affected == 1) {
152     ++$num_service;
153   }
154 }
155
156
157 # Print usage
158 sub usage
159 {
160   print "Usage: $0 [OPTION]..\n";
161   print "Passive Asset Detection using Netflow.\n\n";
162   print "  -r, --read <file>       : read from specified CSV file\n";
163   print "  -w, --write <database> : write to specified database\n";
164   print "  -h, --help              : display this help and exit\n\n";
165 }
```

B.3 flow-map

```
1  #!/usr/bin/perl -w
2  #
3  # Flow-map - Present assets stored in database
4  # Copyright (C) 2011-2012  Mats Klepsland <matsekl@ifi.uio.no>
5  #
6
7  use DBI;
8  use strict;
9  use Getopt::Long;
10 use Socket;
11
12 my $version = '1.0';
13
14 # File containing service information
15 my $service_file = 'nmap-services';
16
17 # Command line options
18 my $help   = '';
19 my $list   = '';
20 my $stats  = '';
21 my $read   = '';
22 my $target = '';
23 my $num    = '';
24 GetOptions ('usage|help|h|?' => \$help, 'list|l' => \$list,
25           'stats|s' => \$stats, 'read|r=s' => \$read,
26           'target|t=s' => \$target, 'num|n=i' => \$num);
27
28 # Read from asssets.db if --read is not specified
29 if (!$read) {
30   $read = "assets.db";
31 }
32
33 # Default to 10 top services to show in summary
34 if (!$num) {
35   $num = 10;
36 }
37
38 # Connect to database
39 my $db = DBI->connect("dbi:SQLite:$read", "", "",
40 {RaiseError => 1, AutoCommit => 1});
41
42 # Print usage
43 if ($help) {
44   usage();
45   exit;
46 }
47
48 # Print stats
49 if ($stats) {
50   print_stats();
51   exit;
52 }
53
54 # Print usage if no target is specified
55 if (!$target) {
56   print "You must specify a target.\n\n";
57   usage();
58   exit;
59 }
60
61 # Allow hostname as target
62 unless($target =~ m/^(\d\d?\d?).*$/) {
63   my @host = (gethostbyname($target))[4];
64   if (@host) {
65     $target = inet_ntoa($host[0]);
66   } else {
67     print "Error. Could not resolve hostname.\n";
68     exit;
```

```
69    }
70  }
71
72  print "Starting Flow-map " . $version . " at " . localtime(time) . "\n";
73
74  my $sql;
75
76  # If a complete IP address is provided we only match one host
77  if ($target =~ m/\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}/) {
78    $sql = "SELECT * FROM host WHERE IP = '$target'";
79  } else {
80    $sql = "SELECT * FROM host WHERE IP like '%$target%' ORDER BY IP";
81  }
82
83  my $targets = $db->selectall_arrayref($sql);
84  my $count = 0;
85
86  foreach my $row (@$targets)
87  {
88    my ($ip, $os, $first, $last) = @$row;
89
90    # Resolve hostname
91    my $host = gethostbyaddr(inet_aton($ip), AF_INET);
92
93    # Print host
94    print "Host: " . $ip;
95    print " (" . $host . ")" unless !$host;
96    print "\n";
97
98    # Print services if list option is not specified
99    if (!$list) {
100     print "First seen: " . $first . "\n";
101     print "Last seen:  " . $last . "\n";
102     print "OS: " . $os . "\n" unless !$os;
103     print_services($ip);
104     print "\n";
105   }
106
107   ++$count;
108 }
109
110 print "Flow-map done. Found $count hosts.\n";
111
112
113 # Print services for a specific host
114 sub print_services
115 {
116   my ($ip) = @_;
117   my $sql = "SELECT * FROM service WHERE ip = '$ip'";
118   my $services = $db->selectall_arrayref($sql);
119
120   printf "%-10s%-20s%s\n", "PORT", "SERVICE", "DESCRIPTION";
121
122   foreach my $service (@$services)
123   {
124     my ($port, $ip) = @$service;
125
126     match_service($port);
127   }
128 }
129
130
131 # Find service name and description based on port
132 sub match_service
133 {
134   my ($port, $count) = @_;
135
136   open (DATA, $service_file) or die $!;
137   my @list = grep /\b$port\b/i, <DATA>;
138
139   if (@list) {
```

```
140       my @desc = split('# ', $list[0]);
141       my @name = split(' ', $list[0]);
142
143       my $name = $name[0] ? $name[0] : " ";
144       my $desc = $desc[1] ? $desc[1] : " \n";
145       my $count = $count ? $count : "";
146
147       printf "%-9s", $count unless !$count;
148       printf "%-10s%-20s%s", $port, $name, $desc;
149     } else {
150       # Only print port number and unknown if we can't find the service
151       printf "%-9s", $count unless !$count;
152       printf "%-10s%-20s\n", $port, "unknown";
153     }
154   }
155
156
157   # Print database statistics
158   sub print_stats
159   {
160     my $subnet = $target ? $target : "*";
161
162     my ($hosts) = $db->selectrow_array("SELECT COUNT(*) FROM host WHERE IP like '%$target%'");
163     my ($services) = $db->selectrow_array("SELECT COUNT(*) FROM service WHERE IP like '%$target%'");
164
165     print "--- Flow-map " . $version . " statistics ---\n";
166     printf "%-22s %d\n", "Hosts in database:", $hosts;
167     printf "%-22s %d\n\n", "Services in database:", $services;
168
169     print "---- Top Services ----\n\n";
170     printf "%-9s%-10s%-20s%s\n", "AMOUNT", "PORT", "SERVICE", "DESCRIPTION";
171
172     # Get the ten most common ports
173     my $sql = <<SQL;
174   SELECT port, COUNT(port) FROM service
175   WHERE IP like '%$target%'
176   GROUP BY port HAVING COUNT(port) > 1
177   ORDER BY COUNT(port) DESC LIMIT $num
178   SQL
179
180     my $topten = $db->selectall_arrayref($sql);
181
182     foreach my $row (@$topten)
183     {
184       my ($port, $count) = @$row;
185       match_service($port, $count);
186     }
187   }
188
189
190   # Print usage
191   sub usage
192   {
193     print "Usage: $0 [OPTION]..\n";
194     print "Passive Asset Detection using Netflow.\n\n";
195     print "  -t, --target <target> : target to display\n";
196     print "  -s, --stats           : summary of assets in database\n";
197     print "  -n, --num <number>    : number of top services to show in summary\n";
198     print "  -l, --list            : simple list of targets in given IP range\n";
199     print "  -r, --read <database> : read from specified database\n";
200     print "  -h, --help            : display this help and exit\n\n";
201     print "Target can be either a hostname, a single IP or an entire IP range.\n";
202   }
```

B.4 config.pl

```
1  # This is the configuration file for Flow-dump.
2
3  # Enable/disable debugging.
4  $debug = 1;  # on
5
6  # Path to Nfdump files.
7  $filepath = "/usit/nettflyt/flow/live";
8
9  # Flow sources to monitor.
10 @flowsources = ( "uio-gw8" );
11
12 # Define networks to monitor (subnet/netmask).
13 @networks = ( "129.240.0.0/16" );
14
15 # Select output mode. Supported modes are STDOUT, FIFO and CSV.
16 $outputmode = "CSV";
17
18 # Blacklist known port scanners.
19 @blacklist = ();
20
21 # File containing Windows Update IP addresses
22 $windowsupdate = "updateservers.csv";
23
24 # Define update servers for different Operating systems
25 @updateservers = (
26                    "129.240.2.25,RedHat",    # yum.uio.no
27                    "65.55.0.0/16,Windows",   # update.microsoft.com
28                    "129.240.12.27,Windows", # wsus.uio.no
29                    "17.250.248.95,Darwin",   # swscan.apple.com
30                    "129.241.93.37,Ubuntu"    # no.archive.ubuntu.com
31                  );
32
33 # Blacklist services that generate false positives
34 # Use 'dst' when you want to blacklist based on the port that the
35 # service is conntacted on (dstport). Use 'src' to blacklist a
36 # service in general
37 @port_blacklist = (
38                     "dst,4045", # blacklist lockd used by nfs
39                     "dst,2049", # blacklist portmap (nfs and nis)
40                     "dst,4046", # another port used by nfs
41                     "src,7"     # echo
42                   );
```

B.5 ad-check-os.pl

```
1  #!/usr/bin/perl -w
2  #
3  # ad-check-os.pl - Read a list of hosts and compare OS against Active Directory
4  # Copyright (C) 2011-2012  Mats Klepsland <matsekl@ifi.uio.no>
5  #
6
7  use strict;
8  use Net::LDAP;
9  use Socket;
10
11 my $ad;          # LDAP object
12 my $base;        # search base
13 my $results;     # search results
14 my $count = 0;   # number of entries returned
15 my $total = 0;   # number of hosts checked
16 my $true  = 0;   # number of hosts that acually run Windows
17 my $false = 0;   # number of false positives
18
19 my $file = $ARGV[0];
20
21 # Print usage if no arguments are supplied
22 if (!$ARGV[0]) {
23   print "Usage: $0 <file>\n";
24   print "Read list of hosts and compare OS against Active Directory.\n";
25   exit;
26 }
27
28 $ad = Net::LDAP->new("ldap://alexander.uio.no")
29           or die("Could not connect to LDAP server.");
30
31 # Bind to LDAP server
32 $ad->bind("username\@uio.no", password=>"password");
33
34 # Define the base
35 $base    = "DC=uio,DC=no";
36
37 # Read file and check OS of hosts
38 open (FILE, $file) or die $!;
39 while (<FILE>)
40 {
41   chomp;
42
43   # Resolve hostname
44   my $host = gethostbyaddr(inet_aton($_), AF_INET);
45
46   # Do not check if hostname does not resolve
47   if (!$host) {
48     next;
49   }
50
51   # Remove domain
52   my @hostname = split(/\./, $host);
53
54   if (compare($hostname[0])) {
55     ++$true;
56   } else {
57     ++$false;
58     print "False: $host\n";
59   }
60
61   ++$total;
62 }
63
64 # Print results
65 print  "+-------+---------+-------+\n";
66 print  "| Total | Correct | False |\n";
67 print  "+-------+---------+-------+\n";
68 printf "| %5s | %7s | %5s |\n", $total,$true,$false;
```

```
69  print  "+--------+----------+--------+\n";
70
71  # Cleaning up :)
72  close (FILE);
73  $ad->unbind;
74
75
76  # Compare
77  sub compare
78  {
79    my $filter = "(cn=$_[0])";
80
81    $results = $ad->search(base=>$base,filter=>$filter);
82    $count = $results->count;
83
84    # Loop through results
85    for (my $i=0; $i<$count; $i++) {
86
87      my $entry = $results->entry($i);
88      my $os_string = $entry->get_value('operatingSystem');
89
90      if (!$os_string) {
91        next;
92      }
93
94      my @os = split(" ", $os_string);
95
96      # Check for Windows operating system
97      if ($os[0] eq 'Windows') {
98        return 1;
99      }
100   }
101   return 0;
102 }
```

Appendix C

Shell Scripts

This chapter contains shell scripts written during the thesis. It includes a script for creating the database used by Flow-store and Flow-map, a script for processing a day of flows, and a script for maintaining a list of Windows update servers.

C.1 create_db.sh

```
1  #!/bin/bash
2  # Create database used by flow-store to store asset data
3  # Copyright (C) 2011-2012  Mats Klepsland <matsekl@ifi.uio.no>
4
5  sqlite3 assets.db "CREATE TABLE host (
6                       IP TEXT PRIMARY KEY,
7                       OS TEXT,
8                       first_seen TEXT,
9                       last_seen TEXT
10                    );"
11
12 sqlite3 assets.db "CREATE TABLE service (
13                      port TEXT,
14                      IP TEXT,
15                      PRIMARY KEY(port, IP)
16                    );"
```

C.2 runwholeday.sh

```
1  #!/bin/bash
2  # Simple script to run through an entire day worth of flows
3  # Copyright (C) 2011-2012  Mats Klepsland <matsekl@ifi.uio.no>
4
5  CONCURRENT=11  # number of processes to run at the same time
6  count=0        # counter to keep track of the number of processes
7  run_number=0   # number of times runned since last Flow-store
8
9  # Print usage if no argument is provided
10 if [ -z $1 ]; then
11   echo "Usage: $0 <FOLDER>"
12   exit
13 fi
14
15 # Loop through all files in folder
16 for flowfile in `ls $1`
17 do
18   # Wait if too many processes are running
19   if [ "$count" -eq "$CONCURRENT" ]
20   then
21     wait $!
22     count=0
23     run_number=$((run_number+1))  # increment run number
24
25     # Rotate assets file and clean up
26     if [ $run_number -ge 3 ]
27     then
28       mv assets.csv assets.csv.store
29
30       ./flow-store -r assets.csv.store &
31       time=`date +'%F %H:%M'`
32       echo "$time - Rotating assets file and running flow-store"
33
34       run_number=0
35     fi
36   fi
37
38   time=`date +'%F %H:%M'`
39   echo "$time - Parsing file: $1$flowfile"
40   ./flow-dump -r $1/$flowfile &
41
42   count=$((count+1))  # increment counter
43 done
44
45 wait $!
46 time=`date +'%F %H:%M'`
47 echo "$time - Running flow-store one last time, and cleaning up"
48 ./flow-store
49 rm assets.csv assets.csv.bac assets.csv.store assets.csv.store.bac
50 echo "Finished."
```

C.3 winupdate.sh

```
1  #!/bin/bash
2  # Maintain a list of the fifteen last IP addresses used by Windows Update
3  # Copyright (C) 2011-2012  Mats Klepsland <matsekl@ifi.uio.no>
4  #
5  # Windows Update uses load balancing on their update servers and spreads
6  # the traffic over several network blocks. The IP address they are using
7  # changes approximately once every five minutes.
8
9  hosts=("update.microsoft.com")     # domains to resolve
10 file="updateservers.csv"           # store list in this file
11 number_of_entries=15               # only store a certain amount of entries
12
13 # Create file if it does not exist
14 if [ ! -e $file ]; then
15     touch $file
16 fi
17
18 # Loop forever
19 while true
20 do
21   for i in "${hosts[@]}"
22   do
23     # We do not want the file to become huge, so we'll remove entries so
24     # we always have a certain amount.
25     if [ `wc -l < $file` -ge $number_of_entries ]; then
26       sed -i '1d' $file  # remove first line
27     fi
28
29     ip=`nslookup $i | grep Address | grep -v '#' | cut -f 2 -d ' '`
30     echo $ip >> $file  # save resolved IP in file
31   done
32
33   # Remove duplicate lines from file
34   # This saves us a lot of processing when looking for update servers in
35   # Flow-dump. It also makes us able to store about 2.5 hours worth of update
36   # servers in just fifteen entries.
37   res=$(awk '!x[$0]++' $file)
38   echo "$res" > $file
39
40   sleep 30  # sleep for 30 seconds
41 done
```

Appendix D

NetFlow Version 9 Field Type Definitions

Field Type	Value	Length (bytes)	Description
IN_BYTES	1	N (default is 4)	Incoming counter with length N x 8 bits for number of bytes associated with an IP flow.
IN_PKTS	2	N (default is 4)	Incoming counter with length N x 8 bits for the number of packets associated with an IP Flow.
FLOWS	3	N (default is 4)	Number of flows that were aggregated.
PROTOCOL	4	1	IP protocol byte.
SRC_TOS	5	1	Type of Service byte setting when entering incoming interface.
TCP_FLAGS	6	1	Cumulative of all the TCP flags seen for this flow.
L4_SRC_PORT	7	2	TCP/UDP source port number i.e.: FTP, Telnet or equivalent.
IPV4_SRC_ADDR	8	4	IPv4 source address.
SRC_MASK	9	1	The number of contiguous bits in the source address sybnet mask i.e.: the submask in slash notation.
INPUT_SNMP	10	N	Input interface index; default for N is 2 but higher values could be used.
L4_DST_PORT	11	2	TCP/UDP destination port number i.e.: FTP, Telnet, or equivalent.
IPV4_DST_ADDR	12	4	IPv4 destination address.
DST_MASK	13	1	The number of contiguous bits in the destination address subnet mask i.e.: the submask in slash notation.
OUTPUT_SNMP	14	N	Output interface index; default for N is 2 but higher values could be used.

Field Type	Value	Length (bytes)	Description
IPV4_NEXT_HOP	15	4	IPv4 address of next-hop router.
SRC_AS	16	N (default is 2)	Source BGP autonomous system number where N could be 2 or 4.
DST_AS	17	N (default is 2)	Destination BGP autonomous system number where N could be 2 or 4.
BGP_IPV4_NEXT_HOP	18	4	Next-hop router's IP in the BGP domain.
MUL_DST_PKTS	19	N (default is 4)	IP multicast outgoing packet counter with length N x 8 bits for packets associated with the IP Flow
MUL_DST_BYTES	20	N (default is 4)	IP multicast outgoing byte counter with length N x 8 bits for bytes associated with the IP Flow.
LAST_SWITCHED	21	4	System uptime at which the last packet of this flow was switched.
FIRST_SWITCHED	22	4	System uptime at which the first packet of this flow was switched.
OUT_BYTES	23	N (default is 4)	Outgoing counter with length N x 8 bits for the number of bytes associated with an IP Flow.
OUT_PKTS	24	N (default is 4)	Outgoing counter with length N x 8 bits for the number of packets associated with an IP Flow.
MIN_PKT_LNGTH	25	2	Minimum IP packet length on incoming packets of the flow.
MAX_PKT_LNGTH	26	2	Maximum IP packet length on incoming packets of the flow.
IPV6_SRC_ADDR	27	16	IPv6 Source Address.
IPV6_DST_ADDR	28	16	IPv6 Destination Address.
IPV6_SRC_MASK	29	1	Length of the IPv6 source mask in contiguous bits.
IPV6_DST_MASK	30	1	Length of the IPv6 destination mask in contiguous bits.
IPV6_FLOW_LABEL	31	3	IPv6 flow label as per RFC 2460 definition.
ICMP_TYPE	32	2	Internet Control Message Protocol (ICMP) packet type; reported as ((ICMP Type*256) + ICMP code).
MUL_IGMP_TYPE	33	1	Internet Group Management Protocol (IGMP) packet type.
SAMPLING_INTERVAL	34	4	When using sampled NetFlow, the rate at which packets are sampled i.e.: a value of 100 indicates that one of every 100 packets is sampled.
SAMPLING_ALGORITHM	35	1	The type of algorithm used for sampled NetFlow: 0x01 Deterministic Sampling ,0x02 Random Sampling.
FLOW_ACTIVE_TIMEOUT	36	2	Timeout value (in seconds) for active flow entries in the NetFlow cache.

Field Type	Value	Length (bytes)	Description
FLOW_INACTIVE_TIMEOUT	37	2	Timeout value (in seconds) for inactive flow entries in the NetFlow cache.
ENGINE_TYPE	38	1	Type of flow switching engine: RP = 0, VIP/-Linecard = 1.
ENGINE_ID	39	1	ID number of the flow switching engine.
TOTAL_BYTES_EXP	40	N (default is 4)	Counter with length N x 8 bits for bytes for the number of bytes exported by the Observation Domain.
TOTAL_PKTS_EXP	41	N (default is 4)	Counter with length N x 8 bits for bytes for the number of packets exported by the Observation Domain.
TOTAL_FLOWS_EXP	42	N (default is 4)	Counter with length N x 8 bits for bytes for the number of flows exported by the Observation Domain.
IPV4_SRC_PREFIX	44	4	IPv4 source address prefix (specific for Catalyst architecture).
IPV4_DST_PREFIX	45	4	IPv4 destination address prefix (specific for Catalyst architecture).
MPLS_TOP_LABEL_TYPE	46	1	MPLS Top Label Type: 0x00 UNKNOWN 0x01 TE-MIDPT 0x02 ATOM 0x03 VPN 0x04 BGP 0x05 LDP.
MPLS_TOP_LABEL_IP_ADDR	47	4	Forwarding Equivalent Class corresponding to the MPLS Top Label.
FLOW_SAMPLER_ID	48	1	Identifier shown in "show flow-sampler".
FLOW_SAMPLER_MODE	49	1	The type of algorithm used for sampling data: 0x02 random sampling. Use in connection with FLOW_SAMPLER_MODE.
FLOW_SAMPLER_RANDOM_INTERVAL	50	4	Packet interval at which to sample. Use in connection with FLOW_SAMPLER_MODE.
MIN_TTL	52	1	Minimum TTL on incoming packets of the flow.
MAX_TTL	53	1	Maximum TTL on incoming packets of the flow.
IPV4_IDENT	54	2	The IP v4 identification field.
DST_TOS	55	1	Type of Service byte setting when exiting outgoing interface.
IN_SRC_MAC	56	6	Incoming source MAC address.
OUT_DST_MAC	57	6	Outgoing destination MAC address.
SRC_VLAN	58	2	Virtual LAN identifier associated with ingress interface.
DST_VLAN	59	2	Virtual LAN identifier associated with egress interface.
IP_PROTOCOL_VERSION	60	1	Internet Protocol Version Set to 4 for IPv4, set to 6 for IPv6. If not present in the template, then version 4 is assumed.

Field Type	Value	Length (bytes)	Description
DIRECTION	61	1	Flow direction: 0 - ingress flow, 1 - egress flow.
IPV6_NEXT_HOP	62	16	IPv6 address of the next-hop router.
BPG_IPV6_NEXT_HOP	63	16	Next-hop router in the BGP domain.
IPV6_OPTION_HEADERS	64	4	Bit-encoded field identifying IPv6 option headers found in the flow.
MPLS_LABEL_1	70	3	MPLS label at position 1 in the stack. This comprises 20 bits of MPLS label, 3 EXP (experimental) bits and 1 S (end-of-stack) bit.
MPLS_LABEL_2	71	3	MPLS label at position 2 in the stack. This comprises 20 bits of MPLS label, 3 EXP (experimental) bits and 1 S (end-of-stack) bit.
MPLS_LABEL_3	72	3	MPLS label at position 3 in the stack. This comprises 20 bits of MPLS label, 3 EXP (experimental) bits and 1 S (end-of-stack) bit.
MPLS_LABEL_4	73	3	MPLS label at position 4 in the stack. This comprises 20 bits of MPLS label, 3 EXP (experimental) bits and 1 S (end-of-stack) bit.
MPLS_LABEL_5	74	3	MPLS label at position 5 in the stack. This comprises 20 bits of MPLS label, 3 EXP (experimental) bits and 1 S (end-of-stack) bit.
MPLS_LABEL_6	75	3	MPLS label at position 6 in the stack. This comprises 20 bits of MPLS label, 3 EXP (experimental) bits and 1 S (end-of-stack) bit.
MPLS_LABEL_7	76	3	MPLS label at position 7 in the stack. This comprises 20 bits of MPLS label, 3 EXP (experimental) bits and 1 S (end-of-stack) bit.
MPLS_LABEL_8	77	3	MPLS label at position 8 in the stack. This comprises 20 bits of MPLS label, 3 EXP (experimental) bits and 1 S (end-of-stack) bit.
MPLS_LABEL_9	78	3	MPLS label at position 9 in the stack. This comprises 20 bits of MPLS label, 3 EXP (experimental) bits and 1 S (end-of-stack) bit.
MPLS_LABEL_10	79	3	MPLS label at position 10 in the stack. This comprises 20 bits of MPLS label, 3 EXP (experimental) bits and 1 S (end-of-stack) bit.
IN_DST_MAC	80	6	Incoming destination MAC address.
OUT_SRC_MAC	81	6	Outgoing source MAC address.
IF_NAME	82	N	Shortened interface name i.e.: "FE1/0".
IF_DESC	83	N	Full interface name i.e.: "'FastEthernet 1/0".
SAMPLER_NAME	84	N	Name of the flow sampler.
IN_PERMANENT_BYTES	85	N (default is 4)	Running byte counter for a permanent flow.
IN_PERMANENT_PKTS	86	N (default is 4)	Running packet counter for a permanent flow.
FRAGMENT_OFFSET	88	2	The fragment-offset value from fragmented IP packets.

Field Type	Value	Length (bytes)	Description
FORWARDING STATUS	89	1	Forwarding status is encoded on 1 byte with the 2 left bits giving the status and the 6 remaining bits giving the reason code.
MPLS PAL RD	90	8 (array)	MPLS PAL Route Distinguisher.
MPLS PREFIX LEN	91	1	Number of consecutive bits in the MPLS prefix length.
SRC TRAFFIC INDEX	92	4	BGP Policy Accounting Source Traffic Index.
DST TRAFFIC INDEX	93	4	BGP Policy Accounting Destination Traffic Index.
APPLICATION DESCRIPTION	94	N	Application description.
APPLICATION TAG	95	1+n	8 bits of engine ID, followed by n bits of classification.
APPLICATION NAME	96	N	Name associated with a classification.
postipDiffServCodePoint	98	1	The value of a Differentiated Services Code Point (DSCP) encoded in the Differentiated Services Field, after modification.
replication factor	99	4	Multicast replication factor.
DEPRECATED	100	N	DEPRECATED.
layer2packetSectionOffset	102	-	Layer 2 packet section offset. Potentially a generic offset.
layer2packetSectionSize	103	-	Layer 2 packet section size. Potentially a generic size.
layer2packetSectionData	104	-	Layer 2 packet section data.