Using Machine Learning Techniques for Advanced Passive Operating System Fingerprinting

Julius Schwartzenberg

August 12, 2010

Abstract

TCP/IP fingerprinting is the active or passive collection of information usually extracted from a remote computer's network stack. The combination of such information can be then used to infer the remote operating system (OS fingerprinting). OS fingerprinting is traditionally based on a database of "signatures". A signature comprises several features (i.e., pairs attribute/value) extracted from network packets generated by a known operating system. Signatures are manually generated (and updated) by observing several operating systems. There are two types of fingerprinting: active and passive. In this work, we focus on automating the generation and updating of the signatures for passive fingerprinting. By using classification algorithms we deal with fingerprints which do not have an exact match with an already known signature.

Chapter 1

Introduction

The inference of a remote OS is an important step for several activities related to computer security. Malicious users might fingerprint an OS to select only the correct exploits that they will later use to break-in, in order to lower the chances of being detected by an intrusion detection system (IDS). On the other hand, the same IDS might infer the OS of hosts it monitors in order to make its detection more precise, by discarding attack signatures tailored for a different OS. Knowledge about the remote OS may also be used to detect changes on remote systems: a remote system may have been replaced; or to detect unallowed OSs: not all OSs may be allowed by company policy. Active fingerprinting requires the parties to exchange some packets, which is not always desirable. An attacker may want to be as "hidden" as possible, an IDS may not be authorized to send out packets on the monitored network. Thus, passive fingerprinting offers a choice to meet such constraints. Several systems have been developed to perform passive OS fingerprinting, which although accurate, fail when newer systems or tweaked systems are introduced and databases are not maintained.

The following subsections describe the exact goal of this thesis. Then Chapter 2 shows the current state of the art on both active and passive operating system fingerprinting. All the existing methods and tools are described in chronological order in this chapter. Chapter 3 describes the improved architecture. Chapter 4 details the implementation and the testing. The final chapter is the conclusion with some detailed information about possible future work.

1.1 Goal

1.1.1 Initial state

The state of the art gives a complete overview of all the existing fingerprinting tools. The most noteworthy tool of these is p0f, because it is the most mature passive tool available, also see Appendix E.4. To recognize a given system, p0f needs to have a fingerprint for it in its fingerprint database. When p0f fingerprints a system and its fingerprint does not match, p0f will mark the system as "unknown". As newer operating systems (TCP/IP stacks) are released, p0f's fingerprint database needs to be manually updated to correctly match new systems.

1.1.2 Problem

Although passive fingerprinting tends to be accurate, keeping the fingerprint database up to date is essential to achieve high accuracy. Right now the process of creating a new fingerprint and adding it to the database is done manually. The current version of p0f and its fingerprint database are, at the time of writing, almost 4 years old [Zal06c]. This shows that keeping the database up to date is apparently too much effort. The result is that p0f's accuracy has dropped significantly, only because it is missing the fingerprint information needed to recognize modern operating systems, such as MS Windows Vista and MS Windows 7. According to the statistics on W3Schools, a website about web development, these versions were used by more than a quarter of all computer users in February 2010 [w3s10].

1.1.3 Research Question

The goal is to automate the fingerprinting process, so that new operating systems can be recognized with as little human effort as possible. Right now this is not possible because for every system a fingerprint needs to be created manually. This means fingerprints should be generated automatically with only the proper tagging of the fingerprinted operating system being necessary.

Can machine learning techniques be used to recognize new operating systems which current fingerprinting systems cannot automatically recognize?

Chapter 2

State of the art

2.1 Introduction

The ability to detect which operating system is running on a remote system has several advantages. A common use is statistics gathering by companies that are interested in market share. However, there are several other uses which are even more significant.

When scanning a network for possible vulnerabilities, knowing which operating systems (OSs) are running, may quickly indicate which systems are outdated and need updating. This knowledge is controversial because it is very useful for an attacker. System administrators can use the same knowledge to protect their systems. Internet service providers could also help their customers with protecting their systems by redirecting them to a webpage which points them to updates. By using a packet filter which can detect what system they are using, they can redirect all HTTP request to the appropriate page.

In the contexts of network monitoring and intrusion detection the knowledge about the network context plays a crucial role, as argued also by Kruegel and Robertson [KR04] and by Pietraszek [Pie04]. Hence, the knowledge about the operating systems gives additional insights about the attack. If an attack is observed targeting a specific vulnerability on a GNU/Linux system while the actual system turns out to be running MS Windows based on the response traffic, it may indicate a Bot which is brute forcing a single attack against any host it can come across or perhaps an unskilled attacker. On the other hand, when the attack properly matches the OS, this could

indicate that the attacker is skilled and not just a Bot. Enhancing an intrusion detection system (IDS) with OS detection capabilities would thus enable it to make a much better estimate of the attack impact. Taleck [Tal03] explicitly highlights the importance of OS fingerprinting for IDSs.

Despite the clear advantages of OS fingerprinting, there are few papers in the literature which address this problem.

2.2 Active and passive

The most important distinction between different OS detection mechanisms is whether the detection is done actively or passively. Active systems generate specifically crafted data to identify systems based on their response. Passive systems on the other hand stay completely silent and will determine the OS based on the traffic that happens to come by. This means that passive systems can never be detected through the network, but they are dependant on existing traffic. Another difference is that some systems are never reached by active scanners, because they are behind a firewall or some network address translation mechanisms are in place [EF94]. Passive scanners would be able to scan the traffic coming from these systems. The active and passive systems are described in the following sections. Some passive systems also have the capability to actively trigger packets and could thus be seen as a combination of an active and passive system. Since the fingerprinting logic is aimed at passive scanning in these cases, those systems are also described in the section about passive fingerprinting.

2.3 Active

This section describes the most relevant existing active fingerprinting tools.

2.3.1 Before TCP/IP fingerprinting

Before TCP/IP fingerprinting became popular, software that had to perform the same task of remote operating system detection used methods that were often very simplistic. The original QueSO documentation lists the following:

- rpcinfo (RPC (Remote procedure call) [Whi75] is a framework for network-based resource sharing and common on Unix systems to support services such as NFS and NIS)

- snmp (the Simple Network Management Protocol [CFSD90] is used to monitor networked systems)
- telnet banner (unencrypted remote terminals often display the running system name before the log-in prompt)
- SendMail version
- analyse binaries available through ftp
- other slow and suspicious stuff (likely other application dependant ways)

These methods are simplistic and system administrators can prevent these types of identification, by not advertising the OS in a banner or by just upgrading these services to their modern equivalents, which do not leak this information anymore. Another problem these methods have is that they are heavily dependant on specific services running on the remote host, which are not always there. A better method should be used to properly detect a remote operating system.

2.3.2 sirc, checkos and SS

According to an article written in 1998 by Gordon Lyon for Phrack Magazine [Lyo98] sirc, checkos and SS were the first programs available which were capable of recognising different OSs using TCP/IP fingerprinting. They were limited in doing so and could only identify whether a system was one from a handfull of OS classes. Commercially available software at that time was only capable of checking banners however. These tools do not seem to be referenced anymore and not much information can be found about them.

2.3.3 QueSO

The first wide-spread tool which could detect operating systems remotely using TCP/IP fingerprinting was QueSO. It was originally developed by Jordi Murg in 1998. It is the first tool which used fingerprints stored in a separate file. This allowed new operating systems to be added without modifying the software.

The method of detection uses what are called ambiguous packets. Those are packets for which the RFCs do not specify a precise way to handle them.

Because of this ambiguity, this results in different behaviour from different TCP/IP stacks. [Mur98] The specific packets QueSO sends out are described in Appendix E.1.

QueSO is able to differentiate about 100 different operating systems. [Mur99] The fingerprint database is not kept up to date anymore with new OS releases however, so it is possible QueSO could detect more different operating systems with an updated database.

2.3.4 Nmap

Shortly after QueSO was released, the popular network mapping tool Nmap, written by Gordon Lyon, also gained support for OS detection based on TCP/IP fingerprinting in October 1998. It continued on the idea that there should be a separate fingerprint database. The biggest improvement over QueSO was the number of tests it could do. Nmap added several extra tests which significantly increased the number of different systems it could recognise. [Lyo09a, Lyo98]

Since the original OS detection support was added to Nmap, it has seen quite a lot of improvements including one complete revision. The current version of Nmap sends out up to 16 TCP, UDP and ICMP probes. These are analysed in Appendix E.2.

Having so many tests, Nmap is one of the most comprehensive scanners available. With its tests it is able to differentiate an enormous amount of systems. A disadvantage of using so many tests is that it allows for easy detection. This is described by Greenwald and Thomas [GT07a]. Some of the tools described in the next sections try to avoid this. Another recurring problem with active tools is the fact they can often only scan a firewall and not the systems behind it. Only passive systems will fully solve this.

2.3.5 Xprobe

After tools which could differentiate TCP stacks became common, Ofir Arkin decided to look into the RFCs related to the ICMP and found that the ICMP has quite some potential to be used in OS fingerprinting. [RK04]

Together with Fyodor Yarochkin he released the first version of a tool called Xprobe in August 2001. It was based on an algorithm named X, designed by Arkin, which was based on a combination of information ob-

tained using ICMP with tree based decisions. Using ICMP instead of TCP had several advantages, especially when very similar TCP stacks were used. Examples are MS Windows NT4 and MS Windows 9x or MS Windows 2000 and MS Windows XP, where the different systems could not be distinguished from each other before.

Another difference was the small amount of traffic it takes. With the reply from a single datagram, Xprobe can already differentiate eight different systems. A maximum four datagrams are sent in total, which also makes the tool fast. The probes are unlikely to be picked up by IDSs because they are not malformed and normal ICMP datagrams are already common among networks. [Ark02]

Xprobe was also the first program that used "fuzzy logic". A potential OS is given a score based on how well it matches with certain tests. The end result is a list of scores together with a probability. [Kol05]

Details on Xprobe's mechanisms are described in Appendix E.3.

2.3.6 RING

In April 2002 Franck Veysset, Olivier Courtay, and Olivier Heen of the Intranode Research Team published a paper [VCH02] in which they described a concept to distinguish between OSs on a network by measuring the behaviour of their TCP retransmission time-out lengths. While the TCP RFC includes an algorithm for retransmissions, it is not imposed. It turns out that although most systems implement a variation of the algorithm, there are differences which can be exploited to reveal the OS.

As a proof of concept, the tool RING was developed. It tries to initiate a connection with a server, but does not acknowledge the initial acknowledgement. This causes the other host to retransmit its acknowledgement at timed intervals. By measuring the time between the incoming packets, it is possible to figure out the TCP implementation. The fingerprint in this case consists of the retransmit times an OS uses to retransmit its acknowledgements.

It turns out that this method gives quite accurate results, even though it is dependant on only a single open port. A disadvantage however is that it takes more time than the other techniques. It takes for example 12 seconds before MS Windows 2000 and FreeBSD can be distinguished, because only then, a difference in their retransmissions behaviour appears. This is shown

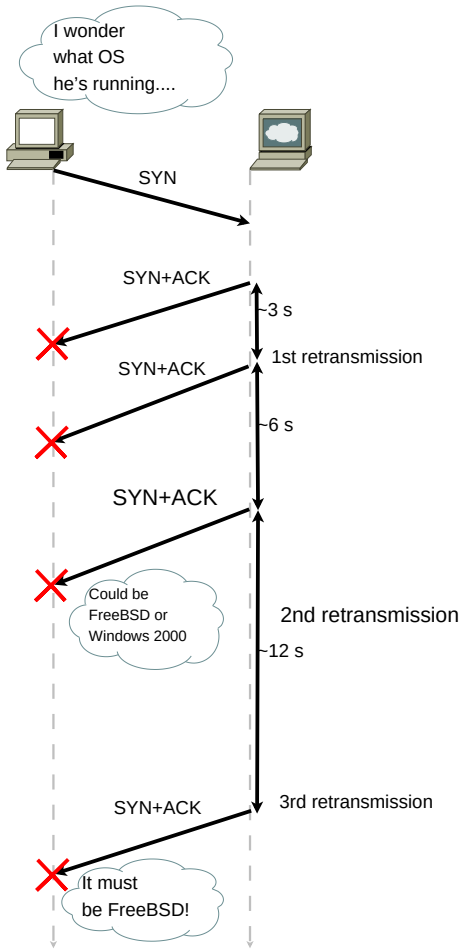


Figure 2.1: Successful detection of the remote OS by measuring the retransmission times

in Figure 2.1.

2.3.7 Neural networks

Carlos Sarraute and Javier Burroni from the company Core Security Technologies published a paper in 2008 in which they described their experiments using neural networks to improve OS fingerprinting [SB08]. Their method uses a neural network which, although mostly based on Nmap's fingerprint database, uses some application level tests which made use of the RPC service to test for specific MS Windows versions.

One flaw they notice in the way Nmap treats its own fingerprints is that Nmap gives a relatively high score very quickly to fingerprints which contain only a small amount of information. This happens because Nmap uses the Hamming distance between the received packets and the fingerprints in its database to compute the result. In their tests, they leave out these finger-

prints by using an initial neural network to determine which fingerprints are relevant for the attacker. They reduce the testing space of possible responses by using two techniques called Correlation Matrix Reduction and Principal Component Analysis. The Correlation Matrix Reduction notes similar behaviour in similar systems and reduces the tests for this family of systems to the most discriminative tests. Principal Component Analysis is a technique where the factors with the most variance are given more emphasis than factors with less variance when comparing results.

The system is trained using a initial dataset containing inputs and expected outputs. This is called supervised training. During their test they used Monte Carlo simulation to generate inputs using Nmap's signature database. The paper does not go into Monte Carlo simulation. This book [BH02] by Binder and Heermann describes Monte Carlo simulation in detail.

Using these techniques, their detection is slightly more accurate than Nmap in their test environment. Especially specific versions of operating systems could be detected more precisely. However being dependant on the RPC service severely limits the useability of the system. Even within private networks the RPC service is unlikely to be active and running it on a public network is generally considered to be a bad security practise. The initial step to test a given fingerprint for its relevance does not seem logical. According to the text an irrelevant operating system is one for which an attacker has no exploits. One of their examples is a system using Mandrake 7.2 which is detected by Nmap as a broadband router made by ZyXel. Considering Mandrake 7.2 was eight years old at the time the paper was published and had also been succeeded by more than eight releases while on the other hand broadband routers are increasingly common as the public host among households, this idea of relevance is disputable. Limiting the fingerprinting options in this way is a big drawback of the system.

The consessions done to obtain the desired results appear to be too big to make neural networks suitable to be applied generally for OS fingerprinting.

2.4 Passive

This section describes the most relevant existing passive fingerprinting tools.

2.4.1 p0f

The first version of p0f was written by Michal Zalewski in 2000. After it had been maintained for a while by William Stearns, it was completely rewritten again by its original author.

p0f has four different detection modes used in four different scenarios respectively:

- Incoming connection fingerprinting (what system is connecting to yours)
- Outgoing connection fingerprinting (what system are you connecting to)
- Outgoing connection refused (what system is refusing your connection)
- Established connection fingerprinting (what systems do you have a connection with)

Of these four, only the first method is really well supported. It works by analysing certain properties of SYN packets from incoming TCP/IP connections. With the other methods SYN+ACK, RST and stray ACK packets are analysed. Appendix E.4 describes p0f's fingerprinting. [Zal06b]

2.4.2 Packet Filtering in OpenBSD

The packet filter included with OpenBSD is also capable of doing passive operating system fingerprinting. This support seems to be based on p0f, using a similar fingerprint file. Only p0f's SYN filtering is available. With this support is it possible to write rules based on the detected OS. This could mean connections from certain OSs are blocked for example or requests going out on the HTTP port (port 80) for a known vulnerable system could point to an upgrade.

2.4.3 OSF - passive OS fingerprinting for iptables

This is a module for Linux' netfilter similar to the support available for OpenBSD. netfilter is the infrastructure in modern Linux kernels used for packet filtering. Typical use cases are firewalling, network address translation (NAT) and packet mangeling. This support is largely based on p0f, using the same fingerprint file as the support in OpenBSD. The OSF website credits Michal Zalewski (the author of p0f) for the original idea.

2.4.4 Ettercap

Ettercap is a tool which can be used for many purposes. Some of its most prominent features are its man-in-the-middle attack abilities. Ettercap also includes plug-ins for specific features including a plug-in named "finger" which describes itself as a passive fingerprinting plug-in. While it is using the same techniques passive tools also use, it is not really passive as it sends a SYN itself to the other host and uses the corresponding SYN+ACK reply to fingerprint the other system.

Its fingerprinting mechanism is rather simple. The database contains a list of fingerprints sorted on the properties. This means systems which have a similar fingerprint are near each other. When a SYN+ACK packet has arrived, Ettercap starts matching this fingerprint with a fingerprint from its database. When a match cannot be found, it will give the nearest match which it had reached in the database. Its database has not been updated since 2004, so it is quite outdated.

2.4.5 SinFP

A very interesting tool written in Perl by Patrice Auffret is SinFP. Using one set of signatures, it is capable of doing both active and passive OS fingerprinting. Its first release was in June 2005, which makes it one of the more modern tools. In June 2006 a completely rewritten version was released. One of the interesting points the changelog (the report of changes between versions) makes, is that this version has more accurate passive OS fingerprinting. [Auf06] The program was originally written because it turned out that Nmap's OS detection capabilities were getting into trouble in a specific situation which had become more and more common on the internet. Many home users are connected using a router which performs IP masquerading in combination with network address translation (NAT). Both controversial technologies because they broke some of the original intentions of the IP, they also bring issues for OS detection tools. In addition to those problems, also packet normalisation is emerging which is making Nmap's approach of fingerprinting obsolete. To prevent getting into the same problems, SinFP was written specifically to avoid using tests that could be influenced by the problems described. [Auf08]

SinFP is also the first tool to use a real database for its fingerprints. The

database is based on SQLite and new versions of the fingerprint database can easily be downloaded and used with SinFP. The previously described tools all used plain text files which contained the fingerprint information. Although these are similarly easily updatable, it would be easier to integrate SinFP into another system with which it could share fingerprinting information. Another advantage of using a database oriented design to store the fingerprints, is the capability to store certain recurring values into a separate table. Next to the fingerprint table, there are tables used to store recurring patterns in: TCP flags; TCP maximum segment size (MSS); TCP options, TCP window size and certain binary properties.

Other interesting features include:

- IPv6 support (including matching against IPv4 signatures)
- Very small amount of packets (often just one) needed to get a match
- Both online and offline modes
- Heuristic matching algorithm to be able to deal with customised stacks

Auffret [Auf08] describes these features in detail.

2.4.6 Satori

Eric Kollmann created a closed source tool called Satori, which is also capable of doing passive OS fingerprinting. The documentation is rather sparse. One interesting aspect is that it prints the results from multiple detection engines, so one immediately sees the results of different detection mechanisms. By default Satori uses its own TCP and DHCP fingerprinting together with an engine which can work with Ettercap fingerprints and an engine with p0f fingerprints. For this reason, in addition to its own fingerprint files, also p0f's and Ettercap's fingerprint files are included with the Satori download.

When a TCP packet passes by, Satori will display the output of its own TCP engine, Ettercap and p0f below each other. This quickly gives an impression of the quality of the different fingerprintings, which normally only differ because one lists a specific version number the other lacks.

A really interesting feature on local area networks (LANs) is Satori's DHCP fingerprinting engine. DHCP packets are always broadcasted over the whole network. This means nothing like ARP poisoning is necessary to

obtain such packets from other systems to get an impression of the operating systems that are popular on a LAN. This feature does not work past your own LAN, so its not useful for really distant remote OS detection over the Internet.

There is very little documentation and it is also not possible to look at Satori's source code. The author has written a paper [Kol05] however which mentions many OS fingerprinting techniques, most of which are employed by Satori.

2.4.7 Bayesian classifier

A paper published in 2004 by Robert Beverly from MIT proposes using a Bayesian classifier for passive fingerprinting [Bev04]. The author uses a naive Bayesian classifier that considers the TTL, DF, window size and SYN size field from initial SYN packets. Naive means all the different fields are considered to be fully independent. While this is not completely true, this assumption does not hinder performance.

In two separate testing rounds the classifier is once tested while it is trained using p0f's existing fingerprint database. The second round it was trained using traffic from a public non-technical webserver and identifying the operating system based on the operating system part from the user-agent value in the HTTP header. Although this may introduce errors due to proxies and useragent spoofing, these are considered to be statistically insignificant. The results of both tests were roughly similar, so both training methods would seem equally suitable with the last one having the advantage new systems may be added automatically.

The classifier was tested against "approximately 38M packets from an hour long mid-week trace collected at a US internet exchange at 16:00 PST in 2003". It turns out that the success rate of the system is quite comparable to p0f. This method has two specific advantages over p0f. First, since every value has its own probabilistic weight, it can handle TCP stacks that have been tweaked by the user. Second, the classifier can produce a "maximum-likelihood guess" when there is no accurate fingerprinting data available. A disadvantage of this approach is that a large number of hosts is necessary to verify whether the classifier is working correctly. With the traditional approach, a rule is already valid when it corresponds to a single host.

The paper uses the results to identify hosts behind a NAT router. Al-

though they are able to observe independent systems the results are still very limited. Other methods to count such hosts currently give much better results, which is why is it is only suggested that this method may be suitable in combination with other methods. There are far more interesting use cases for this type of passive fingerprinting however.

2.4.8 Answer Set Programming

Gagnon, Esfandiari and Bertossi published several papers [GEB07, GE09] on using Answer Set Programming to perform OS fingerprinting. The main idea of the paper is that while current passive systems often base their full analysis on a single packet, most of the time there are many more packets available and using those too will improve accuracy. This brings a new requirement to the system, it will have to keep track of history. By building a tool using Answer Set Programming it is possible to add more reasoning to the fingerprinting mechanism. Data from older packets can be used and passive and active techniques may be unified. Vladimir Lifschitz [Lif02] describes Answer Set Programming (ASP) as "representing a given computational problem by a logic program whose answer sets correspond to solutions, and then use an answer set solver to find an answer set for this program". An example scenario where this system would work optimal is described in the paper. Initially each computer is considered to be capable to run any OS. As traffic is seen for specific systems, the set of possible OSs is updated eliminating systems that cannot generate the observed traffic using the passive scanner. When a user then requests information about a particular system, specific active tests may be executed to obtain the rest of the information if necessary. The test results of the system are very promising. Their implementation was able to correctly recognise more than 80% of the 95 OSs tested. Other tools, including Nmap, p0f and Xprobe, had at most a score of 34%.

2.5 Future

2.5.1 IPv6

Although the migration to IPv6 [DH98] is taking longer than expected, more and more internet service providers are upgrading their networks to IPv6

and are handing out IPv6 subnets to their customers. In practise most hosts still use two stacks so they are reachable using both IPv4 and IPv6. Active tools will not have to necessarily use IPv6 yet to be able to scan them. For passive detection IPv6 support will be necessary as many applications tend to prefer the IPv6 route over the IPv4 one. This means that a dual stack host will use IPv6 as much as possible and passive tools have to support IPv6 to be able to fingerprint them. As IPv6 adoption is continuing also hosts with IPv6 only addresses will start to appear, which will require full IPv6 support from active tools.

It should not be very difficult to add IPv6 support to the current generation of tools. Even the fingerprints could stay the same, since most of them are based on differences in the transport protocol implementations, which should not be influenced by a change in the network protocol.

Currently the only OS detection program that appears to support IPv6 is SinFP. Although Nmap has some IPv6 support, e.g. for port scans, it cannot do OS detection over IPv6 yet.

2.5.2 IPsec

IPsec, specified in RFC 2401 [KA98], is a system which allows full encryption of any data carried inside the network layer. This means that transport protocol data is not visible while flowing over the internet.

It appears that hosts which would only be capable of accepting secure connections will not appear soon. IPsec can be optionally negotiated, but is not mandatory for all connections that would be made to the host. This means current active tools can still scan a system. When IPsec only hosts will appear, it would still be possible for active tools to go over IPsec and scan the hosts.

Passive tools on the other hand will not be able to do much anymore when IPsec is used, as they cannot see anything of the transport data. A noteable exception is the case where the passive scanner is running on one of the end hosts itself. (This could be the case when someone wants to keep track of the systems connecting to his server.) Then it could potentially access the unencrypted transport protocol data and perform its fingerprinting.

2.5.3 Masking and Scrubbing

Research has also been done in the area of preventing the above tools from accurately fingerprinting the operating system. Watson, Smart, Malan and Jahanian [SMJ00, WSMJ04] created a packet scrubber which they claim normalises all the traffic and thus leaves no system specific details in the outgoing traffic. Kalia and Singh [KS05] describe some tweaks that can be done to MS Windows and Linux so that Nmap is not able to recognise them anymore. Also Beck [Bec01] uses this as a basis by letting a system respond differently when it detects an Nmap scan. As masking and scrubbing methods may evolve, it may be more difficult in the future to fingerprint OSs. Currently most masking tools are heavily based on the fingerprinting tools that are currently in wide use, so a new tool might just be able to work when it uses newer fingerprinting methods.

2.6 Conclusion

There are a number of tools available to remotely identify an operating system over the internet. Although operating systems implement a similar set of protocols to communicate over a network, there are enough implementation details which reveal the origin of software running on the system.

Quite a few tools to perform OS detection have been developed. A clear evolution is visible when looking at all the different tools that are available now. Some of the prominent developments:

- The abandoning of the application layer, moving fully to the transport and network layers.
- The use of external fingerprint information as opposed to hardcoding test cases and results.
- Looking at different transport protocols, not just the TCP but also the ICMP and the UDP.
- Limiting the possible detection, sending a few packets or doing completely passive detection.
- Trying different types of algorithms, for example making decisions based on a tree instead of a direct comparison.

- The use of multiple completely separate detection engines.
- Keeping track of past events for use in a comprehensive analysis

Also some experiments have been done using neural networks and Bayesian classifiers which bring quite a different approach to operating system fingerprinting, but unfortunately did not give practical results.

There are still some problems to solve. The traditional problem which active scanners face, a firewall (or a router with IP masquerading) in front of the target, can only be worked around by using a passive scanner. Passive scanners have limitations too. Notably signature databases tends to get out of date and without continuous maintenance the tools will stop working, as they will not be able to recognise newer systems on the internet anymore. Proper building and maintenance of a good fingerprint database appears to be a serious issue which still requires a lot of manual work. Some initiatives such as SinFP's support to use the same fingerprints for both active and passive scanning are already a good step forward in this area.

An overview of the advantages and disadvantages of the different types of scanners:

	Active	Passive
Range	Can only fingerprint systems directly connected to the network	**May fingerprint systems behind a firewall**
Target	**Can fingerprint any available system**	Can only fingerprint systems of which traffic passes by
Visibility	Fingerprinting can be detected by IDSs	**Detection is impossible**
Granularity	**Can send crafted packets to provoke special and different behaviour in otherwise similar systems**	Can only use information from passing traffic for fingerprinting

More improvements in OS detection can still be achieved regarding fingerprints. Research into the generation, comparison and verification of fingerprints could bring even more possibilities. Which parts of active and passive signatures are actually similar and how much assimilation of sig-

natures can be provided? How do signatures compare between IPv4 and IPv6? Are there similarities between different transport protocols in certain stacks? Could p0f's extensive database for its SYN probing somehow be reused for its other probes? Could a program based on multiple engines share its fingerprints between the engines? Would this make the quality of the detection equal or does this still depend a lot on the detection mechanism? New ideas and performance measurements for tests are also described by Greenwald and Thomas [GT07b]. The finetuning how existing tests are used and testing additional properties may still improve the fingerprinting.

Another good addition would be automated fingerprint generation. This could be done in several ways. One idea would be to run a tool which analyses the source code of unknown systems, maybe by compiling and running parts of it, and see what kind of data the code would generate. Another approach would be to run a tool against a host running a known version of an operating system which is kept up to date. Since updates may influence the behaviour of the network stack, fingerprints would also have a date (or version) property which would indicate the date at which the system was fully patched when it matched that fingerprint. One approach was also shown in the paper about the Bayesian classifier, which suggests using the traditional application layer based headers to collect a large amount of fingerprinting information. This is very prone to errors however.

Chapter 3

Architecture

3.1 Introduction

To achieve fully automatic operating system fingerprinting, several steps are needed. In Figure 3.1 the basic components of the architecture are shown.

The first step consists of collecting the relevant data from the network stream. This data is then handed over to a simple matching based fingerprinting component which fingerprints the data and compares it to the already known fingerprints. When an exact match is found, the process ends and the classification information returns. When no exact match is found, the data is processed by a trained classifier based on machine learning techniques. The classifier tries to find the closest match based on a known training set.

Two full pictures which show in detail how the fingerprinter works are shown in Figure 3.2 and Figure 3.3.

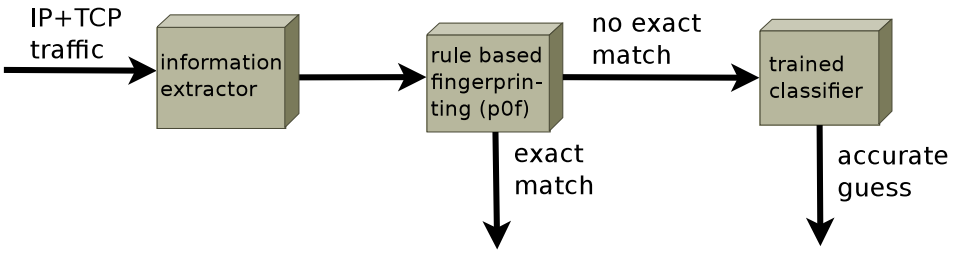


Figure 3.1: Overview of the architecture

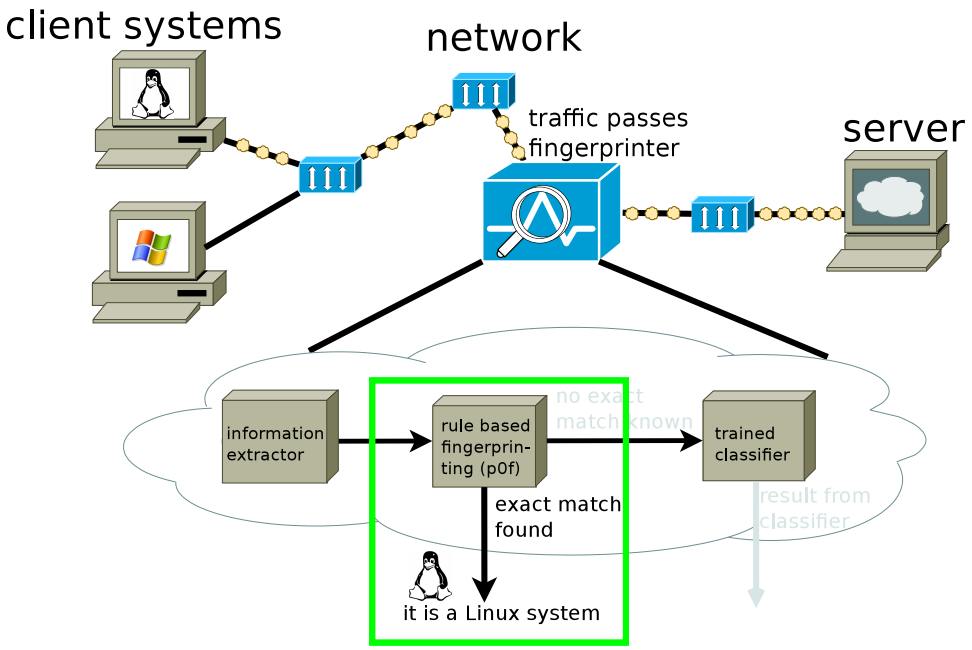


Figure 3.2: Detecting a known system which has an exact match in the database

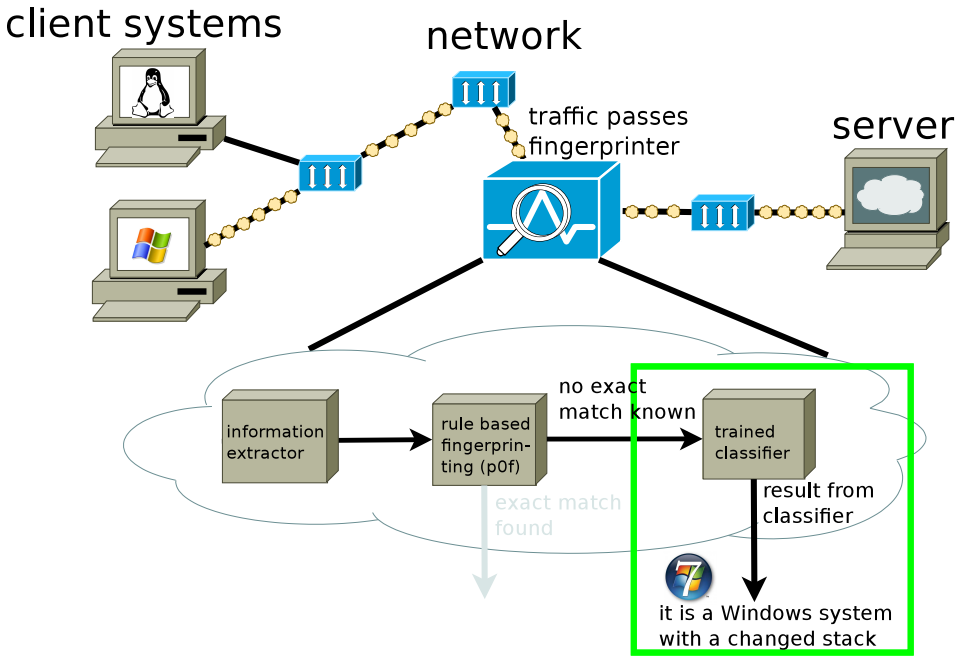


Figure 3.3: Detecting a newer system which requires classification

Table 3.1: Example of a p0f fingerprint

OS genre	OS descr	WSS	TTL	D	size	options	quirks
Linux	2.6	S4	64	1	60	M*,S,T,N,W7	-

3.2 Information extractor

The first step in the process is extracting relevant information from a packet stream. This can either be captured live or a stored capture of IP traffic. From the stream, the only information that is interesting are packet and segment headers of IP packets which include TCP SYN or SYN+ACK segments, because fingerprinting is limited to them. Details on TCP SYN and SYN+ACK headers are described in Appendix E.1. All the other data can be discarded. Initially regular packets were also analyzed but in general they produced fingerprints that could not be distinguished, because such packets generally do not have any options or other specific details through which they could be distinguished.

The collected header data is the information on which the classification process will be based. It is fed in a sanitized way to the other components. This is done in the same format as rules used by p0f. A representation of a single p0f rule is shown in Table 3.1.(The next section explains all the fields in detail.)

3.2.1 Normalizing fingerprint data

The initial problem is concerned with the automatic generation of fingerprints. This means that the observed network headers are stored in such a way that a comparison is possible between different runs against systems. It is not possible to base fingerprinting directly on the raw headers, because of addressing fields for example which contain information on which differentiation should be avoided. Based on the full packet and segment headers, the relevant fields which are important for accurate fingerprinting need to be extracted. In the case of p0f this is done manually according to the specification that instructs users how they can generate a new fingerprint to add to p0f's database. This section details how this process works for p0f, the tool automates this.

When building a simple fingerprint, based on one segment by just reading the header data from a packet, discarding obvious addressing data, it will

Table 3.2: Example of unprocessed fingerprint data, the OS genre & description are not known yet

OS genre	OS descr	WSS	TTL	D	size	options	quirks
(Linux)	*(2.6)*	5860	64	1	60	M1460,S,T,N,W7	-

Table 3.3: FreeBSD 5.1 fingerprint from p0f's database

OS genre	OS descr	WSS	TTL	D	size	options	quirks
FreeBSD	5.1 (1)	65535	64	1	60	M*,N,W0,N,N,T	Z

be similar to what is represented in Table 3.2. This notation is based on how entries in p0f's database look. The first value of the fingerprint, WSS, is the window size. The next value is the initial TTL. Then the do not fragment flag is listed. Either 1 for true or 0 for false. The next value is the overall packet size (IP header + TCP header). The final two columns are special. The first one contains all the options that the TCP header specifies in order. The second one indicates whether a system has special quirks or other abnormalities. An example of this is usage of the IP options field (which is almost never used) or a non-zero URG pointer. Table 3.3 shows a Z quirk which indicates the ID field in the IP header was zero.

The new tool is capable of generating p0f compatible fingerprints from raw packet data. The automated process has been based on the process p0f users needed to follow manually. It builds a complete fingerprint using just a single packet and segment header combination. See Appendix E.4 for more information on the original p0f fingerprints. The main areas of attention are the WSS and the MSS option. The WSS needs to be calculated based on the MSS option (if it is available). In case the value is a multiple of the MSS, the maximum segment size, it will be the multiplier prefixed with an S. There are also cases where systems tend to use a multiple of the MTU, the maximum transfer unit, then this multiplier will be prefixed with a T. The MSS itself should normally be wildcarded as it is link dependant. The other values can be copied over without any changes. Almost all fingerprints from p0f's database could have been generated automatically in this way. The only fingerprints that could not have been automatically generated in this way, are fingerprints which contain a modulo operator signifying the value in place is always a divisor of the value behind it. The reason is that these fingerprints would have to be based on multiple different segments from

the same system with different values in place to find a common divisor. An automatic system just adds separate fingerprints for each of the new values that it comes across, so this may not be even necessary. A search through p0f's current fingerprinting database shows that only some AIX and very few MS Windows versions have a fingerprint containing a modulo sign. Apparently some AIX versions always uses a window scale which is divisible by two and some Windows versions tend to have a window size which is divisible by exactly 8192. All other fingerprints can be generated fully by software, except for the OS genre and details, which of course have to be verified by a human the first time.

Although p0f's fingerprints are a good starting point, it is important to have flexibility and be able to store more details than p0f originally does. Being able to match p0f is significant however, because it allows comparison of the accuracy with it.

3.3 Exact fingerprint based matching

Traditionally, when fingerprinting systems, the scanned fingerprint is compared to all known fingerprints in its database. When the fingerprinted system gives a fingerprint that matches exactly with an existing fingerprint from the database, this system is considered to be the same as the one using which the original fingerprint was made. P0f has become quite an advanced tool for obtaining fingerprints and its selection of header information allows for accurate differentiation.

When the operating system is newer or unknown or maybe slightly tweaked, there will be no match and the system will be considered as unknown. In this situation it is necessary to use the trained classifier described in the next section, Section 3.4, to detect the operating system.

3.4 Advanced fingerprinting using machine learning

When no exact match for a fingerprint is available it is possible to use heuristics to find a match among the dataset of already known operating systems. Using the already existing extensive dataset, a trained classifier is able to give an accurate match for fingerprints that do not have an exact

match. A classification algorithm is used to determine the closest match among the known classes.

There are many different types of classification algorithms. The classification algorithm must meet several requirements, namely: support multiple classes, as there are several operating systems, and support high-dimensional data, with more than five analysis dimensions.

Three different algorithms of three different types were chosen: a functional algorithm: SVM, a rule based algorithm: RIPPER and a tree based algorithm: C4.5. The primary reason to choose these, is because they are known to give high-quality classifications, while they also perform good. Especially RIPPER is known to do well in network traffic oriented contexts. Also SVM outperforms competing algorithms in 50% of the tests and ranks in the top 3 in 90% of them. [MLH03, Lee99, LFM+02, Qui93]. Using known high performing algorithms of different classes is also a better idea than using several algorithms of the same type which do something very similar. In this way they are meant to compensate for each others weaknesses making it possible to determine which algorithm is more suitable for this context. They also have a fast retrain phase, so they can be quickly retrained when new OSs are added to the classification. They support multiple classes and high-dimensional input data. This is necessary because there are multiple OSs which are all modeled as different classes and the number of attributes which are derived from the input data is large. Although the full dataset needs to be kept to be able to retrain, the dataset can be stored in a very compact way, so the storage requirements are not very high.

The algorithms implement supervised-learning techniques, which means they use a specific machine learning technique to deduce a function to a predefined specific classification rather than seeking how data is organized by measuring inter-data similarity and let the algorithm form associations itself. Supervised-learning algorithms generally achieve a better result than unsupervised-learning algorithms. The learning process is non-incremental, which means the algorithm goes over the samples of the initial training set multiple times to build an optimal classification model. Because all the known test data needs to be stored for the previous matching algorithm anyhow, it is always possible to let the classifier retrain itself when adding new samples. A full retraining is necessary every time new samples need to be incorporated. The advantage over an incremental algorithm is that

non-incremental algorithms have significantly higher classification precision.

More details on the different algorithms will now be described.

3.4.1 RIPPER

RIPPER stands for Repeated Incremental Pruning to Produce Error Reduction and is an optimized version of IREP. It was proposed by William Cohen [Coh95]. In addition to the optimizations, another enhancement over IREP is its support for targets with multiple values. The original IREP could only work with boolean values. This is necessary in this case, because there are several fields which can contain multiple values and some fields even contain numerical values, such as when window scaling is used.

The algorithm has an initialization phase in which it creates its ruleset. These rules can then be applied to classify data.

The initialization works as follows. RIPPER starts with an empty ruleset. When creating a rule, the training data is divided into two sets. A growing set, consisting of two thirds of the data and a pruning set consisting of the final third. Using the growing set, rules are grown and added to the ruleset when they significantly change the matching. A rule is grown by adding one condition at a time until there are no errors on the growing set. Then an optimization phase is done where two variants of each rule are created from randomized data using a similar procedure as the original, but one variant is generated from an empty rule and the other is generated by adding antecedents to the original rule. From these the rule with the smallest error rate is picked.

An example of a rule from a ruleset generated by RIPPER:

```
IF
    (wss_divisible = mss) and (tcpopt_wscale <= 6) and (initial_ttl <= 64)
THEN
    class = linux
```

All rules produced by RIPPER consist of a binary condition which consists of one or more individual conditions each based on a single attribute. Only when all the conditions hold, the rule applies, so the individual conditions are ANDed together.

3.4.2 The C4.5 decision tree algorithm

The C4.5 algorithm was designed by Ross Quinlan [Qui93]. He based it on his original ID3 algorithm, to which he added some extra features including handling of both continuous and discrete attributes. Continuous attributes are supported by using thresholds. It can handle training data with missing attribute values. It can handle attributes with differing costs based on the number of samples. It prunes trees after creation to get rid of branches that will not help the classification process.

It generates a decision tree that can then be used to classify systems. Such a tree is built in the following way. A recursive algorithm is used with the following base cases: All samples belong to the same class; None of the features provide any information gain. When the algorithm encounters an instance of new class, it goes back up the tree. The algorithm then checks which attribute will give the highest information gain and create a decision node in the tree for that attribute. Then the algorithm will recurse down this tree. When the algorithm is finished, it will prune the tree, trying to remove branches that do not help by replacing them with leaves.

A part of a generated J48 tree:

```
packet_size ≤ 60
| tcpopt_wscale ≤ 1
| | packet_size ≤ 44:  linux
| | packet_size > 44
| | | tcpopt_wscale ≤ 0:  windows_xp
| | | tcpopt_wscale > 0:  bsd
```

Quinlan also created a successor to the C4.5 algorithm called C5.0. This algorithm has not been made public however and is marketed commercially [Qui09].

3.4.3 Support Vector Machines (SVM)

The original SVM by Vapnik and Lerner was a binary classifier [VL63]. It is an algorithm which maps the training set into a higher dimension. Then the most optimal linear separating hyperplane is sought, which separates each class as accurately as possible. This process is also shown in Figure 3.4. The dotted lines, the margins, are pushed against the data sets to obtain

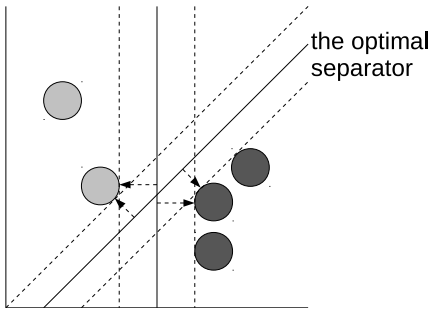


Figure 3.4: Finding the optimal linear separation

the largest separation. Using these, the plane with the maximum margin is found. Once a model has been calculated, new instances are classified by mapping them in the same grid and determining the class based on the existing separators.

The version of SVM used in this case is based on a radial basis function which can also classify non-linear relations. This version was later introduced by Vapnik together with Boser and Guyon and can make use of several different non-linear functions [BGV92]. The support for multiple classes was added by reducing them all to multiple binary sub-problems.

3.4.4 Performance Highlights

Table 3.4 gives some highlights from the results. It is possible to achieve an accuracy of nearly 80% using the above described algorithms. The rule based algorithm RIPPER performs especially well and is accurate. Tree based algorithms like C4.5, appear to be most suitable for this job. The performance of SVM was low and the classification was less accurate than that of C4.5. It appears an algorithm like SVM is not very suitable for this type of classification. During the tests both RIPPER and C4.5 were able to create a model within a second and use it to classify almost 80% correctly. SVM took more than 2 minutes to create a model with which it also classified almost 80% correctly. The classification process with SVM is significantly longer however as it takes hours to do a 10 fold check as opposed to minutes for RIPPER and C4.5. The full performance results are in Section 4.2.2.

Table 3.4: Accuracy and execution time of the different algorithms

Algorithm	Correct	Incorrectly	Execution time
RIPPER	79.56%	20.44%	18.64 seconds
C4.5	79.80%	20.20%	5.83 seconds
SVM	79.73%	20.27%	1798.02 seconds

3.4.5 Conclusion

The steps necessary to automate passive operating system fingerprinting further are feasible and have the potential to result in a system which may discover new OSs without the support from humans. Of course these systems will still be labeled as tweaked or newer versions of current systems, however they will not be treated as being completely unknown. In the end also the database upkeep will be significantly simplified, because it will only consist of tagging the new systems instead of generating a complete fingerprint manually. This should encourage the maintenance of databases and prevent them from becoming outdated. This should make passive OS fingerprinting significantly more accurate in the long term.

Chapter 4

Implementation & Tests

4.1 Implementation

4.1.1 Information extractor

To detect what operating system is running, information from the network and the transport layer is used. This is the most suitable layer for OS fingerprinting. Differences in the physical layer and data link layer are more dependent on the used hardware and can only be measured in the local area network. This makes these layers unsuitable for fingerprinting. The layers above the application data tend to contain only application specific information as was described in Section 2.3.1, and thus are also not suitable for fingerprinting. The data from the network layer and transport layer is the lowest layer that makes it across the network and the implementation is very often OS dependant. Thus from the traffic data, the network and transport layer information needs to be extracted.

Libpcap

To be able to determine which headers should be processed and to extract the right information a custom-developed tool was created based on libpcap [TCP10]. Libpcap is a library to easily read and select data from network traffic. It has data structures for all kinds of traffic data, can read out the packet and segment headers, and also read out their individual fields. Using this library, the header data is read and fingerprint entries are generated in the p0f format. This component automates the steps described in Section

3.2. The tests were all based on SYN fingerprinting, so that the performance could be easily compared to p0f. The details of this analysis will be described in the following section.

Processing a packet

To process a packet, the tool goes through several steps in sequential order. The input data will be a traffic stream of network data. This can either be realtime data or data stored in pcap logs. The pcap format is a simple format used by libpcap and many other programs in which network traffic can be stored. From this data only the headers on the network and transport level are interesting.

First it checks whether a packet starts with an IP header. This can either be an IPv4 or an IPv6 header. If the header is not an IP header, the packet is discarded. Then the tool checks whether the included segment is a TCP segment and without a TCP segment the packet is also dropped.

When the tool has reached this point, it will start analyzing the headers, both the IP and the TCP headers. The tool analyzes and fingerprints every single TCP segment. In practice it can also discard any non-SYN segment at this point, as only SYN segments are interesting. The original tool did not do this, to allow easy observation of the details that could possibly be obtained from different segment types. When observing the analyzed general traffic (details in Section 4.2.1) by looking at the generated output, almost no differences could be seen. Sometimes the TTL values would differ, but options were never included and fingerprints are indistinguishable from each other. This means that general traffic does not show any significant differences between different stacks and only specific TCP segments, such as SYN segments contain useful fingerprinting information.

The header analysis consists of several steps including:

- Rounding up the hop limit (TTL) to the first power of 2 (32, 64, 128, 255) to guess the original value, as explained in Appendix E.2
- Check the state of the DF flag
- Calculating the overall packet size
- Process all the TCP options to determine their type and parameters, analyze the usage of NOP and EOL options

Table 4.1: An automatically generated fingerprint

OS genre	OS descr	WSS	TTL	D	size	options	quirks
@/172.16.246.128	Auto generated	65535	64	1	64	M*,N,W0,N,N,T,S.	-

- Look for special quirks as described in Appendix E.4

After this analysis it will generate a p0f compatible fingerprint. The final two fields are filled in differently however as these consist of the OS family and version. Since these are still unknown at this time, they are filled with information data such as the IP address of the origin and a note the fingerprint was generated automatically. A typical result looks like Table 4.1.

4.1.2 Existing fingerprint matching

The data that was generated in the previous section is suitable to allow for comparison. When comparing it with another instance generated by the same software, it should result in an exact match. When the fingerprinting results in an exact match with an existing fingerprint, the system details of the existing fingerprint are returned, for example "Windows XP SP1+, 2000 SP3". The database contains unique fingerprints, whenever two systems would "share" one fingerprint, they are also likely used the same stack and are listed as one entry. This works similarly to p0f as explained in Appendix E.4.

4.1.3 Classifier

The classifier is triggered when no exact match can be made with an existing fingerprint. The normalized data will now be fed into the classifier. The classifier will return a closest result from its training set which should be a close match to the fingerprinted system. An example is "Windows XP" indicating it is likely a tweaked version or variant of this system. This section will describe the details of the classifier.

Weka

Weka (Waikato Environment for Knowledge Analysis) is a tool which includes many different machine learning algorithms and allows a user to easily

test a single dataset against any of the included machine learning algorithms [HDW94]. This allows one to quickly determine which algorithm is the most optimal for a specific task.

Preparing data for classification

To be able to use Weka, it is necessary to have a representative amount of data in its format, called the Attribute-Relation File Format (ARFF). A converter was written to convert p0f-type fingerprints to ARFF. This converter accepts a file with one or more p0f fingerprints and outputs a specific ARFF format.

ARFF files first start with a header which describes the format. Then all occurrences are stored with comma separated values each on their own line. To use a training set built from one source with another, both ARFF files need to have the exact same header. The format used was heavily based on the p0f rules. For all fields in a p0f fingerprint, separate attributes are defined. The ARFF format has no notion of order. Because the order of TCP options is an important characteristic to differentiate systems, it was necessary to encode order in ARFF too. This was done by specifying ten separate attributes for every option and allowing each of them to have any of the options. Unfortunately this still does not represent the true order of the options as it only shows the fixed positions and the algorithms likely are unaware of any order. All the quirks are simply defined as separate booleans.

The final attribute in an ARFF file is always the result. In this case, it represents which type of system generated the packet. When fingerprinting a new system, this is the attribute that will be determined by the system.

The full ARFF header used for all the data can be found in Appendix C.

Determining relevant factors

The first part of the problem is determining which information of the segment and packet headers is actually needed to obtain information about the OS.

The relevant factors for the classifier should be chosen dynamically as new fingerprints may need to depend on properties of the header which may

not have been considered before to be able to differ from existing fingerprints. This dynamic determination should be done automatically. This is possible by feeding the header or a sanitized version of the header plus the classification to a machine learning algorithm. Once the algorithm has been fed enough classification data, it should be able to classify unknown segments as coming from a specific operating system.

Together with a suitable dataset it is easy to do different test runs with different algorithms and discover the most optimal algorithm. To test the different algorithms one needs to obtain a large dataset which is already correctly classified. Using this data Weka will let the algorithm train itself and check different parts of the data against the training and note the overall accuracy.

As the attributes are refined and more details about the traffic are provided, it is possible to improve the accuracy. The attributes are now largely based on p0f. Representing the full packet and segment headers in the attributes would be the most ideal, although the effect on the performance of RIPPER or other algorithms is unknown.

The most relevant factors in the test data were determined using Principal Component Analysis. Removing specific attributes from the testing data, each time the accuracy of RIPPER was checked. It turned out that the main differences in the testing dataset came from the window scale and packet size.

4.2 Tests

4.2.1 Collecting a dataset for testing

To test how the tool performs with a large sample size, a large amount of test data was collected. To perform the research, 250 GB of traffic data collected over 3 days from the university's wireless network was provided. To build up a large set of classified fingerprints, a tailored tool was created which can correlate SYN segments with HTTP useragents. HTTP is an application layer protocol that is generally used by web browsers and other web based applications. When an HTTP client sends out a request, it starts with a header which contains some information about the useragent itself and the platform it is running on. The tool fingerprints all packets which include a TCP segment with destination port 80 and keeps them in a hash

map with the hash of the source IP address as the hash key. Then when the fingerprinter comes across a segment which includes HTTP header data, it analyzes the useragent header field, takes the OS information from it, and correlates this with the existing fingerprint in the hash map. In this way more than 100.000 fingerprints with classification data were obtained from this 250 GB of traffic, containing 62 different system types. It was this dataset that was used to test the different algorithms. Because this dataset is based on a recent capture, it contains fingerprints for all the systems that are popular at this time. This makes it very suitable for testing the classification, because it covers the systems that are indeed generally used. Because HTTP useragents are sent out by connecting clients, the resulting dataset mainly consists of typical client operating systems.

4.2.2 Testing classification

With the obtained dataset, classification tests were performed using Weka. Weka first lets the algorithm train itself using the data set and then uses different subsets to measure the performance of the trained algorithm. This is done using a process called cross-validation. The complete set of samples is partitioned into subsets. A single subset is used to validate the model, while the other subsets are used to train the model. In this case, a ten-fold cross-validation is done, so ten subsets are created. The complete process is repeated ten times, each time with a different subset used as the validation subset and the rest as the training data for the model as visualized in Figure 4.1. The final conclusion is the combination of the separate results. The results of these tests are in Appendix B.

It turns out that tree-based classifiers are both the fastest and the most accurate when dealing with this type of data. Rule-based classifiers also work well. Since the relevant attributes may change in the future, the optimal algorithm may also change, but it is likely that tree based classifiers will stay the most optimal for now. This also confirms Arkin's research. He based his active ICMP based fingerprinter on a hardcoded logic tree, because he found it was the most optimal approach [Ark02]. Other algorithms give inferior results with both performance and accuracy.

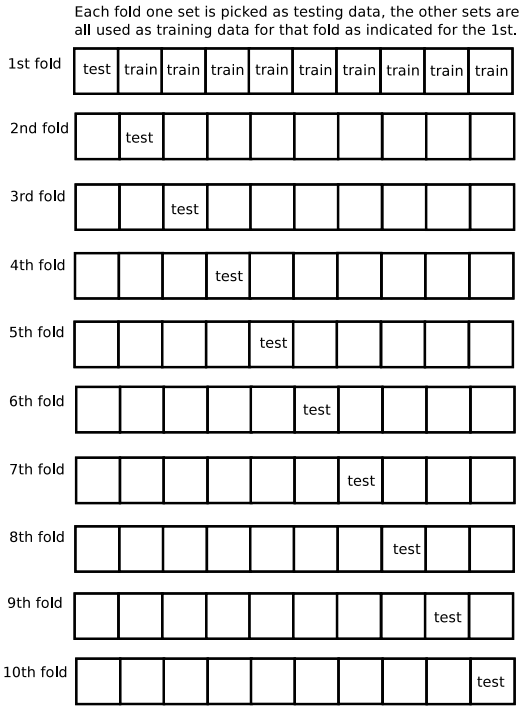


Figure 4.1: 10-fold cross-validation

Diving into the generated classifiers

The details of two classifiers are compared to an existing p0f rule of a known OS. This shows the effectiveness and correctness of the result of the classification algorithms.

This is an example of the ruleset generated by RIPPER:

```
IF
    (wss_divisible = mss) && (initial_ttl <= 64)
    ||
    (packet_size >= 76)
THEN
    class=linux

IF
    (packet_size <= 52) && (initial_ttl <= 64) && (tcpopt_wscale >= 2)
THEN
    class=windows_7

IF
    ( (tcpopt_wscale <= 0) && (packet_size >= 64) && (wss_divisible = no) )
    ||
    ( (tcpopt_wscale <= 0) && (tcpopt_eol = FALSE) &&
      (initial_ttl <= 64) && (packet_size >= 52) )
```

Table 4.2: An example from p0f's database

OS genre	OS descr	WSS	TTL	D	size	options	quirks
Windows	XP SP1+, 2000 SP3	S44	128	1	48	M*,N,N,S	-

```
    ||
    ( (tcpopt_wscale <= 0) && (initial_ttl >= 128) &&
      (packet_size <= 52) && (wss_divisible = mss) )
    ||
    ( (packet_size <= 48) && (initial_ttl >= 128) )
    ||
    ( (initial_ttl >= 128) && (packet_size >= 64) && (wss_divisible = no) )
    ||
    ( (packet_size <= 52) && (tcpopt_wscale >= 3) && (tcpopt_wscale <= 3) )
THEN
    class=windows_xp

IF
    ( (tcpopt_eol = TRUE) )
    ||
    ( (tcpopt_wscale <= 1) && (tcpopt_wscale >= 0) )
THEN
    class=bsd

ELSE
    class=windows_vista
```

For each system it comes up with one or more rules which classify the system. These roughly correspond to p0f's database entries, except that now for each system only the specific relevant attributes are noted. To compare, one line from p0f's database is shown in Table 4.2 and one of the rules generated by RIPPER is shown below. Both are generated to recognize MS Windows XP.

```
IF
    ( (tcpopt_wscale <= 0) && (initial_ttl >= 128) &&
      (packet_size <= 52) && (wss_divisible = mss) )
THEN
    class=windows_xp
```

As can be seen both rules are indeed likely to be from the same system. There is no WS specified, so this value is 0 for RIPPER. The initial TTL is equal, in both cases (at least) 128. The packet size is 48 in the p0f rule, while RIPPER accepts anything smaller than 52. In both cases a packet size of 48 will make the statement true. The final statement in the rule specifies the WSS should be divisible by the MSS. This is also true for the p0f rule, as it specifies an S. This shows that the generated rules by RIPPER indeed match with the fingerprints as they are known by p0f. To observe

Table 4.3: Accuracy and execution time of the different algorithms

Algorithm	Correct	Incorrectly	Execution time
RIPPER	79.56%	20.44%	18.64 seconds
C4.5	79.80%	20.20%	5.83 seconds
SVM	79.73%	20.27%	1798.02 seconds

the accuracy, the summary of the report shows the exact results:

```
Correctly Classified Instances       83208               79.5594 %
Incorrectly Classified Instances     21378               20.4406 %
```

Which shows directly how well the classifier has performed. For comparison, here is the summary when using the hand-crafted p0f rules as input for the same dataset:

```
Correctly Classified Instances       32694               31.2604 %
Incorrectly Classified Instances     71892               68.7396 %
```

The same example also works for the tree based decision algorithm C4.5. The corresponding part of the generated decision tree looks like this:

```
wss_divisible = mss
|   tcpopt_5 = s: windows_xp
```

The full tree is shown in Appendix B.2. As can be seen, with a smart tree, accurate classification can be performed very fast.

4.2.3 Test Analysis

The accuracy of the different algorithms is comparable. The execution time of the algorithm differs significantly however both can be seen in Table 4.3. This shows that SVM has a very significant performance disadvantage and the tree based C4.5 is both the fastest and the most accurate algorithm.

Windows Vista and Windows 7 are basically behaving the same. Since those two operating systems are basically the same, a higher score is obtained when treating both as a single system. The scores are in Table 4.4.

Other mix-ups are likely to be caused by tweaked systems, while passive OS fingerprinting is really focused on the default connection parameters. Since with some tweaking almost everything can be changed on most operating systems, such mix-ups are unavoidable. Because of the large amount of different possible configurations for a typical network stack, the detection can only target the default settings of systems.

Table 4.4: Accuracy of the different algorithms with MS Windows Vista and 7 merged

Algorithm	Correct	Incorrectly	Execution time
RIPPER	86.42%	13.58%	43.61 seconds
C4.5	86.63%	13.37%	5.15 seconds
SVM	86.57%	13.43%	1245.2 seconds

Table 4.5: RIPPER's performance

correct	incorrect	percentage	name	notes
21085	5455	79,4 %	BSD & variants	Most mistakes were tagged as Windows XP and Windows Vista, which are based on the same stack
7608	3	100 %	Linux	
0	**16**	0 %	Solaris	Were all detected as Windows Vista
0	**14**	0 %	Windows 2000	Few occurrences, were all tagged as Windows XP or Windows Vista
12115	1547	88,7 %	Windows XP	All mistakes were tagged as Windows Vista or Windows 7 (maybe patchlevel related?)
41573	4454	90,3 %	Windows Vista	All mistakes were tagged as Windows XP or Windows 7
827	**9730**	7,8 %	Windows 7	The majority of the mistakes was tagged as Windows Vista, the rest as Windows XP
0	**127**	0 %	Windows CE	Appears to use stacks based on either BSD and the variant used by Windows Vista
83208	21378	79.56 %	total	The majority of the systems is classified correctly

4.2.4 Test results

This section contains detailed reports about the performance of the different algorithms. The operating systems shown are the ones that were used to access the network.

RIPPER took 18.64 seconds to build the model. Table 4.5 shows the accuracy per system and the performance summary from Weka. Because the algorithms always use the classification with the highest number of instances as the result when no rule matches. In the used dataset, Windows Vista had the highest presence. For this reason there are systems with very few instances like Solaris, that are classified as Windows Vista.

C4.5 took 5.83 seconds to build the model. Table 4.6 shows the accuracy per system and the performance summary from Weka.

SVM took 1798.02 seconds to build the model. Table 4.7 shows the accuracy per system and the performance summary from Weka.

Table 4.6: C4.5's performance

correct	incorrect	percentage	name	notes
21336	5204	80,4 %	BSD & variants	Most mistakes were tagged as Windows XP and Windows Vista, which are based on the same stack
7607	4	99,9 %	Linux	
0	**16**	0 %	Solaris	Were all detected as Windows Vista
0	**14**	0 %	Windows 2000	Few occurrence, were all tagged as Windows XP or Windows Vista
12115	1550	88,7 %	Windows XP	Almost all mistakes were tagged as Windows Vista or Windows 7 (maybe patchlevel related?)
41574	4477	90,3 %	Windows Vista	Almost all mistakes were tagged as Windows XP or Windows 7
827	**9735**	7,8 %	Windows 7	The majority of the mistakes was tagged as Windows Vista, the rest as Windows XP
0	**127**	0 %	Windows CE	Appears to use stacks based on BSD
83459	21127	79.80 %	total	The majority of the systems is classified correctly

Table 4.7: SVM's performance

correct	incorrect	percentage	name	notes
21336	5204	80,4 %	BSD & variants	Most mistakes were tagged as Windows XP and Windows Vista, which are based on the same stack
7608	3	100 %	Linux	
0	**16**	0 %	Solaris	Were all detected as Windows Vista
0	**14**	0 %	Windows 2000	Few occurrences, were all tagged as Windows XP or Windows Vista
12044	1621	88,1 %	Windows XP	Almost all mistakes were tagged as Windows Vista or Windows 7 (maybe patchlevel related?)
41574	4477	90,3 %	Windows Vista	All mistakes were tagged as Windows XP or Windows 7
829	**9733**	7,8 %	Windows 7	The majority of the mistakes was tagged as Windows Vista, the rest as Windows XP
0	**127**	0 %	Windows CE	Appears to use stacks based on either BSD and the variant used by Windows Vista
83391	21195	79.73 %	total	The majority of the systems is classified correctly

Table 4.8: Three Linux 2.6 fingerprints from p0f's database

OS genre	OS descr	WSS	TTL	D	size	options	quirks
Linux	2.6 (newer, 1)	S4	64	1	60	M*,S,T,N,**W5**	-
Linux	2.6 (newer, 2)	S4	64	1	60	M*,S,T,N,**W6**	-
Linux	2.6 (newer, 3)	S4	64	1	60	M*,S,T,N,**W7**	-

Table 4.9: Comparing fingerprints from MS Windows XP and Vista

OS genre	OS descr	WSS	TTL	D	size	options	quirks
Windows	XP SP1+, 2000 SP3	S44	128	1	48	M*,N,N,S	-
(Windows)	(Vista)	8192	128	1	52	M*,N W8,N,N,S	-

4.3 Preliminary future work

4.3.1 Detecting progression in network stacks

An interesting feature is the detection of newer versions of known operating systems. As operating systems evolve, their network stacks are also improved and sometimes even replaced adding new features and options. The effect this has on the resulting packet and segment headers, is exactly what allows fingerprinting to differentiate operating systems.

Table 4.8 is an example of a part of p0f's fingerprint database [Zal06a]. The trend in the table is obvious. The default value of the parameter of the WS option is apparently increased over time. It turns out that there are indeed explicit trends where new features or options are implemented similarly throughout time for different operating systems. The most notable is the TCP Window Scale (WS) option [JBB92]. This option was introduced to deal with "Long-Fat Networks" (LFN). As networks get faster, more and more operating systems start to use the WS option to deal with LFNs. And as links get even faster, newer systems keep increasing the value of the window scale. The result of this is that systems which use the WS option or use it with a higher value, but are otherwise comparable to older fingerprints are likely to be newer versions of the systems these older fingerprints were based on. A quick look through p0f's fingerprint database already showed this pattern to be true for Linux. Detailed testing confirmed the same phenomenon on MS Windows as can be seen in Table 4.9. The table shows the addition of the WS option. It should be noted that since the WS option plus its parameter often consists of 24 bits and many systems align the TCP options to 32 bits, the addition of the WS option may also bring an

extra no-operation (NOP) option. This is also the case for Windows' stack. Other than that, the packet size is incremented by 4 to take the extra 32 bits (= 4 bytes) into account. Since the WSS value is related to the WS, it obviously changes too. Basically this means that all the expected values have changed in a way that is logical and thus likely also predictable. This could be detected using heuristics. Details of this test are in Section 4.3.2.

With options like WS the evolution of OSs and the evolution of the network stack is really visible in the fingerprinting and this information can be used to predict the fingerprints enhanced systems will have. Because such options are often already specified long before they appear in implementations (and before they are turned on by default), one can possibly even predict the options that may appear in future versions of existing network stacks by watching the RFCs from IETF or studying which options are actually implemented already, but which have not been enabled yet. As with the case of the WS option, newer versions are likely to have higher values for the WS, so also this is a good indicator of a new version of an already known system.

Although the classification algorithms as described above cannot be made explicitly aware of these progressions, this enables understanding how progression can be detected by them.

4.3.2 Detecting clients versus detecting servers, SYN fingerprinting against SYN+ACK fingerprinting

Most of the work that has been gone into passive operating system fingerprinting was based on TCP SYN segments. Because a SYN segment is sent out by a client when it wants to initiate a connection. This is useful when someone offering online services wants to fingerprint the operating systems her users are using. In other cases however, it may be more interesting to fingerprint the server instead of the client. In this case, SYN+ACK fingerprinting is required. A SYN+ACK segment is a segment that is sent out by a server in the three-way handshake (explained in Appendix E.1).

An example of a SYN+ACK fingerprint from p0f's database is shown in Table 4.10. As can be seen, these fingerprints are roughly equal to p0f's SYN fingerprints. One notable thing is the A quirk. It is always present for SYN+ACK segments, because they should always have the ACK flag set. As described also by p0f's author, SYN+ACK segments are often based on

Table 4.10: A SYN+ACK fingerprint from p0f's database

OS genre	OS descr	WSS	TTL	D	size	options	quirks
Windows	XP SP1	S44	64	1	64	M*,N,W0,N,N,T0,N,N,S	A

Table 4.11: A response to a SYN with the WS option

OS genre	OS descr	WSS	TTL	D	size	options	quirks
(Linux)	(2.6)	5792	64	1	60	M*,S,T,N,W6	ZAT

the SYN segment that initiated the connection [Zal06a]. This means that differences in SYN+ACK segments may actually result from differences that were in the SYN segment which the SYN+ACK segment is a reply to. If, for instance, a SYN segment is sent without any WS option, the answer is unlikely to contain a WS option, since it assumes the first system does not have support for this option anyhow. This is shown in the following example. First, a normal connection with default settings is established from one Linux system to another and then the WS option is disabled on the client side. Compare the responses. The response to a packet with the WS option is shown in Table 4.11. The response to a packet without the WS option is shown in Table 4.12. This shows the adaptive behavior of the stack on the connection accepting side. The Z quirk in this case is triggered by the DF flag which is consistently set in these cases, so IP identification is not needed. The details on how this test was performed are described in the following Subsection. Thus, when fingerprinting one should realize the SYN+ACK segment may have been based on a SYN segment with a limited option set specified and so it may not reveal many details. Other values that are likely to depend on the initiating SYN segment are the WSS and the MSS. Taking these into account however, there are still many things that can be used to differentiate systems based on SYN+ACK segments, especially when multiple SYN+ACK segments are collected based on different initiating SYN segments. With a feature rich SYN segment passing by, the corresponding SYN+ACK will likely reveal a lot of information about the supported

Table 4.12: A response to a SYN without the WS option from the same system as in Table 4.11

OS genre	OS descr	WSS	TTL	D	size	options	quirks
(Linux)	(2.6)	5792	64	1	56	M*,S,T	ZAT

Table 4.13: SYN packet with WS enabled

OS genre	OS descr	WSS	TTL	D	size	options	quirks
(Linux)	(2.6)	S4	64	1	60	M*,S,T,N,W7	

Table 4.14: SYN+ACK response to SYN with WS enabled

OS genre	OS descr	WSS	TTL	D	size	options	quirks
(Linux)	(2.6)	5792	64	1	60	M*,S,T,N,W6	ZAT

feature set. Together with some segments which advertise less features, patterns such as NOP-aligning may be recognized. With other variants, other behavior may be noted. P0f's fingerprinting is completely based on dealing with a single segment. No correlation between corresponding segments is performed. In this situation it indeed appears as if SYN+ACK fingerprinting is also partially fingerprinting the OS which initiated the connection. With more knowledge of state and a proper database, SYN+ACK fingerprinting may be potentially quite thorough and complicated, but still feasible with a good design.

Showing the SYN+ACK TCP options can depend on initial SYN segments

As stated in Section 4.3.1, the options a TCP stack will return in a SYN+ACK segment often depend on which options were specified. This can be verified using the following test, which consists of connecting to a single server using a different option set specified in the initial segment.

It was earlier described that the window scaling feature is only advertised by some servers when the initial connection also advertises this feature. To test this, the following commands can be used to disable or enable the window scaling feature on the fly.

```
echo 0 | tee /proc/sys/net/ipv4/tcp_window_scaling
echo 1 | tee /proc/sys/net/ipv4/tcp_window_scaling
```

When the window scaling setting is enabled, a SYN packet sent out by a system running Linux kernel version 2.6.31, has the properties as shown in Table 4.13. The response from a system using Linux kernel version 2.6.24 is shown in Table 4.14. Then the WS option is disabled in the network stack, the initial segment is now as shown in Table 4.15. The response from the same system is now as shown in Table 4.16.

Table 4.15: SYN packet with WS disabled

OS genre	OS descr	WSS	TTL	D	size	options	quirks
(Linux)	(2.6)	S4	64	1	56	M*,S,T	

Table 4.16: SYN+ACK response to SYN with WS disabled

OS genre	OS descr	WSS	TTL	D	size	options	quirks
(Linux)	(2.6)	5792	64	1	56	M*,S,T	ZAT

More testing was done using varying window sizes. It is possible to decrease the WS to 6 (from 7) using these commands:

```
echo 4096 8192 2097152 | tee /proc/sys/net/ipv4/tcp_rmem
echo 4096 8192 2097152 | tee /proc/sys/net/ipv4/tcp_wmem
```

Using decreased numbers the WS can be lowered further. However this does not have any effect on the WS of the replying system in our tests. This was tested against Linux kernel version 2.4.27, version 2.6.26 and Windows Server 2008. This shows that comprehensive SYN+ACK fingerprinting indeed gives results as described and could be used to fingerprint servers more comprehensively.

4.3.3 RST+ACK fingerprinting

When fingerprinting the server-side of a connection, RST+ACK fingerprinting may also be relevant. When a system does not accept a connection using a SYN+ACK segment, it will usually send back a RST+ACK segment. This means that when aiming to detect the running system on a server, RST+ACK fingerprinting may also be necessary.

In practice there is not much information that can be gained from RST+ACK segments however and also p0f's database for them is very limited. P0f has a single mode in which it fingerprints all types of RST segments, including RST+ACK segments. P0f's RST database contains just five RST+ACK fingerprints, which shows how limited the detecting possibilities are. Two are shown in Table 4.17. As can be seen, the fingerprints

Table 4.17: Two RST+ACK fingerprints from p0f's database

OS genre	OS descr	WSS	TTL	D	size	options	quirks
Linux	2.0/2.2	0	255	0	40	-	K0A
Windows	XP/2000	0	128	0	40	-	K0A

are very similar, with the only differentiating factor being the initial TTL. This is also the reason the different versions cannot be differentiated. It is thus unlikely that RST+ACK fingerprinting can contribute significantly compared to SYN+ACK fingerprinting.

Chapter 5

Conclusion

The original research question of the work was formulated as:

"Can machine learning techniques be used to recognize new operating systems which current fingerprinting systems cannot automatically recognize?"

It is argued that current passive fingerprinting systems have problems with recognizing modern systems and that this causes the accuracy of these systems to be very limited. To improve the accuracy, it is necessary to either update the fingerprint database manually or use machine learning techniques to deal with newer systems. Because it was observed manual updating is too complex, there is a necessity to use classification.

In the thesis, an architecture is described which extends the traditional passive operating system fingerprinting with additional classification functionality. First the process of information extraction and normalization is discussed showing it is possible to fully automate this part either using an automated implementation of the existing process or by using classification to obtain an overview of the differentiation fields. Then the next step in the architecture is described, using the traditional technique, already known systems can be accurately classified. After that, when a system cannot be recognized, classification is used to classify the operating system using existing systems as a training set. After describing the architecture for this process, an implementation is described which was used to test the feasibility of the system. Both the information extraction and exact matching are based on the already existing p0f fingerprinter. The classification process is

defined after testing several types of algorithms and determining which ones were suitable for the classification of operating systems based on passive fingerprinting data. Two algorithms that are very suitable for automatic classification turned out to be C4.5 and RIPPER. Both are accurate enough and also perform fast. The testing section shows classification is indeed quite accurate when using for fingerprinting and suitable to be used to recognize previously unrecognized systems.

5.1 Future work

Several notable developments will have a notable impact on passive OS fingerprinting in the future. Some of these were already mentioned in Chapter 2.

IPv6 The thesis was mainly focused on IP version 4 and did not detail much about version 6. IPv6 adds several field however, such as the flow label, which also have potential to be useful for fingerprinting. More information can be found in Appendix A.

IPsec IPsec will provide full encryption for the transport layer, which would have a significant impact on the described fingerprinting because of its big dependence on TCP. It may become almost impossible to properly fingerprint connection which are set up by two foreign entities, because in that case the TCP header, which is the main differentiation source, is fully encrypted. Maybe IPsec itself will allow for new methods of differentiation however.

Correlation of packet series Right now the complete fingerprinting process is based on just a single packet. Although much more complex, using multiple packets for fingerprinting could make the fingerprinting significantly more accurate. Some initial testing on specific correlation of SYN+ACK responses to SYN segments is already described in Section 4.3.2.

Appendix A

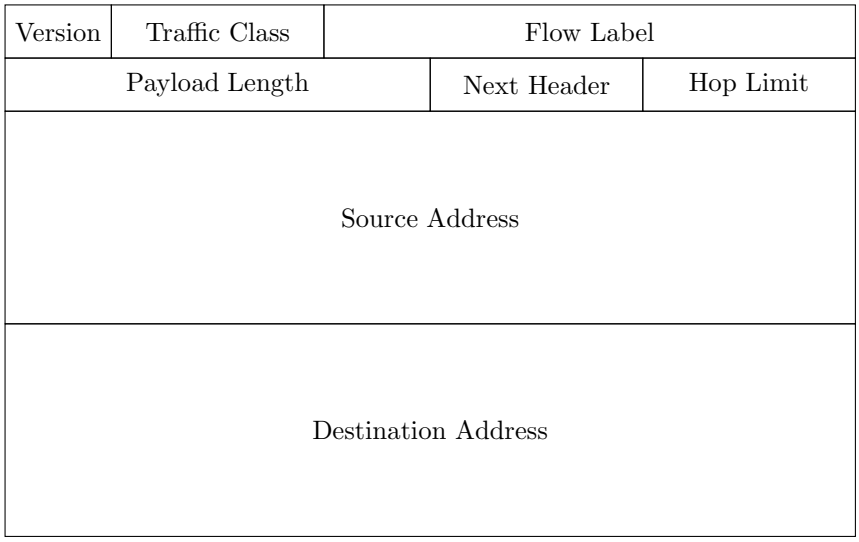
The IP & the TCP

Addressing, routing on the internet are handled by the Internet Protocol (IP) [Pos81b]. Several versions of the IP have been developed. At this moment version 4 and version 6 [DH98, ASNN07] are in widespread use. Both the older IPv4 and newer IPv6 header are shown:

The IPv4 header

<table>
<tr><td>Version</td><td>IHL</td><td>Type of Service</td><td colspan="2">Total Length</td></tr>
<tr><td colspan="3">Identification</td><td>Flags</td><td>Fragment Offset</td></tr>
<tr><td colspan="2">Time to Live</td><td>Protocol</td><td colspan="2">Header Checksum</td></tr>
<tr><td colspan="5">Source Address</td></tr>
<tr><td colspan="5">Destination Address</td></tr>
<tr><td colspan="4">Options</td><td>Padding</td></tr>
</table>

The IPv6 header



This header is used by routers to route a packet to the right destination. Especially the destination address is relevant. Based on the routing tables and the address from the header, packets are routed to their destination. A short overview of all the fields in the IPv6 header is given here. They mostly correspond to the most relevant fields in the IPv4 header.

Version This field is either 6 for IPv6 or 4 for IPv4.

Traffic Class This field specifies the type of traffic.

Flow Label This field is reserved for future use, so hosts can request special handling of certain traffic flows. For example, realtime voice communication.

Payload Length The full length of the packet, minus 40 octets (bytes) (this header).

Next Header Identifies the type of the header that follows the IP header.

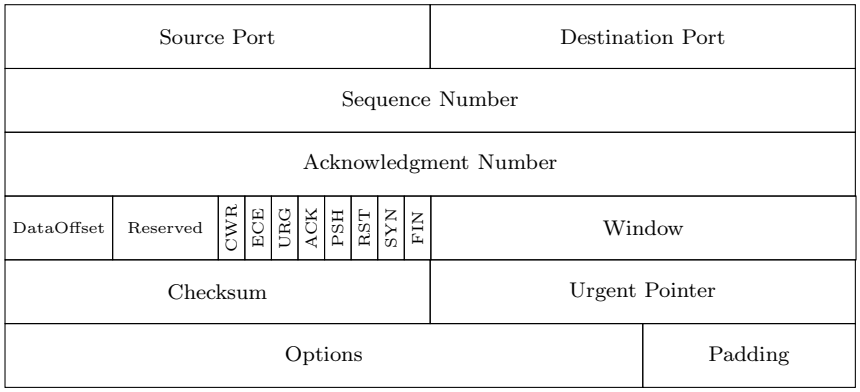
Hop Limit This value used to be called TTL, but because it was always treated as a hop limit, it was renamed. This value is subtracted with one by every router this packet passes through.

Source Address The address from which the packet originates.

Destination Address The address of the intended recipient of the packet.

To ensure reliable communication the Transport Control Protocol [Pos81c, Bra89, RFB01] was developed. It ensures data between two applications arrives without errors and in the original order. The TCP header generally follows the IP header in a packet and looks as shown.

The TCP header



Source Port The port number from which the segment originates.

Destination Port The port number to which the segment is destined.

Sequence Number The sequence number of the first data octet or the initial sequence number.

Acknowledgment Number The sequence number of the next segment the sender is expected to receive.

Data Offset The length of this header in 32 bit words.

Reserved These bits are unused for now and should be set to zero.

Control Bits Specific connection set-up and tear-down related flags and urgency flags. The first two bits are related to congestion notification. [RFB01]

Window The number of data octets starting with the one in the acknowledgment field the sender of the segment is willing to accept.

Checksum A checksum calculated over the TCP header and some field from the IP header.

Urgent Pointer This field points to the sequence number of the octet following the urgent data when the URG flag is set.

Options Advertisement of options supported by a TCP stack.

All systems participating in the internet send out packets containing the above headers. Due to subtle difference in the way they fill these headers, it is possible to distinguish systems and obtain more information about the system from which a packet originates. Doing this technique based on previously observed and correlated packets is called fingerprinting. This is similar to observing and correlating human fingerprints for identification purposes.

Appendix B

Detailed test reports

B.1 RIPPER

```
=== Run information ===

Scheme:       weka.classifiers.rules.JRip -F 3 -N 2.0 -O 2 -S 1
Relation:     ip+tcp_fingerprinting-weka.filters.supervised.attribute.
  AttributeSelection-Eweka.attributeSelection.CfsSubsetEval-Sweka.attributeSelection.BestFirst -D 1 -N 5
Instances:    104586
Attributes:   6
              wss_divisible
              initial_ttl
              packet_size
              tcpopt_wscale
              tcpopt_eol
              class
Test mode:    10-fold cross-validation

=== Classifier model (full training set) ===

JRIP rules:
===========

(wss_divisible = mss) and (tcpopt_wscale <= 6) and (initial_ttl <= 64) => class=linux (6591.0/21.0)
(wss_divisible = mss) and (initial_ttl <= 64) => class=linux (1164.0/137.0)
(packet_size >= 76) => class=linux (11.0/0.0)
(packet_size <= 52) and (initial_ttl <= 64) and (tcpopt_wscale >= 2) => class=windows_7 (910.0/83.0)
(tcpopt_wscale <= 0) and (packet_size >= 64) and (wss_divisible = no) => class=windows_xp (231.0/1.0)
(tcpopt_wscale <= 0) and (tcpopt_eol = FALSE) and (initial_ttl <= 64) and (packet_size >= 52)
  => class=windows_xp (232.0/93.0)
(tcpopt_wscale <= 0) and (initial_ttl >= 128) and (packet_size <= 52) and (wss_divisible = mss)
  => class=windows_xp (23.0/0.0)
(packet_size <= 48) and (initial_ttl >= 128) => class=windows_xp (19773.0/8766.0)
(initial_ttl >= 128) and (packet_size >= 64) and (wss_divisible = no) => class=windows_xp (665.0/4.0)
(packet_size <= 52) and (tcpopt_wscale >= 3) and (tcpopt_wscale <= 3) => class=windows_xp (55.0/0.0)
```

```
(tcpopt_eol = TRUE) => class=bsd (21037.0/12.0)
(tcpopt_wscale <= 1) and (tcpopt_wscale >= 0) => class=bsd (448.0/138.0)
 => class=windows_vista (53446.0/11872.0)

Number of Rules : 13


Time taken to build model: 18.64 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances       83208               79.5594 %
Incorrectly Classified Instances     21378               20.4406 %
Kappa statistic                          0.7037
Mean absolute error                      0.0579
Root mean squared error                  0.1701
Relative absolute error                 44.894  %
Root relative squared error             67.0061 %
Total Number of Instances           104586

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 0.794     0          0.998     0.794     0.885      0.916    bsd
                 1         0.002      0.98      1         0.99       1        linux
                 0         0          0         0         0          0.705    solaris
                 0         0          0         0         0          ?        windows_9x
                 0         0          0         0         0          ?        windows_nt
                 0         0          0         0         0          0.728    windows_2000
                 0.887     0.098      0.577     0.887     0.699      0.914    windows_xp
                 0         0          0         0         0          ?        windows_2003
                 0.903     0.209      0.773     0.903     0.833      0.873    windows_vista
                 0.078     0.001      0.909     0.078     0.144      0.684    windows_7
                 0         0          0         0         0          0.832    windows_ce
Weighted Avg.    0.796     0.105      0.832     0.796     0.769      0.879

=== Confusion Matrix ===

     a     b     c     d     e     f     g     h     i     j     k   <-- classified as
 21085   137     0     0     0     0  1835     0  3483     0     0 |     a = bsd
     2  7608     0     0     0     0     1     0     0     0     0 |     b = linux
     0     0     0     0     0     0     0     0    16     0     0 |     c = solaris
     0     0     0     0     0     0     0     0     0     0     0 |     d = windows_9x
     0     0     0     0     0     0     0     0     0     0     0 |     e = windows_nt
     0     0     0     0     0     0    10     0     4     0     0 |     f = windows_2000
     3     0     0     0     0     0 12115     0  1509    38     0 |     g = windows_xp
     0     0     0     0     0     0     0     0     0     0     0 |     h = windows_2003
     3    21     0     0     0     0  4409     0 41573    45     0 |     i = windows_vista
     5     0     0     0     0     0  2634     0  7096   827     0 |     j = windows_7
    26     0     0     0     0     0     0     0   101     0     0 |     k = windows_ce
```

B.2 C4.5

```
=== Run information ===

Scheme:       weka.classifiers.trees.J48 -C 0.25 -M 2
Relation:     ip+tcp_fingerprinting
Instances:    104586
Attributes:   31
              wss_divisible
              initial_ttl
              df_flag
              packet_size
              tcpopt_individual_nop_align
              tcpopt_nop_start
              tcpopt_mss
              tcpopt_sack
              tcpopt_ts
              tcpopt_wscale
              tcpopt_eol
              tcpopt_0
              tcpopt_1
              tcpopt_2
              tcpopt_3
              tcpopt_4
              tcpopt_5
              tcpopt_6
              tcpopt_7
              tcpopt_8
              tcpopt_9
              quirk_data
              quirk_options_past_EOL
              quirk_zero_ipid
              quirk_ipoptions
              quirk_nonzero_urg
              quirk_nonzeo_unused
              quirk_nonzeo_ack
              quirk_nonzeo_2nd_t
              quirk_weird_flags
              class
Test mode:    10-fold cross-validation

=== Classifier model (full training set) ===

J48 pruned tree
------------------

wss_divisible = no
|   tcpopt_eol = TRUE: bsd (21038.0/12.0)
|   tcpopt_eol = FALSE
|   |   tcpopt_wscale <= 1
|   |   |   initial_ttl <= 64
```

```
|   |   |   |   tcpopt_2 = n
|   |   |   |   |   df_flag = set: windows_7 (2.0)
|   |   |   |   |   df_flag = unset: linux (11.0)
|   |   |   |   tcpopt_2 = m: windows_xp (0.0)
|   |   |   |   tcpopt_2 = s: windows_xp (0.0)
|   |   |   |   tcpopt_2 = t: windows_xp (0.0)
|   |   |   |   tcpopt_2 = w: windows_xp (232.0/93.0)
|   |   |   |   tcpopt_2 = e: windows_xp (0.0)
|   |   |   |   tcpopt_2 = null: windows_xp (0.0)
|   |   |   initial_ttl > 64
|   |   |   |   tcpopt_5 = n: windows_xp (0.0)
|   |   |   |   tcpopt_5 = m: windows_xp (0.0)
|   |   |   |   tcpopt_5 = s: bsd (436.0/138.0)
|   |   |   |   tcpopt_5 = t: windows_xp (230.0)
|   |   |   |   tcpopt_5 = w: windows_xp (0.0)
|   |   |   |   tcpopt_5 = e: windows_xp (0.0)
|   |   |   |   tcpopt_5 = null
|   |   |   |   |   tcpopt_ts = TRUE: windows_vista (7.0)
|   |   |   |   |   tcpopt_ts = FALSE: windows_xp (19773.0/8766.0)
|   |   tcpopt_wscale > 1
|   |   |   initial_ttl <= 64: windows_7 (910.0/83.0)
|   |   |   initial_ttl > 64
|   |   |   |   tcpopt_wscale <= 2: windows_vista (46536.0/10363.0)
|   |   |   |   tcpopt_wscale > 2
|   |   |   |   |   tcpopt_wscale <= 5: windows_xp (720.0/4.0)
|   |   |   |   |   tcpopt_wscale > 5: windows_vista (6901.0/1507.0)
wss_divisible = mss
|   tcpopt_5 = n: linux (0.0)
|   tcpopt_5 = m: linux (0.0)
|   tcpopt_5 = s: windows_xp (23.0)
|   tcpopt_5 = t: linux (0.0)
|   tcpopt_5 = w: bsd (12.0)
|   tcpopt_5 = e: linux (0.0)
|   tcpopt_5 = null: linux (7755.0/158.0)
wss_divisible = mtu: windows_vista (0.0)

Number of Leaves  :  29

Size of the tree :  41


Time taken to build model: 5.83 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances       83459               79.7994 %
Incorrectly Classified Instances     21127               20.2006 %
Kappa statistic                          0.7074
Mean absolute error                      0.0569
Root mean squared error                  0.1687
```

```
Relative absolute error                 44.1343 %
Root relative squared error             66.4392 %
Total Number of Instances           104586

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 0.804     0.002      0.993     0.804     0.889      0.929    bsd
                 0.999     0.002      0.979     0.999     0.989      0.999    linux
                 0          0          0         0         0          0.756    solaris
                 0          0          0         0         0          ?        windows_9x
                 0          0          0         0         0          ?        windows_nt
                 0          0          0         0         0          0.786    windows_2000
                 0.887     0.097      0.577     0.887     0.699      0.919    windows_xp
                 0          0          0         0         0          ?        windows_2003
                 0.903     0.203      0.778     0.903     0.836      0.875    windows_vista
                 0.078     0.001      0.909     0.078     0.144      0.722    windows_7
                 0          0          0         0         0          0.998    windows_ce
Weighted Avg.    0.798     0.103      0.833     0.798     0.771      0.888

=== Confusion Matrix ===

     a     b     c     d     e     f     g     h     i     j     k   <-- classified as
 21336   137     0     0     0     0  1812     0  3255     0     0 |     a = bsd
     2  7607     0     0     0     0     2     0     0     0     0 |     b = linux
     0     0     0     0     0     0     0     0    16     0     0 |     c = solaris
     0     0     0     0     0     0     0     0     0     0     0 |     d = windows_9x
     0     0     0     0     0     0     0     0     0     0     0 |     e = windows_nt
     0     0     0     0     0     0    10     0     4     0     0 |     f = windows_2000
    12     0     0     0     0     0 12115     0  1500    38     0 |     g = windows_xp
     0     0     0     0     0     0     0     0     0     0     0 |     h = windows_2003
     3    21     0     0     0     0  4408     0 41574    45     0 |     i = windows_vista
     6     2     0     0     0     0  2632     0  7095   827     0 |     j = windows_7
   127     0     0     0     0     0     0     0     0     0     0 |     k = windows_ce
```

B.3 SVM

```
=== Run information ===

Scheme:       weka.classifiers.functions.LibSVM
  -S 0 -K 2 -D 3 -G 0.0 -R 0.0 -N 0.5 -M 40.0 -C 1.0 -E 0.001 -P 0.1
Relation:     ip+tcp_fingerprinting
Instances:    104586
Attributes:   31
              wss_divisible
              initial_ttl
              df_flag
              packet_size
              tcpopt_individual_nop_align
              tcpopt_nop_start
```

```
               tcpopt_mss
               tcpopt_sack
               tcpopt_ts
               tcpopt_wscale
               tcpopt_eol
               tcpopt_0
               tcpopt_1
               tcpopt_2
               tcpopt_3
               tcpopt_4
               tcpopt_5
               tcpopt_6
               tcpopt_7
               tcpopt_8
               tcpopt_9
               quirk_data
               quirk_options_past_EOL
               quirk_zero_ipid
               quirk_ipoptions
               quirk_nonzero_urg
               quirk_nonzeo_unused
               quirk_nonzeo_ack
               quirk_nonzeo_2nd_t
               quirk_weird_flags
               class
Test mode:    10-fold cross-validation

=== Classifier model (full training set) ===

LibSVM wrapper, original code by Yasser EL-Manzalawy (= WLSVM)

Time taken to build model: 1697.7 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances       83391               79.7344 %
Incorrectly Classified Instances     21195               20.2656 %
Kappa statistic                          0.7064
Mean absolute error                      0.0368
Root mean squared error                  0.192
Relative absolute error                 28.5761 %
Root relative squared error             75.6003 %
Total Number of Instances           104586

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 0.804     0.002      0.992     0.804     0.888      0.901    bsd
                 1         0.002      0.98      1         0.99       0.999    linux
                 0         0          0         0         0          0.5      solaris
```

```
                 0         0          0         0         0          ?         windows_9x
                 0         0          0         0         0          ?         windows_nt
                 0         0          0         0         0          0.5       windows_2000
                 0.881     0.097      0.576     0.881     0.697      0.892     windows_xp
                 0         0          0         0         0          ?         windows_2003
                 0.903     0.204      0.777     0.903     0.835      0.85      windows_vista
                 0.078     0.001      0.909     0.078     0.145      0.539     windows_7
                 0         0          0         0         0          0.5       windows_ce
Weighted Avg.    0.797     0.103      0.832     0.797     0.771      0.847

=== Confusion Matrix ===

     a     b     c     d     e     f     g     h     i     j     k   <-- classified as
 21336   137     0     0     0     0  1812     0  3255     0     0 |     a = bsd
     2  7608     0     0     0     0     1     0     0     0     0 |     b = linux
     0     0     0     0     0     0     0     0    16     0     0 |     c = solaris
     0     0     0     0     0     0     0     0     0     0     0 |     d = windows_9x
     0     0     0     0     0     0     0     0     0     0     0 |     e = windows_nt
     0     0     0     0     0     0    10     0     4     0     0 |     f = windows_2000
    28     0     0     0     0     0 12044     0  1555    38     0 |     g = windows_xp
     0     0     0     0     0     0     0     0     0     0     0 |     h = windows_2003
     3    21     0     0     0     0  4408     0 41574    45     0 |     i = windows_vista
     6     0     0     0     0     0  2632     0  7095   829     0 |     j = windows_7
   127     0     0     0     0     0     0     0     0     0     0 |     k = windows_ce
```

Appendix C

The ARFF header

This is the exact ARFF header as it was used for the classification data.

```
% IP+TCP stack fingerprints for passive OS fingerprinting using SYN packets
% Largely based on p0f rules

@relation ip+tcp_fingerprinting

@attribute wss_divisible {no, mss, mtu}

@attribute initial_ttl integer

@attribute df_flag {set, unset}

@attribute packet_size integer

@attribute tcpopt_individual_nop_align {TRUE, FALSE}
@attribute tcpopt_nop_start {TRUE, FALSE}
@attribute tcpopt_mss {TRUE, FALSE}
@attribute tcpopt_sack {TRUE, FALSE}
@attribute tcpopt_ts {TRUE, FALSE}
@attribute tcpopt_wscale integer
@attribute tcpopt_eol {TRUE, FALSE}
@attribute tcpopt_0 {n,m,s,t,w,e,null}
@attribute tcpopt_1 {n,m,s,t,w,e,null}
@attribute tcpopt_2 {n,m,s,t,w,e,null}
```

```
@attribute tcpopt_3 {n,m,s,t,w,e,null}
@attribute tcpopt_4 {n,m,s,t,w,e,null}
@attribute tcpopt_5 {n,m,s,t,w,e,null}
@attribute tcpopt_6 {n,m,s,t,w,e,null}
@attribute tcpopt_7 {n,m,s,t,w,e,null}
@attribute tcpopt_8 {n,m,s,t,w,e,null}
@attribute tcpopt_9 {n,m,s,t,w,e,null}

@attribute quirk_data {TRUE, FALSE}
@attribute quirk_options_past_EOL {TRUE, FALSE}
@attribute quirk_zero_ipid {TRUE, FALSE}
@attribute quirk_ipoptions {TRUE, FALSE}
@attribute quirk_nonzero_urg {TRUE, FALSE}
@attribute quirk_nonzeo_unused {TRUE, FALSE}
@attribute quirk_nonzeo_ack {TRUE, FALSE}
@attribute quirk_nonzeo_2nd_t {TRUE, FALSE}
@attribute quirk_weird_flags {TRUE, FALSE}

@attribute class {bsd, linux, solaris, windows_9x, windows_nt, windows_2000,
  windows_xp, windows_2003, windows_vista, windows_7, windows_ce}


@data
```

Appendix D

Other notable results

D.1 Mobile phone operating systems

There has been a large trend recently which brought increased usage of mobile devices to access the internet. Using passive SYN fingerprinting these systems can be fingerprinted similarly to non-mobile operating systems. Most mobile operating systems are based on existing operating system kernels. Platforms such as MeeGo and Android are based on the Linux kernel, the iPhone is in turn running a Mac OS X (BSD) kernel. This means that when fingerprinting such mobile systems, the fingerprints are likely to be very similar or fully correspond to the tested desktop versions of these kernels. Table D.1 shows the fingerprint from an Android system. The fingerprint corresponds exactly to the fingerprint in p0f's database for newer Linux systems. Also a Nokia N900, which is based on the MeeGo platform with a Linux kernel has the same fingerprint.

D.2 Strange window scale behavior on Windows Vista

While testing, strange behavior was observed when using Windows Vista. While numerous logged HTTP SYN segments had a WS of 2, during manual

Table D.1: The fingerprint of an Android system

OS genre	OS descr	WSS	TTL	D	size	options	quirks
(Linux)	(2.6)	S4	64	1	60	M*,S.T,N,W7	-

tests the WS was consistently 8. The initial suspection was that this had been changed over time with updates, so 4 VMs were set up, each running a different version, SP0, SP1, SP2 and Windows 7. All those systems showed the exact same behavior however. After some more searching it turned out this difference was related to the fact certain applications using HTTP, were made to work with a WS of 2 on purpose while other applications would use the WS of 8. The reason behind this separation is still unclear. It has been suggested some routers could not handle the larger window scale, but this would not explain the separation between HTTP and non-HTTP traffic.

Appendix E

Pre-research

E.1 How does QueSO work?

Figure E.1 shows how a TCP connection is initiated. In case the server is not able to accept the connection, for example because there is no application that is listening on the specified port, it will send back an RST segment instead of an SYN+ACK. [RK04]

From the initial SYN+ACK response, QueSO can already determine some properties.

TCP Initial Windowsize The total number of bytes that are allowed to be transmitted when they have not been acknowledged is called the window size. This size is limited for the purpose of flow control. [RK04] The initial

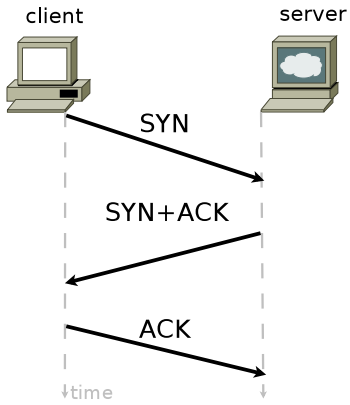


Figure E.1: TCP's three-way handshake

value for this setting is highly dependent on the OS. Some systems just use a fixed value for initial segments, where others use a multiple of the maximum segment size (MSS) or the maximum transfer unit (MTU). The MSS is the largest amount of data, which the system can handle in a single segment. The MTU is the largest amount of data which fits in a single packet. Normally the MTU is equal to the MSS plus the IP header size. Sometimes the value is arbitrary. Systems with limited memory and smaller buffers are likely to work with smaller window sizes. [Zal06a]

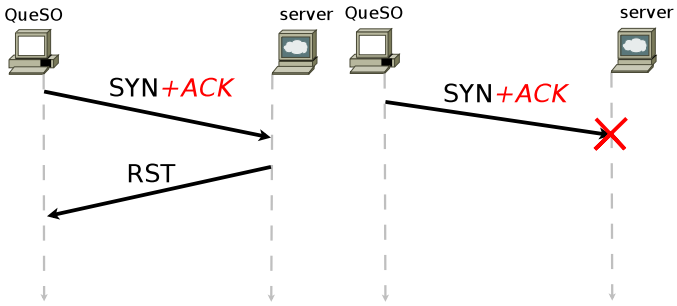
All the above is all clearly defined in the TCP RFC [Pos81c]. A packet is also sent by QueSO to verify whether there is indeed an application listening on the specified address and port number. Once QueSO has verified that the remote system is working, it starts sending its ambiguous packets.

E.1.1 SYN + ACK test

This is the second packet QueSO sends out. QueSO sends a SYN packet which also has the ACK flag set. This looks like a reply to a connection initiation request but is not covered by the TCP RFC when sent initially. A logical response could be responding like shown in Figure E.2(a) or E.2(b). It turns out the responses to this packet already show quite some differences in TCP stacks. While many systems send a RST back, such as Linux (1.x, 2.0), with no other values set, others leave a window size in the reply or even just accept the connection, completely ignoring the ACK flag. Some stacks used in different printservers and Novell's TCP/IP stack for DOS respond like shown in Figure E.2(c). Another observed behaviour is shown in Figure E.2(d).

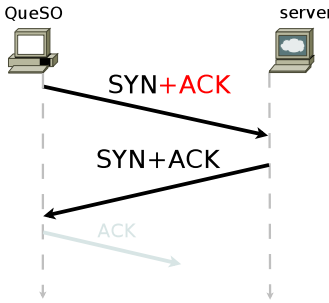
E.1.2 FIN test

A technique also used by QueSO's predecessors is the FIN probe. An initial packet is sent with just the FIN flag set. Although the RFC defines the correct response to this (the packet should be ignored), there happen to be many broken implementations which do respond. These are mostly again simple systems, but also MS Windows 9x and NT return a packet in this case. In many cases this packet contains an ACK bit, which seems to suggest that these stacks do not keep track of the connection states properly and

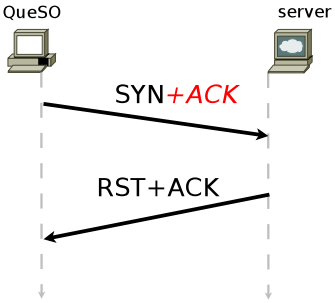


(a) Linux sends a RST back

(b) Many firewalls ignore (drop) QueSO's second packet.



(c) This stack does not check the ACK flag, another ACK would finish the three-way handshake.



(d) Stacks sending a RST back with an ACK likely reuse memory and forget to reset the ACK flag

Figure E.2: QueSO's second packet results in varying responses

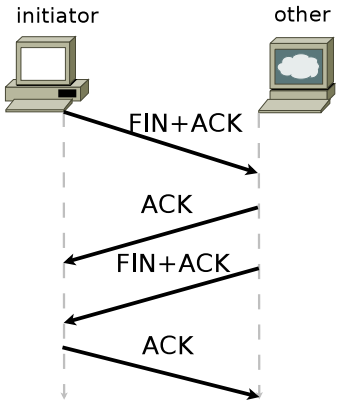


Figure E.3: TCP connection tear-down

they do not check the ACK of the incoming packet. A FIN packet without the ACK bit set is never a legal packet. [Tze08]

E.1.3 FIN+ACK test

Once a client and a server have established a connection like described earlier, one of them may want to close the connection. This process is shown in Figure E.3. In case either of the last packets is missing, the FIN segment will be sent and be again acknowledged, with the hope neither will not get lost again. To be able to resend this acknowledgement, the connection state is generally kept for 30, 60 or 120 seconds by the final acknowledging party after its acknowledgement was sent. [RK04]

Sending this as an initial packet, results in an RST reply from many stacks which reply nothing for the previous packet. It is not illogical to reply with an RST here, because otherwise the above described acknowledgement timer on the other side may be unnecessarily waiting in case the other host really believes some connection existed. Some systems however, mainly firewalled ones, also do not respond to this packet.

E.1.4 SYN+FIN test

This is also a packet that is always illegal [Tze08]. Since SYN is the flag for connection initialisation and FIN is the flag to close a connection, they should never appear in the same packet. It appears that many stacks, including MS Windows, some Linux versions and even Solaris, ignore the FIN

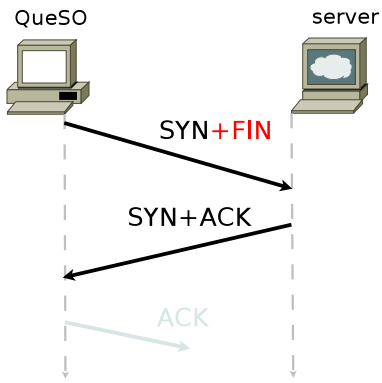


Figure E.4: Many systems ignore the presence of the FIN flag

flag and continue with the normal connection initiation like shown in Figure E.4. This likely has a similar cause the behaviour in Figure E.2(c) has, the value of the bit is not checked. There are some systems which stay silent when they receive a SYN+FIN packet.

E.1.5 PSH test

The PSH flag, used for the push function of the TCP, is normally set in ongoing connections when data should immediately be passed on to the receiving application once it arrives at the other side. A single PSH could never initiate a connection and most systems ignore such packets. Most others respond with a RST packet.

E.1.6 SYN+... test

In this test QueSO sends a SYN packet including some flags which are currently unused. Most hosts seem to accept such packets as a normal connection initiation packet. According to the RFC they should only echo the capabilities they support. A few systems from HP seem to trip over these extra bits, causing them to drop the packet.

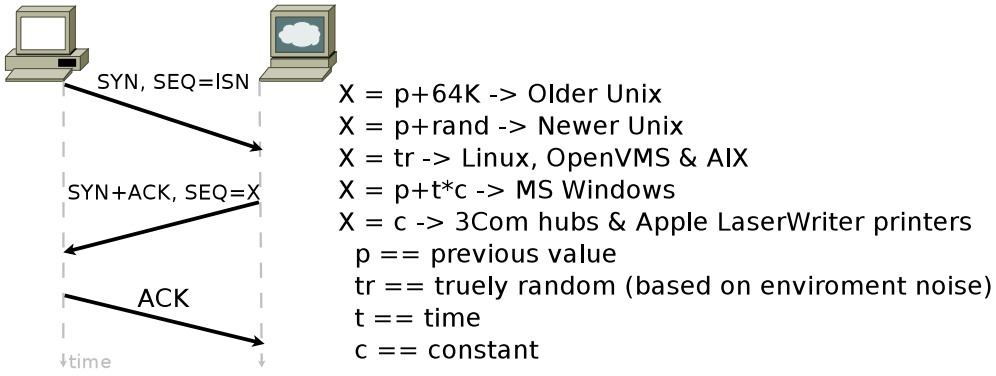


Figure E.5: TCP's three-way handshake: noting the ISNs

E.2 How does Nmap work?

E.2.1 TCP test 1

The first TCP test done by Nmap sends out six different TCP SYN probe packets. They differ based on the window size, window scale, timestamp flag and value, the SACK permitted flag and the options field. The acknowledgement and sequence numbers are random, but saved for use during the analysis. The following tests are based on the responses to these six packets.

TCP Initial Sequence Number (ISN) Sampling This sampling is done based on the sequence numbers in the replies of several of the probe packets. When setting up a TCP connection, an initial sequence number needs to be chosen. TCP uses sequence numbers for duplicate detection. The initial sequence number for a new connection needs to be determined in such a way that no packet from previous connections can be mistaken as a duplicate packet from a new connection. There are several ways to determine an appropriate initial sequence number and it turns out that this also allows for quite some differentiation. This is shown in Figure E.5. The possibility of guessing an ISN has security implications, so modern systems are expected to generate this truly randomly.

Nmap has several tests to find regularities in TCP ISNs. It checks for the greatest common divisor of the increment of the received values. The average increment of the ISN is checked against the time. The variability of the increments is also noted as the "sequence predictability index".

IP ID sequence generation Other sequence related tests are based on the IP identification (ID) field. "This field is used to distinguish fragments of one datagram from those of another." [Pos81b] It should thus be unique for a certain destination and protocol at a time. Some systems always leave this at zero or always use the same IP ID. Sometimes it is just incremented for every new packet. In other cases there are different kinds of randomness, such as smaller random increments versus larger random increments, which reveals details about the host. One interesting case is MS Windows where the byte order is based on the host architecture. Although this is not a violation, it does reveal extra details about the system.

Options All the options in the returned packets are recorded. Since the options a certain implementation does or does not support is one of the biggest differentiations between systems, a lot of information can be gained based on this. Another interesting thing is that the order in which the options are specified can be completely arbitrary. This means that even two systems which support similar options could still be differentiated because they advertise support for them in a different order. [Zal06a]

Window sizes Similar to QueSO, Nmap also keeps track of the window sizes. Since the sizes may be different for responses to packets which were initiated with different sizes, Nmap keeps track of the window size specified in all the six packets.

Based on just the first probe packet, also the following properties are used for differentiation.

Do not fragment flag (DF) The standard IPv4 header enables packets to be fragmented by routers in case a link has a smaller maximum transfer unit (MTU) than the size of the packet. By setting the do not fragment flag, an end-system can prevent a packet from being fragmented by a router. When a link's MTU is too small, an ICMP error will be returned instead. Modern OSs use this to discover the path MTU (PMTU). [Zal06a]

IP initial time-to-live guess When Nmap does not receive a response to the UDP probe it will need to guess the TTL. This is often possible because

there are only a few common TTL values, which have a bigger difference than the hop count will generally be between two hops.

Window scaling (WSCALE) RFC 1323 [JBB92] specifies several additions to TCP which allow TCP to work efficiently with high-speed networks. It mainly allows the use of larger window sizes, which used to be limited by the maximum value of the window field. This feature is used and advertised by modern OSs, that have incorporated the features of this RFC. Examples in this case are MS Windows which supports this feature since MS Windows 2000 and Linux where version 2.6.9 is the first release to have the feature turned on by default.

TCP Sequence number The sequence number in the reply is sometimes based on the acknowledgement number which was in the initial packet. This test verifies this condition.

TCP Acknowledgement number This is similar to the previous test, except that now the number is compared with the sequence number in the initial packet.

TCP Flags Also the TCP flags in the response are noted for the reply to the first probe packet.

TCP RST data checksum When the initial probe packet is replied to with a packet that has the RST flag set, this test is triggered. Some OSs do return data in such packets while others do not. This data often consists of standard error messages in plain ASCII. Nmap records this by CRC32 checksumming the data and reporting this checksum. Noteable systems that return such data are HP-UX and the original Mac OS.

Non zero field after the TCP header length This field is reserved and should normally be zero within packets. The original TCP RFC specifically states this. Only when explicit congestion notification (ECN) is done, this value may be different, but this initial packet does not set that yet. [Pos81c, RFB01]

Non-zero URG pointer Normally the URG pointer can point to a specific section of the payload which contains urgent data. This is a 16 bit field which points to the last byte of the "urgent" data. Since SYN packets do not have any payload and the URG flag is not set in them, this pointer is ignored. Normally it is set to zero, but some systems, notably MS Windows 2000/XP, do not properly do that and simply leave garbage. On newer service pack versions this turned out to be a memory leak. [Zal06a]

ICMP test

Nmap also does some tests using ICMP to differentiate systems. It sends two ICMP echo request packets with only some small distinctions and analyses the replies with the following tests. OS detection using ICMP was pioneered by Xprobe, which will be described in Section 2.3.5.

Do not fragment (DF) The DF feature in the ICMP is similar to the one in the TCP. Nmap sets the DF flag in the first probe, while the second probe does not have it set. This gives four possible combinations for the flag in the reply: neither have it set, it is echoed, both have it set or it is toggled. According to the Xprobe authors it should always be zero.

IP ID sequence generation with ICMP The final IP ID related tests checks whether the IP IDs from ICMP and TCP are based on a single sequence generator or whether they are separate.

Based on the answer of just the first ICMP probe, the following tests are also done.

IP initial TTL guess This test is the same as the one performed on the first TCP probe when the TTL cannot be discovered with the UDP probe.

ICMP Response code Nmap sends the first packet with an echo request code of nine (which should be zero) and the second with an echo request code of zero. The code value of the ICMP echo reply should always be zero. Some systems send other values however, especially when the echo request of the initiating packet also has a non-zero value. This allows some more differentiation.

E.2.2 TCP Explicit Congestion Notification (ECN) probe

ECN is a mechanism to let routers signal congestion without dropping packets. It extends the IP, but also requires support from the transport protocol. Basically this probe tests for ECN support and some of the earlier described tests are also done on the response to this packet.

E.2.3 TCP test 2 to 7

Now Nmap sends out 6 different TCP probes. Each with different characteristics. The details of these packets can be found in the official book about Nmap [Lyo09b]. All of the above described TCP tests will be performed on the responses.

E.2.4 UDP test

The final protocol used by Nmap for OS detection is UDP. Nmap sends a single UDP probe to a closed port. The data consists of 'C' repeated 300 times and the IP ID has a fixed value of 0x1042. The expected reply is an ICMP port unreachable message, which is subjected to the following tests.

Do not fragment flag (DF) This test corresponds to the one for the TCP probes.

IP initial time-to-live The TTL field is decreased by one each time a packet passes through a router. This is to prevent packets to get routed infinitely (e.g. in a routing loop). The initial value of this field is often left at its default setting, which is different depending on the OS. When Nmap sends the UDP probe, the response will include the (first part of) original UDP probe packet with the decremented TTL field. Subtracting this from the original TTL value that was sent out with the packet will give the hop distance. The response to the probe also has its own TTL value which added to the hop distance will result in the original TTL value used by the target system.

IP total length When an ICMP destination port unreachable message is returned, the amount of data included can be arbitrary (must be higher

than 8 bytes). Since different systems include different amounts, this can be used for differentiation. More details are in section E.3.1.

Unused port unreachable field nonzero An ICMP port unreachable packet is eight bytes long. The last four bytes of an ICMP port unreachable message are unused and should be set to zero according to the RFC [Pos81a]. Some systems do not correctly zero this however, this value is recorded.

Returned probe IP total length value Many systems corrupt this value. Details are described in Section E.3.2.

Returned probe IP ID value Normally the exact value is returned here, but some systems, notably some printers from HP and Xerox, somehow flip the bytes.

Integrity of returned probe IP checksum value The checksum is not always calculated properly in the returned header. More in Section E.3.2.

Integrity of returned probe UDP checksum and data The UDP header checksum should not be changed in the response. Apparently some systems corrupt this. The integrity of the returned data is also checked against corruption.

E.3 How does Xprobe work?

The details which are used by Xprobe 2 to differentiate OSs will be described in the following subsections.

E.3.1 ICMP Error Message Quoting Size

When an error occurs in the network, for example when establishing a TCP connection with a missing host, at some point an ICMP message will return indicating the destination is not reachable. This message was generated by a router that was unable to find a path to the specified host.

An ICMP error message has to contain at least the first eight data bytes of the datagram that caused the error. It is allowed however to send more

than eight bytes. While most systems will return exactly eight bytes, several will quote more including different versions of Linux, Solaris and the original Mac OS.

E.3.2 ICMP Error Message Echo Integrity

When sending back the IP header of an offending packet, it should not be changed from the one originally received. Compared to the header which was originally sent, the only changes should thus be in the TTL field and the header checksum. There are cases however, where more alterations are made by the replying host, which enables differentiation.

- The IP Total Length Field is changed by some systems. Some add and others subtract 20 bytes. Correct systems echo the original value.
- The IP ID field also tends to be incorrectly echoed. Arkin describes bit order changes and a bug in early Linux 2.4 kernels where this field was set to zero.
- Fragmentation flags and the fragmentation offset field are also susceptible to bit order changes.
- The IP header checksum is sometimes calculated incorrectly or set to zero.
- The Type of Service (TOS) [Alm92] byte affects several types of ICMP error messages. In source quench messages it has to echo the precedence field exactly. In all other ICMP error messages this value should be 6 or 7. Some systems tend to threat this field differently however. The TOS field specifies how packets should be routed, e.g. reliability versus speed.
- The DF bit should always be set to 0, however sometimes it is quoted from the offending packet.
- When sending an ICMP Echo request with strange value (it should be eight), MS Windows will always send back an ICMP code field value of zero. others will echo the original code field value back.

In contrast to the earlier tools described, Xprobe does not do separate tests separately, combining the information to form a single signature. Instead Xprobe contains a tree structure which it descents through, taking decisions based on the results of its probing. In addition to this, it can combine several probes in a single datagram, because they do not influence each other's result. The first datagram Xprobe sends out is an UDP datagram to a (hopefully) closed UDP port in order to receive an ICMP Port Unreachable Error message. From this message, the many details are obtained by checking the echo integrity as described.

E.4 How does p0f work?

E.4.1 Metrics used by Incoming connection fingerprinting (SYN)

The fingerprint files that come with p0f describe all the metrics that are used. This information comes mostly from there. [Zal06a]

Window size Similar to QueSO and Nmap, p0f tracks the window size used in the packets.

Overall packet size This is influenced by all the IP and TCP options used and also some buggy stacks can produce some strange sizes.

Initial time-to-live (TTL) Altough p0f will never see the original TTL, from the TTL of the received packet, the initial TTL can often be determined, because the differences between common initial TTLs are much larger than the number of hops between two hosts. This is similar to how Nmap finds the TTL when it needs to guess it.

Do not fragment flag (DF) This can be observed by p0f in a similar way Nmap does it.

Maximum segment size (MSS) This setting is usually dependant on the links used for the connection, which specify a maximum packet size.

Window scaling (WSCALE) This can be similarly observed like the initial time-to-live value.

Timestamp To be able to calculate a correct round-trip-time (RTT) on certain link types (often with long delays, but high data rates) it is necessary to include a timestamp in the TCP header. This timestamp is sometimes based on the system's up-time, other systems set it to zero in the initial SYN packet.

Selective ACK permitted Normally packets can only be acknowledged cumulatively. This can lead to big inefficiencies when only a single packet was lost and all the others cannot be acknowledged cumulatively. RFC 2018 specifies selective acknowledgement which allows the acknowledgement of out-of-order segments selectively. [RK04] Some systems do and others do not implement this functionality.

NOP (No-Operation) option This option code may be used between options, for example, to align the beginning of a subsequent option on a word boundary. [Pos81c] Its occurrence and number are thus arbitrary and heavily dependant on the TCP implementation.

Other options Sometimes options pass by that are unrecognised. p0f just looks whether those options occur or not for OS detection. The only two systems that appear to use unknown options are RiscOS, Amiga and a special firewall system built by Nokia. The fingerprint file suggest that some of these are likely the result of a buggy TCP stack.

EOL (End of Option List) option This option code indicates the end of the option list in case it does not coincide with the end of the TCP header. [Pos81c] It is not used often, usually systems just pad with NOP option codes instead.

The sequence of the TCP options As stated before, the options may appear in a packet in arbitrary order according to the RFC. Because of this the actual ordering in a packet is often heavily dependant on the implementation.

Quirks There are buggy TCP/IP stacks which bring all kinds of strange quirks in their packets. There was even a bug in some versions of MS Windows which resulted in memory being leaked. p0f also treats other irregularities as quirks. These are the quirks p0f is able to detect:

Data past the headers SYN and SYN+ACK packets should not have any payload, but sometimes this does occur.

Options past EOL Sometimes systems have options after the end of option list option. p0f only detects this fact, but does not do anything with the data.

Zero IP ID This test is similar to Nmap's IP ID test. p0f can only check whether an IP ID is zero or not, since there is only a single packet being checked.

IP options Usually IP options are not set, but modern systems are starting to use them more and more. p0f does not examine any options, it only detects their presence.

Non-zero URG pointer This is similarly observed by Nmap. Again the URG flag should never be set and the data in this value will always be ignored by an underlying system, because only SYN packets are being observed. Data left behind there is useful for OS detection. p0f does not examine the actual value only whether it is zero or non-zero.

Unused field value This field is not used (yet) and should always be zero. Some systems however do not clear it. p0f checks for a non-zero value.

ACK number is not zero The ACK number in SYN packets with the ACK flag unset is usually zero and disregarded. Some systems however send junk like with the URG pointer value.

Non-zero second timestamp TCP timestamps are used to compute the round-trip time. They are specified in RFC 1323 [JBB92]. The initial SYN packet should have a zeroed second timestamp. p0f verifies this.

Unusual flags Flags are settings that are specified in a single bit in a segment header. Extra flags that are not really important but can be set in a segment, such as URG and PUSH, are also noted by p0f. ECN flags are currently ignored.

E.4.2 Metrics used by Outgoing connection fingerprinting (SYN+ACK)

This mode is not well supported in p0f. It is largely based on the SYN fingerprinting and also most of its metrics are used. The differences are noted here

ACK number non-zero Since this packet is actually supposed to have a proper ACK number set, having a zero ACK number is very uncommon. In p0f's signature file this is actually indicated by having this quirk set for it.

Non-zero second timestamp Similar to the above quirk, this is supposed to be set on SYN+ACK packets.

E.4.3 Metrics used by Connection refusing fingerprinting (RST+)

This fingerprinting method is also not well supported. There are many interesting variations detectable though. According to p0f's fingerprint file for this type of fingerprinting, this has two reasons. One, because strange flags do not have many consequences, the connection is not established anyhow. Two, RFC 793 is difficult to comprehend regarding these types of responses. The differences p0f uses will be presented here.

Connection refused packet A proper connection refused packet should only be sent, when a connection is refused. There are rare cases however where it is sent in response to an unexpected ACK packet. A normal connection refused packet has the RST and ACK flags, its SEQ value should be zero and its ACK number non-zero.

Error: ACK number is zero Sometimes connection refused packets are sent out with an ACK number set to zero. This is an incorrect response. p0f detects this.

Error: non-zero SEQ value The ACK number is non-zero or zero. This tends to be generated by Cisco routers that accidentally set an ACK flag.

Connection dropped The RST flag is set with a non-zero sequence number. The acknowledgement number should be zeroed, but it is not against the RFC if it is not. Again MS Windows leaks memory there in some cases.

Error: the RST flag and SEQ value are zero The ACK number can be zero or non-zero. This is an obvious error and will not result in the desired effect, since the other host cannot correlate this packet to anything.

E.4.4 Metrics used by Ongoing connection fingerprinting (stray ACK)

This mode in p0f is extremely limited. There are only six fingerprints in p0f's fingerprint file and the functionality has not been tested much yet.

Literature

[Ark02] Ofir Arkin. A remote active OS fingerprinting tool using ICMP. 27(2), April 2002.

[Auf08] Patrice Auffret. SinFP, unification of active and passive operating system fingerprinting. *Journal in Computer Virology*, August 2008.

[Bec01] Rob Beck. Passive-aggressive resistance: OS fingerprint evasion. *Linux Journal*, 2001(89):1, 2001.

[Bev04] Robert Beverly. A Robust Classifier for Passive TCP/IP Fingerprinting. In *Proceedings of the 5th Passive and Active Measurement (PAM) Workshop*, pages 158–167, April 2004.

[BGV92] B.E. Boser, I.M. Guyon, and V.N. Vapnik. A training algorithm for optimal margin classifiers. In *Proc. 5th Annual ACM Workshop on Computational Learning Theory*, pages 144–152. ACM Press, 1992.

[Coh95] W.W. Cohen. Fast effective rule induction. In *Proc. 12th International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

[GE09] François Gagnon and Babak Esfandiari. Using answer set programming to enhance operating system discovery. In *Logic Programming and Nonmonotonic Reasoning*, pages 579–584, 2009.

[GEB07] F. Gagnon, B. Esfandiari, and L. Bertossi. A hybrid approach to operating system discovery using answer set programming. In *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE*

International Symposium on, pages 391–400, 21 2007-Yearly 25 2007.

[GT07a] Lloyd G. Greenwald and Tavaris J. Thomas. Toward undetected operating system fingerprinting. In *WOOT '07: Proceedings of the first USENIX workshop on Offensive Technologies*, pages 1–10, Berkeley, CA, USA, 2007. USENIX Association.

[GT07b] Lloyd G. Greenwald and Tavaris J. Thomas. Understanding and preventing network device fingerprinting. *Bell Lab. Tech. J.*, 12(3):149–166, 2007.

[HDW94] Geoffrey Holmes, Andrew Donkin, and Ian H. Witten. Weka: a machine learning workbench. 1994.

[KR04] Christopher Kruegel and William Robertson. Alert verification - determining the success of intrusion attempts. In *Proc. First Workshop the Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA 2004*, pages 1–14, 2004.

[KS05] S. Kalia and M. Singh. Masking approach to secure systems from operating system fingerprinting. In *TENCON 2005 2005 IEEE Region 10*, pages 1–6, Nov. 2005.

[Lee99] W. Lee. *A data mining framework for constructing features and models for intrusion detection systems.* PhD thesis, Columbia University, New York, NY, USA, 1999.

[LFM+02] W. Lee, W. Fan, M. Miller, S.J. Stolfo, and E. Zadok. Toward cost-sensitive modeling for intrusion detection and response. *Journal of Computer Security*, 10(1-2):5–22, 2002.

[Lif02] Vladimir Lifschitz. Answer set programming and plan generation. *Artif. Intell.*, 138(1-2):39–54, 2002.

[MLH03] D. Meyer, F. Leisch, and K. Hornik. The support vector machine under test. *Neurocomputing*, 55(1-2):169–186, 2003.

[Pie04] Tadeusz Pietraszek. Using adaptive alert classification to reduce false positives in intrusion detection. In *In Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID*, pages 102–124. Springer, 2004.

[Qui93] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[SB08] Carlos Sarraute and Javier Burroni. Using neural networks to improve classical operating system fingerprinting techniques. *Electronic Journal of SADIO*, 8(1):35–47, January 2008.

[SMJ00] Matthew Smart, G. Robert Malan, and Farnam Jahanian. Defeating tcp/ip stack fingerprinting. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 17–17, Berkeley, CA, USA, 2000. USENIX Association.

[VL63] V.N. Vapnik and A. Lerner. Pattern recognition using generalized portrait method. *Automation and Remote Control*, 24, 1963.

[WSMJ04] D. Watson, M. Smart, G.R. Malan, and F. Jahanian. Protocol scrubbing: network security through transparent flow modification. *Networking, IEEE/ACM Transactions on*, 12(2):261–273, April 2004.

General References

[Ark02] Ofir Arkin. A remote active OS fingerprinting tool using ICMP. 27(2), April 2002.

[Auf06] Patrice Auffret. *SinFP Changes*, 2006.

[Auf08] Patrice Auffret. SinFP overview, 2008.

[Kol05] Eric Kollmann. Chatter on the wire: A look at excessive network traffic and what it can mean to network security. August 2005.

[Lyo98] Gordon Lyon. Remote OS detection via TCP/IP Stack Finger-Printing. *Phrack Magazine*, 8(54), December 1998.

[Lyo09a] Gordon Lyon. *nmap - CHANGELOG*, 2009.

[Lyo09b] Gordon Lyon. *Nmap Network Scanning The Official Nmap Project Guide to Network Discovery and Security Scanning*, chapter 8. Insecure.Com LLC., 2009.

[Mur98] Jordi Murg. *Introduction to QueSO*. Apstols, September 1998.

[Mur99] Jordi Murg. 'QueSO project page'. WWW, October 1999.

[Qui09] R. Quinlan. *C5.0: An Informal Tutorial*. RuleQuest Research, November 2009.

[Tal03] Greg Taleck. *Ambiguity Resolution via Passive OS Fingerprinting*. 2003.

[TCP10] TCPDUMP/LIBPCAP team. TCPDUMP/LIBPCAP public repository (website), 2010.

[Tze08] Shane Tzen. firewalling with netfilter / iptables - IP overview, 2008.

[VCH02] Franck Veysset, Olivier Courtay, and Olivier Heen. New tool and technique for remote operating system fingerprinting. April 2002.

[w3s10] w3schools. OS Platform Statistics. WWW, February 2010.

[Zal06a] Michal Zalewski. *p0f - SYN fingerprints*, 2006.

[Zal06b] Michal Zalewski. *p0f 2 - README*, 2006.

[Zal06c] Michal Zalewski. the new p0f (website), 2006.

Books

[BH02] Kurt Binder and Dieter W. Heermann. *Monte Carlo Simulation in Statistical Physics: An Introduction (Springer Series in Solid-state Sciences)*. Springer, 4th, ed. edition, July 2002.

[RK04] Keith W. Ross and James F. Kurose. *Computer Networking: A Top-Down Approach Featuring the Internet*. Pearson Addison-Wesley, third edition edition, 2004.

RFCs

[Alm92] P. Almquist. Type of Service in the Internet Protocol Suite. RFC 1349, July 1992. Obsoleted by RFC 2474.

[ASNN07] J. Abley, P. Savola, and G. Neville-Neil. Deprecation of Type 0 Routing Headers in IPv6. RFC 5095, December 2007.

[Bra89] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122, October 1989. Updated by RFCs 1349, 4379.

[CFSD90] J.D. Case, M. Fedor, M.L. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157, May 1990.

[DH98] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, December 1998. Updated by RFC 5095.

[EF94] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631, May 1994. Obsoleted by RFC 3022.

[JBB92] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, May 1992.

[KA98] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, November 1998. Obsoleted by RFC 4301, updated by RFC 3168.

[Pos81a] J. Postel. Internet Control Message Protocol. RFC 792, September 1981. Updated by RFCs 950, 4884.

[Pos81b] J. Postel. Internet Protocol. RFC 791, September 1981. Updated by RFC 1349.

[Pos81c] J. Postel. Transmission Control Protocol. RFC 793, September 1981. Updated by RFCs 1122, 3168.

[RFB01] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, September 2001.

[Whi75] J.E. White. High-level framework for network-based resource sharing. RFC 707, December 1975.