Remote Operating System Classification over IPv6

David Fifield*
University of California, Berkeley
fifield@eecs.berkeley.edu

Alexandru Geana
Technische Universiteit Eindhoven
Fox-IT, Delft
alex@alegen.net

Luis MartinGarcia
ETSIT, Polytechnic University of Madrid
luis@luismg.com

Mathias Morbitzer
Fox-IT, Delft
m.morbitzer@runbox.com

J. D. Tygar
University of California, Berkeley
doug.tygar@gmail.com

ABSTRACT

Differences in the implementation of common networking protocols make it possible to identify the operating system of a remote host by the characteristics of its TCP and IP packets, even in the absence of application-layer information. This technique, "OS fingerprinting," is relevant to network security because of its relationship to network inventory, vulnerability scanning, and tailoring of exploits. Various techniques of fingerprinting over IPv4 have been in use for over a decade; however IPv6 has had comparatively scant attention in both research and in practical tools. In this paper we describe an IPv6-based OS fingerprinting engine that is based on a linear classifier. It introduces innovative classification features and network probes that take advantage of the specifics of IPv6, while also making use of existing proven techniques. The engine is deployed in Nmap, a widely used network security scanner. This engine provides good performance at a fraction of the maintenance costs of classical signature-based systems. We describe our work in progress to enhance the deployed system: new network probes that help to further distinguish operating systems, and imputation of incomplete feature vectors.

Categories and Subject Descriptors

C.2 [**Computer-Communication Networks**]: Network Protocols

Keywords

OS fingerprinting

*Authors are listed alphabetically.

AISec'15, October 16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3826-4/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2808769.2808777 .

1. INTRODUCTION

The ability to remotely identify the operating system of a network host has several implications for network security:

- Network inventory: identifying hosts that should and should not be on a network.
- Vulnerability assessment: though OS fingerprinting is fairly coarse-grained with respect to version information, it can often identify out-of-date hosts that are more promising targets of attack.
- Exploit tailoring: a penetration tester or network attacker can use knowledge of the operating system to select proper shellcode, for example.

The possibility of OS fingerprinting—the fact that idiosyncrasies in the implementation of network protocols such as TCP and IP are remotely detectable and hard to disguise—has long been known, and there exist many software tools that take advantage of it. However, both research and practice have mainly focused on IPv4. In this paper we apply machine learning techniques to the problem of OS fingerprinting over IPv6, with a thorough examination of the fingerprinting engine we built and which is deployed in the Nmap security scanner. Figure 1 is sample output, which in this case shows that the remote host is running Linux, and one of a small range of kernel versions.

The engine's design is guided by many years' experience in dealing with IPv4-based detection. It is fundamentally based on a logistic regression model trained on a few hundred known OS "fingerprints," which are the packets received in response to up to 18 specially crafted network probes. OS fingerprinting has traditionally relied on a nearest-neighbor match against a database of known fingerprints. The use of machine learning is aimed at enabling the engine to better identify fingerprints it has not seen exactly before; and also to cope with the many ways a packet may be corrupted in transit (unfortunately many of the protocol fields useful for fingerprinting are often modified by intermediate devices). The great diversity of operating systems and types of network interference have in the past required a proportionally large database of known fingerprints. The cost of maintaining such a database is non-trivial and we hope to make it more manageable.

The training set of known operating system fingerprints was initially seeded by us through manual scans of common

```
# nmap -6 -O scanme.nmap.org

Starting Nmap 6.49SVN ( http://nmap.org )
Nmap scan report for scanme.nmap.org (2600:3c01::f03c:91ff:fe18:bb2f)
Host is up (0.069s latency).
Other addresses for scanme.nmap.org (not scanned): 45.33.32.156
Not shown: 998 closed ports
PORT   STATE SERVICE
22/tcp open  ssh
80/tcp open  http
Device type: general purpose
Running: Linux 3.X
OS CPE: cpe:/o:linux:linux_kernel:3
OS details: Linux 3.13 - 3.19
Network Distance: 15 hops

OS detection performed. Please report any incorrect results\
 at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 8.10 seconds
```

Figure 1: Nmap OS detection run against a sample host. In about 8 seconds, the program has scanned 1,000 ports, selected an open and a closed port, and then sent several probes designed to uncover idiosyncrasies of the host's OS. The lines starting with 'Device type:', 'Running:', 'OS CPE:', and 'OS details:' are the output of the OS classification engine. The -6 option switches to IPv6 mode (the default is IPv4) and -O activates OS detection. The "submit" web form is the primary vehicle for the growth of the OS database.

systems, but it is mainly driven by community submissions. When Nmap scans a host and fails to find an OS match of sufficient quality, it displays a textual fingerprint and encourages the user to submit it to a web form if they happen to know the OS. The submissions are individually vetted by a human before being added to the training set. Section 5 describes how the program makes the call of whether to print a potentially useful but low-confidence OS match, or to display a fingerprint and ask for its submission.

Although application protocols may provide a wealth of information, and in many cases an application can be mapped back to the OS hosting it, such techniques are beyond the scope of this paper. We are concerned only with the network and transport layers.

1.1 What makes OS fingerprinting possible

While the interoperability of hosts on the Internet relies on network stacks supporting the same standard protocols, OS vendors have considerable leeway in the implementation of those standards. Some protocol fields are left reserved or unspecified; some have optional features that may or may not be present in any particular implementation; and sometimes specifications are incomplete and implementers must invent their own behavior for some corner cases. Even where specifications are completely unambiguous, an implementation can simply contain bugs that make it deviate from standard behavior. The network-visible features of network traffic are typically deeply tied with the implementation of the operating system and are not easy to disguise. Some differences are not merely a matter of taste or configuration but can have performance and security implications, so they cannot simply be disabled without consequences. A good example of this are TCP timestamps [17], which serve both to enhance round-trip time estimation, and to prevent sequence numbers from wrapping.

The freedom of protocol implementers to make choices that do not affect compliance to the standards often makes the network traffic generated by their implementations to be distinguishable from others. There are many good examples of this.

Versions of Windows up to Windows 98 used a simple incrementing counter for the IP ID field—but sent the field with little-endian byte order. Versions of Solaris use different initial TTLs for TCP and ICMP packets. Other examples of features that vary across operating systems and are useful for fingerprinting are: TCP options (their values and order); TCP initial window sizes; the initial IPv4 TTL or IPv6 hop limit; handling of IPv6 extension headers; and the frequency of the TCP timestamp counter. In addition, OSes vary in their responses to unusual or malformed packets; even the information of whether a response was sent can be useful for fingerprinting.

The basics of OS fingerprinting are fairly robust. Almost always, given an open and a closed TCP port, it is possible to remotely identify at least the operating system family (e.g. "Linux" or "Windows") using nothing more than simple procedural rules and high-level features. (The earliest fingerprinting tools worked that way, building classification logic directly in code [18]. Contemporary systems use fixed classification code and a variable external database of fingerprints.) We know through experience that it is possible, though it requires more care, to make finer distinctions in versioning, in many cases separating different revisions of the same OS. For example, Linux 2.4 is readily distinguishable from Linux 2.6—but a good classifier should be able to do even better than that.

Some examples from the history of the 2.6 series of Linux kernels will illustrate how OS fingerprints can change over time and make different revisions distinguishable. Linux 2.6.8 in 2004 began sending a nonzero TCP window scale option by default [9]; it is thus easy to distinguish from earlier versions. Linux 2.6.22 in 2007 increased by a factor of 15 the frequency of the counter used to set TCP initial sequence numbers. Linux 2.6.38 in 2010 increased the size of the initial congestion window [10]. In 2008, a bugfix and code refactoring in Linux 2.6.27 had the side effect of changing the order of TCP options. The change had to be hurriedly reverted after it was found incompatible with some consumer networking equipment that wrongly required a certain option ordering [13]. Linux TCP segments bearing this peculiar option ordering may as well be stamped "2.6.27": they are permanently and narrowly isolated to a very small number of released kernels and a narrow period of time. Changes like these create clear "break points" between which ranges of OS versions should be distinguishable.

A primary challenge of OS fingerprinting is "middlebox interference," changes made to packets en route by various network devices such as routers, firewalls, and load balancers. Interference means that the packets you receive often do not perfectly reflect the packets a remote host sent. Sometimes the overwritten fields are those that are good for fingerprinting. In addition, some protocol fields are not a function only of the remote operating system but also of the network link. For example, the TCP maximum segment size (MSS) is often characteristic of an OS but is also constrained by the network. Some fields vary naturally, like Linux's window

scale option that partly depends on how much RAM is installed [29]. The trick is to account for variations such as these, without completely ignoring the underlying OS features it may mask. Enumerating all possible forms of interference is difficult and has, for us in the past, led to a proliferation of specific fingerprints or overbroad fingerprints.

1.2 Differences in IPv6

Many IPv4 fingerprinting techniques carry over straightforwardly to IPv6, though IPv6 brings its own challenges and opportunities. Much of the power of OS fingerprinting comes from the TCP layer, the implementation of which is usually the same between IPv4 and IPv6. (Windows XP used different TCP implementations for IPv4 and IPv6, with different window sizes and options. However, this appears to be an exception to the general rule: almost all other OSes, including later versions of Windows, use the same TCP implementation in IPv4 and IPv6, as far as we have seen.)

The IPv6 header lacks some fields of the IPv4 header. There are no type-of-service, identification, or fragment offset fields. In exchange, IPv6 gains traffic class and flow label fields. The hop limit field takes the place of IPv4's time to live (TTL). The most important change in IPv6 is the introduction of extension headers [12] at the IP layer. Instead of the length-limited options block, IPv6 uses a chain of typed headers that terminates in the upper-layer protocol payload (e.g. TCP or UDP). Much like TCP options, extension headers and how they are processed tend to be specific to network stack implementation. Some of our OS probes send specific extension headers in order to uncover differences in header processing.

2. RELATED WORK

OS fingerprinting techniques can generally be divided into two categories: passive and active. Passive techniques rely only on naturally occurring traffic, such as the SYN and SYN/ACK in a TCP handshake. They do not directly stimulate responses from remote hosts and therefore do not make strong assumptions about what kind of packets may have provoked observed traffic. Passive detection has the advantage that it can use both live and recorded traffic. It gives up some discriminating power because it tends to see network stacks only in under normal conditions, and not in the exceptional conditions that may reveal more differences. Active detection, on the other hand, sends probes that are specially designed to elicit different responses from different operating systems. Active techniques have the potential to be more sensitive than passive techniques, but they are noisier, easier to detect and block.

The focus of this paper is the active IPv6 OS detection engine in Nmap. OS detection in Nmap has a long history, going back to 1998 and the debut of its "first-generation" engine for IPv4 [18]. This engine sent a battery of probes: 6 TCP sequencing probes to assess sequence and timestamp generation, 4 TCP probes to an open port, 3 TCP probes to a closed port, and 1 UDP probe to a closed port. Responses to the probes were examined for various features such as the TCP timestamp clock frequency, TCP flags and options, IP fields, and others, and matched against a database of known fingerprints. The database had a simple pattern language that allowed many values to match a particular feature field. The first-generation database contained 1,684 fingerprints when it was retired in 2007.

Nmap's OS detection engine was overhauled in 2006. The "second-generation" system uses roughly the same probes as the first generation [19]. The main difference is the introduction of manually tuned weights (called MatchPoints) on features that allow inexact matches and multiple matches. The MatchPoints assign a penalty for a mismatch in each feature. The more important a feature, the higher its MatchPoints. Certain features have lower MatchPoints in recognition of their susceptibility to middlebox interference. The best match is the one with the fewest penalties. The system has been enhanced in many ways since 2006, but its fundamentals remain the same today. Its principal drawback is the high cost of maintaining the fingerprint database. Because the database pattern language is fairly simple, it is necessary to pad the database with partly redundant fingerprints that represent the most common operating systems under a variety of network conditions: with different firewall configurations, over different physical links, and with varying interference by intermediate routers. Maintaining the database requires substantial human curation, which was one of the motivations to employ machine learning in the IPv6 engine. To date, in Nmap 6.49, the second-generation OS database contains 4,766 fingerprints. To give an idea of the amount of redundancy in the database, 789 (17%) of those fingerprints are varieties of Linux (not including Linux embedded in hardware devices).

The premier passive fingerprinting tool is p0f. It has been rewritten several times: v1 was released in 2000 [36]; v2 in 2003 [37]; and v3 in 2012 [38]. p0f uses the same fingerprints for IPv4 and IPv6; it ignores certain IPv4-specific fields (like the "don't fragment" field) when matching against IPv6 traffic. OpenBSD's packet filter PF incorporates passive OS fingerprinting based on p0f.

Beverly [6] built a classifier that used a subset of p0f's features: the TTL, TCP window size, SYN packet size, and the value of the "don't fragment" bit. The classifier makes the naive Bayes assumption that each feature is independent and outputs the OS match with maximum likelihood. He tried training the classifier using both the p0f database and by setting up a web server and correlating TCP/IP signatures with HTTP-layer OS identification from the server logs.

Xprobe2 [1, 35] is an active tool whose design prioritizes sending a small number of probes. Its default probes emphasize ICMP, though it has a module programming interface that allows for the addition of other tests with weights and reliability ratings. Xprobe2 employs non-strict matching against a database, assigning a score from 0 to 3 to each feature, and returning the match with the highest sum of feature scores. Some of its tests target the application layer.

Some systems skirt the boundary between passive and active by sending only a single SYN probe. Examples of these are RING [33], Snacktime [5], and Hershel [31]. These systems augment the usual response feature set with temporal features, exploiting differences in SYN/ACK retransmission. They wait for a relatively long time (up to 120 seconds) after sending their probe in order to collect a vector of time differences between SYN/ACK retransmissions.

SinFP [3], released in 2006, was the first public tool to do OS classification over IPv6. It does both passive and active classification. It has a database of IPv6 fingerprints, and the ability to fall back to an IPv4 fingerprint if none of the IPv6 fingerprints match. To automatically convert

an IPv4 fingerprint into one that is compatible with IPv6, SinFP applies the correspondences:

IPv4 identification → IPv6 flow label
IPv4 time to live → IPv6 hop limit
IPv4 don't fragment → IPv6 traffic class

The conversion is reasonable, because it is common for an OS to share a TCP implementation across IPv4 and IPv6, and SinFP applies heuristics to resolve inexact matches.

There have been experiments with bootstrapping a machine learning–based classifier using Nmap's large databases of IPv4 OS fingerprints. Sarraute and Burroni [30] built a hierarchy of neural networks trained on the first-generation database (at the time, it contained 1,684 samples). The first neural network separated "relevant" OS families (Windows, Linux, Solaris, OpenBSD, FreeBSD, and NetBSD) from all others. The next separated the "relevant" OSes into families, and further layers separated OS versions within a family. Their raw feature vectors had dimension 568. Medeiros et al. [23] converted the second-generation database into feature vectors (at the time, the database contained 430 samples). They used the vectors to build a contextual map of similar operating systems, with the goal of classifying previously unseen operating systems.

Greenwald and Thomas [16] measured the information gain of tests in the second-generation database, with the goal of achieving high accuracy while sending fewer probes. They achieved accuracy almost as high as Nmap's full set of 13 TCP probes, while sending only 3 or 4. They found that the TCP window size and options were overall the most information-carrying features. They ran their tests when the database contained 417 samples.

3. PROBE SELECTION

As a starting point for our research, we studied a wide variety of RFCs related to IPv6. Examples include:

RFC 2460 (the specification of the IPv6 protocol itself)
RFC 2463 (ICMP for IPv6)
RFC 2473 (Generic Packet Tunneling)
RFC 2675 (Jumbograms)
RFC 3122 (Inverse Discovery)
RFC 3775 (Mobility)
RFC 3971 (Secure Neighbor Discovery)
RFC 4620 (Node Information Queries)
RFC 4782 (Quick-Start)
RFC 4861 (Neighbor Discovery)
RFC 5570 (CALIPSO)

In the RFCs we looked for features, options or fields labeled as "optional" or "recommended," definitions we considered ambiguous, and behaviors described using the verb "should," which in the RFC terminology means that implementers may find valid reasons or particular circumstances to ignore the specified behavior.

From the standards documented in the RFCs, we created a set of 154 network probes. The criteria for the definition of those probes was to meet as many as possible of the following goals:

- Use packets that are likely to elicit responses from the target device (e.g. use ICMP request messages as opposed to replies, which will likely be silently discarded).
- Use packets that test for features or behaviors that are described ambiguously or left as optional or recommended in the RFC.
- Use packets that test for corner cases or non-general purpose protocols that vendors may have decided not to implement.
- Use packets that are not likely to be dropped in transit (e.g. well-formed packets, packets expected to be observed in the network under normal conditions, etc.).

Apple iOS 4.2.1 (iPhone)
Apple OS X 10.6.8
Apple OS X 10.7.0
Apple OS X 10.8.0
Apple TV 3.0.2
Cisco IOS 12.3(26)
Cisco IOS 15.1(3)T1
Cisco IOS (unknown version)
FreeBSD 6.1
FreeBSD 8.1
HP LaserJet M1212nf printer
HP ProCurve 2520G switch
Linux 2.6.11 (Fedora Core 4)
Linux 2.6.15 (Ubuntu 6.06)
Linux 2.6.33 (embedded in Thecus N4100 storage device)
Linux 2.6.35 (Fedora 14)
Linux 2.6.35 (Ubuntu 10.10)
Microsoft Windows 7 Professional SP1
Microsoft Windows 7 Ultimate
Microsoft Windows XP SP3
NetBSD 5.0
OpenBSD 3.8
OpenBSD 4.4
OpenBSD 4.8
OpenIndiana oi_148
OpenSolaris 2009.06

Table 1: List of operating systems against which we ran the ipv6fp.py program. These are the systems whose behavior guided our selection of probes.

We wrote a custom test program, ipv6fp.py [22], that had the ability to create all 154 network probes (crafting network packets with all required network layers), inject them into the network and capture any responses sent by the target host. We also implemented all IPv4 probes currently used by Nmap for OS detection over IPv4, so we could double-check consistency and also have the ability to tell differences in upper layers when the underlying layer-3 changes. The application was written in Python and used the Scapy library.

With the help from Nmap's users and developers, we were able to run the application against 26 different operating systems and versions, which are listed in Table 1.

We then created another application, fpanaly.py [21], to analyze the results. In particular, we wanted to determine which probes produced the highest amount of variability in the responses. In order to facilitate the analysis, our tool displayed the data from different angles:

- For each probe, the different values observed on each protocol field.

- For each probe, percentage of hosts that replied to it.
- For each protocol field, the different values observed on each probe.
- For each protocol field, the different values observed on each OS.
- For all IPv4 tests and their IPv6 equivalent, all differences in upper layers.

After analysis, we saw that 18.06% of the probes generated responses from all hosts, 72.9% were only responded by some hosts, and 9.03% were not responded at all. Discarding the probes that didn't produce any replies, we analyzed all responses to determine which ones showed the most non-trivial differences for the same probe. We narrowed down the list to a set of 18 probes, that we labeled S1 through S6, IE1, IE2, NI, NS, U1, TECN, and T2 through T7. The following sections describe the selected probes.

S1–S6: TCP sequence probes.

These are six TCP-based probes sent from different source ports but to a common open destination port (note that in our implementation, Nmap performs a port scan beforehand, so we already have a list of open ports on the target system). The probes have the SYN flag set so we force the other end to reply back with a SYN/ACK packet, following the standard TCP three-way handshake. In addition, each probe contains a different set of window sizes and TCP options, which forces the other end to negotiate the connection in a different manner for each probe and shows its level of support for optional features like TCP timestamps and window scaling. The table below shows the values that we use. Note that the Timestamp option (indicated by "TS") always has the fixed values TSval=0xFFFFFFFF, TSecr=0.

The probes in this set are sent sequentially, with a constant interval of 100 ms. This allows us to analyze the initial sequence numbers generated by the target host at specific points in time and estimate the increment rate of its internal sequence number counter, which varies across operating systems.

	TCP options	window
S1	WScale=10, NOP, MSS=1460, TS, SAckOK	1
S2	MSS=1400, WScale=0, SAckOK, TS, EOL	63
S3	TS, NOP, NOP, WScale=5, NOP, MSS=640	4
S4	SAckOK, TS, WScale=10, EOL	4
S5	MSS=536, SAckOK, TS, WScale=10, EOL	16
S6	MSS=265, SAckOK, TS	512

TECN, T2-T7: additional TCP probes.

These probes are also used to extract information about the implementation of TCP. In particular, they are designed to test the response to various combinations of TCP flags.

TECN, T2, T3, and T4 are sent to an open port while T5, T6 and T7 are sent to a port that Nmap reported as closed (a closed port is one that responds to a SYN probe with a RST, not one that simply drops SYNs).

TECN sets the SYN flag, plus the two high-order bits of the flags field, ECE and CWR, which are used for explicit congestion notification [28]. It additionally sets the urgent pointer, despite not setting the URG flag. T2 sets no flags at all; T3 sets FIN, SYN, PSH, and URG; T4 and T6 set ACK alone; T5 sets SYN and T7 sets FIN, PSH, and URG. The options and window sizes vary as shown:

	TCP options	window
TECN	WScale=10, NOP, MSS=1460, SAckOK, NOP, NOP	3
T2	WScale=10, NOP, MSS=265, TS, SAckOK	128
T3	WScale=10, NOP, MSS=265, TS, SAckOK	256
T4	WScale=10, NOP, MSS=265, TS, SAckOK	1024
T5	WScale=10, NOP, MSS=265, TS, SAckOK	31337
T6	WScale=10, NOP, MSS=265, TS, SAckOK	32768
T7	WScale=15, NOP, MSS=265, TS, SAckOK	65535

Note that there is no separate T1 probe, since the S1 probe already fills that purpose. The probe names were chosen to keep harmony with the names of the existing IPv4 probes.

U1: UDP probe.

This probe is a UDP packet with a fixed payload (300 octets with the value 0x43), sent to a closed port. Since the UDP protocol does not have any built-in mechanism to indicate a closed state, the expected response is an ICMPv6 Destination Unreachable message. We added this probe because of its historical use in Nmap's IPv4 OS fingerprinting engine, but with no real evidence that the probe provides meaningful information to the classifier.

IE1, IE2, NI: ICMPv6 probes.

This is a set of three probes whose purpose is to test how the target host responds to different types of ICMPv6 packets.

IE1 is a regular ICMPv6 Echo Request message (ICMPv6 type=128), similar to those that sent by the ping utility but with its ICMPv6 field set to 0x09, even though RFC 4443 does not specify a code other than 0x00 [8]. The reason why we set an arbitrary code is because we found that some systems use the same value in their reply while others use zero.

IE2 is also an ICMPv6 Echo Request but in this case, it is designed to test how the remote host handles erroneous extension headers. The probe includes four extension headers: Hop-by-Hop, Destination Options, Routing and again, Hop-by-Hop. Such headers are not compliant with the standard since no header other than Destination Options is supposed to appear more than once, and the Hop-by-Hop header is supposed to appear in the first position only.

Since the packet is incorrect, it is rightly rejected by all the IPv6 implementations that we tested. However, operating systems disagree about what exactly is the nature of the problem and they reject it in different ways.

A small number of implementations (7.69%) discard the packet. All the rest respond with an ICMPv6 Parameter Problem error (type 4). Our tests show that some implementations set the code field to 0 ("erroneous header field"), some others to 1 ("unrecognized Next Header type"), and the rest to 2 ("unrecognized IPv6 option"). In addition, the Parameter Problem message has a field that allows the system originating the packet to point out which part of the original packet is the cause of the problem. That field, the Parameter Problem Pointer, also showed different values across the set of analyzed implementations.

The NI probe is a Node Information query [11] that asks the target to send back its IPv4 addresses. Its Qtype is 4 ("IPv6 addresses") and it has the "A" flag ("all unicast addresses") set. Interestingly, all the operating systems that we have seen responding to this probe (including OpenBSD

and OS X) seem to misinterpret it and send back a host name, not a list of addresses, as if the Qtype had been 2 ("node name").

NS: Neighbor Solicitation probe.

The final probe, NS, uses the Neighbor Discovery protocol [25], which is the equivalent of IPv4's Address Resolution Protocol (its main purpose is to allow the discovery of the link-layer address that belongs to a particular IP address). The expected response to the probe is a Neighbor Advertisement message containing the target's link-layer address. Our tests show that different implementations set different flags in the reply, or even add extra options to it.

Note that in our implementation, the NS probe is only sent to hosts on the same local network segment, since the standard forbids hosts from replying to Neighbor Solicitations if the hop limit is not 255.

4. FEATURE VECTORS

Nmap receives the responses to its probes as flat byte buffers. These it parses and converts to a single feature vector. As of Nmap version 6.49, an IPv6 feature vector consists of 676 features. Most of the features are associated with a single response packet, and therefore are named after the corresponding probe. For example, the hop limit is extracted from every IPv6 packet in features named S1.HLIM, S2.HLIM, S3.HLIM, etc. The only exception to this rule is the TCP_ISR (initial sequence number rate) feature, which is a derived from all the responses to the S1–S6 probes.

Features are scaled and offset to lie within the range $[0, 1]$. The necessary scaling offsets and coefficients are stored with the computed model and used to scale test samples at runtime.

In practice, many fingerprints do not have complete information. For example, if a host lacks a closed TCP port, then there is no way to measure the responses to the T5, T6, and T7 probes. Presumably the host would send some kind of response if firewall rules allowed it—we just do not know what it would be. The fact that a host did not send a response to certain probes should not necessarily prevent it from being classified the same way as another that did send a response. On the other hand, the pattern of probe responses and non-responses, in the absence of any interference by firewalls, can be a useful OS-specific signature. The absence of certain features is always meaningful; for example, a lack of TCP options.

We handle incomplete samples by reserving two special feature values: MISSING and UNKNOWN. Postprocessing on the feature vectors converts them to concrete numerical values. MISSING features are those whose absence is presumed to be meaningful; for example missing TCP options. UNKNOWN features are those that are presumed to exist, but that could not be probed for some reason, for example those belonging to a closed port on a host that does not have a closed port. Although the distinction between MISSING and UNKNOWN is maintained in vectorization, the current engine discards the information and maps both values to -1. Section 6.2 describes ongoing work to impute the values of UNKNOWN features.

The remainder of this section describes each of the features.

IPv6 packet features: PLEN, TC, HLIM.

These generic features are extracted from every IPv6 response: PLEN, the total packet length; TC, the value of the traffic class field; and HLIM, the guessed initial hop limit. All of them have numerical values that map directly to features.

TCP window size: TCP_WINDOW.

The TCP window size, ignoring window scaling.

TCP flags: TCP_FLAG_{F,S,R,P,A,U,E,C,RES8–10}.

These 12 zero–one features correspond to the 8 TCP flag bits, plus the four adjacent reserved bits.

TCP options: TCP_OPT_0–15, TCP_OPTLEN_0–15.

These features record the first 16 TCP option types and their lengths. (Any options after the 16th are ignored, but our OS database does not contain any response with more than 9 options.) As an example, consider these TCP options (which happen to come from a SonicWALL firewall device):

NOP, NOP, MSS=1440, SAckOK, NOP, WScale=0

These options lead to the TCP_OPT features:

$$[1, 1, 2, 4, 1, 3, \text{MISSING}, \ldots, \text{MISSING}]$$

and the TCP_OPTLEN features:

$$[1, 1, 4, 2, 1, 3, \text{MISSING}, \ldots, \text{MISSING}].$$

(NOP has type 1 and length 1, MSS has type 2 and length 4, and so on.) These features ignore the options' values. Separate features, described in the next paragraph, extract the values of some common options.

TCP options: TCP_MSS, TCP_SACKOK, TCP_WSCALE.

These features represent the values of some commonly used TCP options. TCP_SACK is a Boolean feature that indicates whether the SAckOK (selective acknowledgment) option is present. With the options string in the previous paragraph, these features would have the values:

TCP_MSS=1440
TCP_SACKOK=1
TCP_WSCALE=0

TCP initial sequence rate: TCP_ISR.

This feature records the average rate of increase of the sequence numbers in the responses to the S1–S6 probes. Many versions of Linux, for example, use a 1 GHz counter.

5. CLASSIFICATION

The training database is composed mainly of user submissions, with some manual additions by us. The database contains known fingerprints, grouped into classes that have textual labels like "Linux 2.6.38 – 3.2" and "Microsoft Windows 7 Professional SP1". In addition to these freeform labels, each class has one or more machine-readable classification strings, which take the form of "*vendor* | *family* | *version* | *type*" quadruples such as "Linux | Linux | 2.6.X | general purpose" and Common Platform Enumeration [26] strings like "cpe:/o:linux:linux_kernel:2.6".

Using the database of training samples, we train a linear model using LIBLINEAR [14] in its L_2-regularized logistic regression mode. The resulting model is embedded in the Nmap binary. LIBLINEAR produces a number of one-versus-rest binary classifiers, one for each OS class. At runtime, an observed feature vector is tested against each of these classifiers. For compatibility with the score format of the older IPv4 classifier, scores are mapped to the range 0–100%. A match is reported when a class scores 90% or greater and it has sufficiently low novelty; that is, if it does not appear to be too dissimilar from the other feature vectors in the class.

Naively applied, the linear model would not perform well when faced with an feature vector unlike any it has seen before. When faced with a totally new feature vector, one not related to any in the training set, the classifier's output can be nonsense. In the early days of deployment, we observed the naive classifier often to settle on an obscure class with a score near 100%. To deal with this issue, the classifier employs a "novelty threshold" that avoids returning a match when an observed feature vector is sufficiently unlike the members of the winning class. This is not only to avoid presenting users with meaningless output, but also because when there is no match, the system displays a fingerprint and encourages the user to submit it so that the new operating system may be matched in a future version.

Some means of novelty detection is essential in OS detection, because it is not possible to anticipate the universe of possible operating systems. It is easy enough to get good coverage of common desktop and server systems, but the problem is fraught with the huge number of non-mainstream, one-off and custom OSes such as those running in embedded devices. Such embedded devices—for example routers, printers, and firewalls—compose 30% of the IPv6 OS database. Even well-known, mainstream operating systems change over time. Especially given that the primary source of updates to the database is user contributions, the engine must know when to hazard an OS guess, and when to admit ignorance and display a fingerprint for submission.

Our definition of novelty is, briefly, the Euclidean distance from the observed feature vector to the mean of a class, in a space where every dimension is scaled by the inverse of the variance of the corresponding feature. The novelty computation can be viewed as an approximation of the Mahalanobis distance [20]—it would be the same if features were independent. In practice the approximation performs well enough for its function of pruning outlandish matches, and it has the advantage of not needing the entire feature covariance matrix to be present at classification time. The novelty threshold is a manually tuned parameter. In Nmap version 6.49, it is set to 15.0. (Recall that there are ≈ 600 features and that features are scaled to the range $[0, 1]$, or $[-1, 1]$ when considering missing and unknown values. A non-novel vector is one whose length is less than 15.0 in this space.)

When the variance of a feature is zero, as happens when there is only one member of a training class or when a feature takes on only one value within a class, the inverse of the variance is not defined. In this case, we artificially set the variance to a small constant. This has the effect of making any difference from the mean seem highly novel, and makes the engine likely to print a fingerprint for submission if there are even a few such differences. The constant is a parameter that compromises between the desire to acquire a new training fingerprint with a never-before-seen feature value, and the desire to give the user an OS match when the rest of the features match up. It is unfortunately common for classes to contain just one member—another effect of the diversity of OSes in the wild—making the need to acquire new fingerprints for them more acute.

Evaluation of the classifier is hampered by the difficulty of obtaining ground-truth testing samples. To obtain ground truth, one needs to scan a host remotely and to have some level of access to independently verify its operating system. Even cross-validation is difficult, because there are so many training samples that are the sole member of their class—they have no chance of being classified correctly if held out during cross validation. In our current database, 15% of training samples are members of a one-element class, and another 9% are members of a two-element class. The only classes that have more than 10 elements are common desktop OSes: Linux, Windows, and OS X. With these caveats in mind, the accuracy of leave-one-out cross validation on our training set of 285 samples is 70.2%. (Near misses, for example Linux being classified into the wrong range of kernel revisions, are counted as inaccurate classifications for the purpose of this calculation.)

6. WORK IN PROGRESS

In this discussion we discuss fingerprinting possibilities we have discovered and enhancements to the fingerprinting engine we are in the progress of testing and implementing.

6.1 New probes

We have described how we selected our current set of probe packets. Since then, we have learned of other IPv6-specific fingerprinting features that could be tested with additional probes. In choosing whether to implement a new probe, we weigh its potential benefit against the cost of additional network traffic and implementation complexity.

IPv6 lacks the fragment identifier field that is a part of every IPv4 packet, depriving us of tests that measure how sequences of identifiers are generated (for example, whether incremental or random, and whether global or host-specific). Instead, fragmentation in IPv6 uses an extension header that is not sent by default. However, by sending additional probes, it is possible to force a remote host to append the fragmentation extension header to its replies [24, 15], enabling the analysis of identifier sequences.

There are more possibilities for experimentation with extension headers. Probes can be sent using legal and illegal combinations of headers, in different orders of those headers. One example for this is the Routing header, which is intended to provide a list of intermediate node addresses as well as a count of how many of them are still left to be visited. This header could for example contain a very big list of nodes, or a node count that does not fit within the header bounds. In experiments with variations on the Routing header, we have observed hosts to reply normally, to report an erroneous header or destination unreachable error, or to simply drop the probe.

The Multicast Listener Discovery (MLD) protocol [34] allows discovery of multicast listeners on the local link. By sending MLD queries onto the local network, it is possible to identify different OSes by the different multicast groups they listen to [2]. Like the existing NS probe, an MLD probe would work only on the local network.

We have found that even hosts with strict firewalls will respond with an ICMPv6 Time Exceeded message when they time out reassembling a fragmented packet. The Time Exceeded message will contain a hop limit, which allows a coarse determination of the OS. To induce such a reply, it is necessary only to send the first fragment of a fragmented message and wait for a timeout.

There are multiple bugs in IPv6 stacks, such as operating systems dropping ICMPv6 Packet Too Big messages that declare a path MTU smaller than the IPv6 minimum of 1280 bytes, and hosts that stop responding at all or that respond with incorrect TCP checksums after receiving such a message [24]. These differences, despite being potentially disruptive, allow for a better OS guess.

6.2 Feature imputation

Because network connections are not always reliable and packets may be dropped by intermediate nodes on a path between two devices, some of the fingerprints in the training set are not complete. This leads to incomplete feature vectors and a decrease in the accuracy of classification. Scientific literature proposes different solutions to the problem of missing data, although for practical reasons, not all of them are applicable to our scenario. For example, one method of dealing with missing data is to drop all incomplete fingerprints, but the current average number of fingerprints per OS class is not high enough to permit this. Dropping existing fingerprints would negatively influence the classification accuracy. Other methods which do not rely on removing incomplete fingerprints, such as replacing missing values with the average of existing values from the same feature, also have a negative impact since they may introduce bias. We are now experimenting with imputing unknown feature values from related feature vectors.

We chose to apply multiple imputation by chained equations, following a custom-tailored approach adapted to the existing workflow. For practical reasons, we made use of an existing implementation in the R programming language [7]. Multiple imputation is not an algorithm in itself, but rather a framework that specifies how different algorithms can be combined in order to find meaningful values for missing data. The imputation process produces multiple matrices which are copies of the original feature matrix, but without missing values. Each matrix is calculated by repeatedly applying an imputation algorithm that infers missing values of one dependent feature from existing values of all other features [4]. During one iteration, the algorithm is applied once to all features that are missing values. After a certain number of iterations, the imputed matrix should stabilize and converge.

The complete multiple imputation method pools all imputed matrices which then result in a set of parameters that allow features to fit a model of choice. During our testing, we decided that feeding the complete feature matrix to the imputation algorithm was not a feasible approach and decided to group features together to form subsets of the main feature matrix. These sub-matrices are then imputed individually. Because Nmap uses a logistic regression model trained on the complete feature matrix, it was necessary to combine the results of imputing all the sub-matrices into an imputed version of the full original matrix. For this reason, we developed a pooling method that can combine multiple imputed sub-matrices, depending on the type of the variables they contain (e.g. binary, categorical, or continuous).

While developing this imputation method for Nmap's training feature matrix, a number of additional research questions arose. While some have been answered, there is ongoing research that aims to perfect the scheme before being put into practice. The questions fall into four topics that influence the final imputed matrix:

- The exact set of features to impute.
- The exact ways in which to group features into sub-matrices.
- The number of imputed sets for each sub-matrix.
- The number of iterations per imputed set.

These questions influence two main factors, namely the time required to run imputation and the quality of the data in the final imputed matrix. The problem of which set of features to impute was solved by applying a recursive feature elimination algorithm that finds the features that have the greatest effect on accuracy during classification. We used the implementation in the scikit-learn library [27]. Having a minimal set of features, we did further testing to discover how they should be grouped. The results show that features ought to be grouped such that (1) relevant information links them together (i.e., they are correlated); and (2) they are of the same type. The choices for the number of imputed sets and number of iterations per set affect not only the amount of time for execution, but also the quality of the final imputed matrix. Too few iterations mean that imputed sets do not converge, while too many iterations require more time to finish.

Our experiments indicate that imputation improves the logistic regression model. Without imputation, the missing values are mapped to -1 for the purpose of training, and as a result the optimization process (e.g., gradient descent) that computes the weights adds penalization. When the features are imputed with meaningful values, the weights are penalized less and the resulting model becomes more accurate. Imputation realizes increases of 1–2% in cross-validation accuracy and of the logit model even more when the model is used directly with Nmap in our test environment.

7. DEPLOYMENT

The machine learning–based OS classification system we have described was first released in September 2011 in Nmap version 5.61TEST2. It originally had 62 OS classes, collected from our and volunteers' scans during development. We had put out a call for fingerprint submissions using the prototype prober script of Section 3 in July 2011. The first version of the engine did not include the novelty threshold described in Section 5, which probably hindered the rate of submission of new fingerprints until it was integrated in version 5.61TEST5 in March 2012.

When a user runs Nmap OS detection against a target and the classification engine fails to find a sufficiently good match, Nmap prints a fingerprint and asks the user to submit it to a web form, along with any information the user has about the remote operating system. (The user can override this behavior through the `--osscan-guess` option, which forces the program to print the best OS match even if too novel.) Despite this channel of user contributions, the collection of adequate ground-truth training data remains

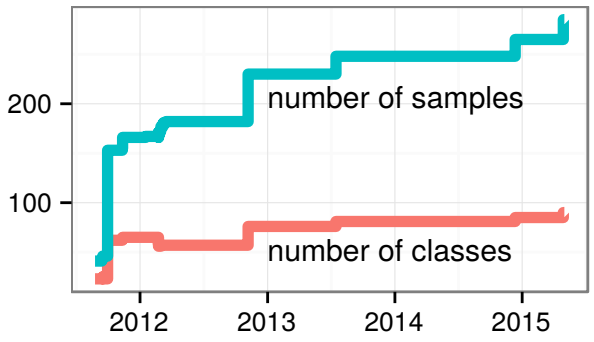

Figure 2: Growth of the IPv6 database over time. Decreases in the number of classes reflect consolidation after manual re-evaluation of training samples.

challenging. Figure 2 shows how the IPv6 OS database has grown over time. The adoption of IPv6 is still far below that of IPv4, and the difference in submission rates between IPv4 and IPv6 is stark. Between June 2013 and February 2015, we received over 4,700 IPv4 submissions. In the same period, we received only 97 for IPv6. We would prefer to have more training samples, in order to reflect real-world diversity in end hosts and network routes, while still keeping distinguishable OS versions in separate, specific classes.

Assigning newly submitted fingerprints to classes is a supervised, mostly manual affair. There are few enough submissions that each one is reviewed manually. The submission form asks the user for information such as how they know what OS is running, and the OS version down to the patch level (using the output of `uname -a` or an equivalent). We have not detected any malicious submissions that appear intended to poison the classification database, though the process of vetting submissions is admittedly subjective and involves human judgment. In reviewing submissions, we look for evidence that the submitter really knew what OS was running, and cross-check the submission against the existing database to see that it is not obviously some other OS. A large fraction of submissions are ignored for a lack of information. Occasionally we follow up with the submitter by email in order to get more details.

A database fingerprint consists of the raw byte contents of the responses to each probe, with certain sensitive fields like the source and destination addresses redacted. There are extra metadata fields for information that cannot be recovered from packet contents alone, such as the time offset at which the response was received. We store these redacted packet dumps, not the processed feature vectors, so that we may add or remove features, or change how features are computed, without having to discard past training samples.

Our system is mostly designed to work against operating systems in their usual configurations, not against adversarial configurations that deliberately obscure their network fingerprint. Most of our probes are distinctive and would be easy to detect and block. Some of the features we use—the initial hop limit for example—are easily modifiable from user space. Our classifier takes advantage of the fact that settings like these are rarely changed away from the defaults—and when they are, they may represent a custom distribution that should be separately classed. It would be difficult to modify one operating system to completely imitate another at the network level. A better way to frustrate fingerprinting is to pass traffic through a gateway that scrubs differences in network stacks, though even this approach presents difficulties [32].

There can arise interesting cases of "hybrid" fingerprints, in which some probes receive responses from one host and other probes from another. In this case, the fingerprint will have a mixture of features from different systems and what label it should receive is not well defined. This can happen, for example, with virtualized systems and transparent proxies. (We once saw a high-latency satellite link that would spoof SYN/ACKs in response to SYN on any port, evidently to decrease perceived latency—a port scan through such a link would show every port on every host as open.) The system we have described in this paper fingerprints whatever device ultimately generates the network- and transport-layer packets, which may not match the user's idea of what the target host really is. Apart from the system we have described, Nmap also has an independent application-layer OS classifier that scrapes version information from SSH banners and the like. It can happen that the low-level OS classification is different than the application-layer classification—and that nevertheless both are correct. An example of this is an web server running behind a load balancer: the low-level fingerprint will be that of the load balancer and the application-layer fingerprint will be that of the web server.

8. SUMMARY

We have presented a machine learning–based classifier for operating systems based on remotely measurable network characteristics over IPv6. The classifier is deployed in the popular Nmap security scanner. We described the challenges and opportunities of operating system fingerprinting over IPv6, explained the reasoning behind our design decisions, and introduced work in progress to improve the classifier.

9. ACKNOWLEDGMENTS

This work was supported in part by funding from the Open Technology Fund through the Freedom2Connect Foundation and from the US Department of State, Bureau of Democracy, Human Rights and Labor. The opinions in this paper are those of the authors and do not necessarily reflect those any funding agency or governmental organization.

10. REFERENCES

[1] O. Arkin and F. Yarochkin. Xprobe v2.0: A "fuzzy" approach to remote active operating system fingerprinting. Technical report, Aug. 2002. http://www.ouah.org/Xprobe2.pdf.

[2] A. Atlas. MLD considered harmful?, Nov. 2014. http://www.insinuator.net/2014/11/mld-considered-harmful/.

[3] P. Auffret. SinFP, unification of active and passive operating system fingerprinting. *Journal in Computer Virology*, 6(3):197–205, Aug. 2010. http://patriceauffret.com/files/publications/jcv.pdf.

[4] M. J. Azur, E. A. Stuart, C. Frangakis, and P. J. Leaf. Multiple imputation by chained equations: what is it and how does it work? *International journal of methods in psychiatric research*, 20(1):40–49, 2011.

[5] T. Beardsley. Snacktime: A Perl solution for remote OS fingerprinting, June 2003. http://www.planb-security.net/wp/snacktime.html.

[6] R. Beverly. A robust classifier for passive TCP/IP fingerprinting. In C. Barakat and I. Pratt, editors, *Passive and Active Network Measurement*, volume 3015 of *Lecture Notes in Computer Science*, pages 158–167. Springer Berlin Heidelberg, 2004. https://www.cl.cam.ac.uk/research/srg/netos/pam2004/papers/260.pdf.

[7] S. Buuren and K. Groothuis-Oudshoorn. mice: Multivariate imputation by chained equations in R. *Journal of statistical software*, 45(3), 2011. http://www.jstatsoft.org/v45/i03/paper.

[8] A. Conta, S. Deering, and M. Gupta. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) specification. RFC 4443, RFC Editor, Mar. 2006. https://tools.ietf.org/html/rfc4443.

[9] J. Corbet. TCP window scaling and broken routers, July 2004. https://lwn.net/Articles/92727/.

[10] J. Corbet. Increasing the TCP initial congestion window, Feb. 2011. https://lwn.net/Articles/427104/.

[11] M. Crawford and B. Haberman. IPv6 Node Information queries. RFC 4620, RFC Editor, Aug. 2006. https://tools.ietf.org/html/rfc4620.

[12] S. E. Deering and R. M. Hinden. Internet Protocol, version 6 (IPv6) specification. RFC 2460, RFC Editor, Dec. 1998. https://tools.ietf.org/html/rfc2460.

[13] J. Edge. Networking change causes distribution headaches, Oct. 2008. https://lwn.net/Articles/304791/.

[14] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008. http://www.csie.ntu.edu.tw/~cjlin/liblinear/.

[15] F. Gont. Processing of IPv6 "atomic" fragments. RFC 6946, RFC Editor, May 2013. https://tools.ietf.org/html/rfc6946.

[16] L. G. Greenwald and T. J. Thomas. Toward undetected operating system fingerprinting. In *1st USENIX Workshop on Offensive Technologies*, 2007. https://www.usenix.org/event/woot07/tech/full_papers/greenwald/greenwald.pdf.

[17] V. Jacobson, B. Braden, and D. Borman. TCP extensions for high performance. RFC 1323, RFC Editor, May 1992. https://tools.ietf.org/html/rfc1323.

[18] G. Lyon. Remote OS detection via TCP/IP stack fingerprinting. *Phrack*, 8(54), Oct. 1998. https://nmap.org/nmap-fingerprinting-old.html.

[19] G. Lyon. *Nmap Network Scanning*, chapter Remote OS Detection. 2009. https://nmap.org/book/osdetect.html.

[20] P. C. Mahalanobis. On the generalised distance in statistics. *Proceedings of the National Institute of Sciences of India*, 2(1):49–55, 1936.

[21] L. MartinGarcia. [tool] fpanaly.py: IPv6 OS detection test result analysis tool, June 2011. https://svn.nmap.org/nmap-exp/luis/ipv6tests/fpanaly.py?p=34606.

[22] L. MartinGarcia. [tool] ipv6fp.py: IPv6 OS detection test suite, June 2011. https://svn.nmap.org/nmap-exp/luis/ipv6tests/ipv6fp.py?p=34606.

[23] J. P. S. Medeiros, A. C. da Cunha, A. M. Brito, and P. S. Motta Pires. Automating security tests for industrial automation devices using neural networks. In *IEEE Conference on Emerging Technologies and Factory Automation*, pages 772–775, Sept. 2007.

[24] M. Morbitzer. TCP idle scans in IPv6. Master's thesis, Radboud University Nijmegen, Aug. 2013. https://www.ru.nl/publish/pages/578936/m_morbitzer_masterthesis.pdf.

[25] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor discovery for IP version 6 (IPv6). RFC 4861, RFC Editor, Sept. 2007. https://tools.ietf.org/html/rfc4861.

[26] National Institute of Standards and Technology. Common Platform Enumeration (CPE). https://nvd.nist.gov/cpe.cfm.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[28] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ECN) to IP. RFC 3168, RFC Editor, Sept. 2001. https://tools.ietf.org/html/rfc3168.

[29] D. W. Richardson, S. D. Gribble, and T. Kohno. The limits of automatic os fingerprint generation. In *Proceedings of the 3rd ACM Workshop on Artificial Intelligence and Security*, AISec '10, pages 24–34, New York, NY, USA, 2010. ACM. https://homes.cs.washington.edu/~yoshi/papers/fuzzing_aisec2010.pdf.

[30] C. Sarraute and J. Burroni. Using neural networks to improve classical operating system fingerprinting techniques. *Electronic Journal of SADIO*, 8(1):35–47, 2008. http://www.coresecurity.com/files/attachments/Sarraute_EJS.pdf.

[31] Z. Shamsi, A. Nandwani, D. Leonard, and D. Loguinov. Hershel: Single-packet OS fingerprinting. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 195–206, New York, NY, USA, 2014. ACM. http://irl.cse.tamu.edu/people/zain/papers/sigmetrics2014.pdf.

[32] M. Smart, G. R. Malan, and F. Jahanian. Defeating TCP/IP stack fingerprinting. In *9th USENIX Security Symposium*, Aug. 2000. https://www.usenix.org/legacy/publications/library/proceedings/sec2000/full_papers/smart/smart.pdf.

[33] F. Veysset, O. Courtay, and O. Heen. New tool and technique for remote operating system fingerprinting. Apr. 2002. http://www.gomor.org/files/ring-full-paper.pdf.

[34] R. Vida and L. Costa. Multicast listener discovery version 2 (mldv2) for ipv6. RFC 3810, RFC Editor, June 2004. https://tools.ietf.org/html/rfc3810.

[35] F. V. Yarochkin, O. Arkin, M. Kydyraliev, S.-Y. Dai, Y. Huang, and S.-Y. Kuo. Xprobe2++: Low volume remote network information gathering tool. In *IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 205–210. IEEE, 2009. http://xprobe.sourceforge.net/xprobe-ng.pdf.

[36] M. Zalewski. p0f - passive os fingerprinting tool, June 2000. http://seclists.org/bugtraq/2000/Jun/141.

[37] M. Zalewski. [tool] the new p0f 2.0.1 is now out, Sept. 2003. http://seclists.org/bugtraq/2003/Sep/36.

[38] M. Zalewski. p0f3 release candidate, Jan. 2012. http://seclists.org/fulldisclosure/2012/Jan/123.