Passive OS Fingerprinting on Commodity Switches

Sherry Bai, Hyojoon Kim, and Jennifer Rexford
Princeton University, Princeton, NJ, USA

***Abstract*—Operating System (OS) fingerprinting allows network administrators to identify which operating systems are running on the hosts communicating over their network. This information is useful for detecting OS-specific vulnerabilities and for administering OS-related security policies that block, rate-limit, or redirect traffic. Passive fingerprinting can identify hosts' OS types without active probes that introduce additional network load. However, existing software-based passive fingerprinting tools cannot keep up with the traffic in high-speed networks. This paper presents P40f, a tool that runs on programmable switch hardware to perform OS fingerprinting and apply security policies at line rate. P40f is a P4 implementation of an existing software tool, p0f. We present our prototype implemented with the P4 language, which compiles and runs on the Intel Tofino switch. We present experiments against packet traces from a real campus network, and make our code publicly available.**

I. Introduction

Information about the operating systems running on end hosts is important for managing enterprise networks. In particular, network administrators use host OS information for keeping device inventory up-to-date, urging users to perform OS upgrades, and applying different firewall rules based on OS type. For example, hosts running outdated OSes (*e.g.*, Windows XP) may be vulnerable to security exploits that may cause them to be compromised; OS fingerprinting can help a network administrator to check how many internal hosts are using outdated OSes and pinpoint them by IP address.

Active probing is a well-known mechanism for OS fingerprinting. Tools such as Nmap [1] and ZMap [2] send probes designed to elicit unusual or distinctive responses from target hosts to reveal OS-specific quirks. However, active fingerprinting has a number of disadvantages. First, such tools exchange multiple packets with each host, leading to longer scan times and extra network load as the number of hosts grows in the network. Second, active probes might miss many hosts, leaving them unaccounted for. For example, active probes have difficulty scanning hosts behind network address translators (NATs). Some hosts plainly block or ignore such probes. Active fingerprinters also cannot run against external hosts that are contacting the internal hosts of the network, which is often essential for network forensics.

Passive OS fingerprinting tools [3], [4], in contrast, monitor existing network traffic to identify a host's OS in real time, while avoiding the need for periodic scans or bypassing of NATs and firewalls. Passive OS fingerprinters are a better fit to today's dynamic bring-your-own-device (BYOD) networks (*e.g.*, eduroam on college campuses [5]), which do not mandate host registration and where hosts come and go rapidly. Existing software passive fingerprinters, however, introduce several challenges in today's networks. First of all, software cannot keep up with large amounts of traffic on ever-increasing high-speed networks. As this was identified as a real operational problem, tools like `k-p0f` were developed, designed for high throughput. However, `k-p0f` also experiences a 38% degradation in throughput on a saturated gigabit link [6]. Load balancing on multiple servers running Data Plane Development Kit (DPDK) [7] might work, yet it is challenging to build and manage such infrastructure. Second, an out-of-band monitoring system cannot take immediate actions on traffic (*e.g.*, blocking, redirecting, or rate-limiting) based on the OS information as packets pass by; the external monitoring system would have to send a separate control-plane message to the data plane to take action, which incurs additional communication load and delay. These limitations could be avoided if *network devices* can perform OS fingerprinting and take direct action on packets *on-the-fly*.

To this end, we present P40f, a passive fingerprinter that runs directly in the data plane. We utilize Protocol Independent Switch Architecture (PISA) programmable switches [8], [9] that provide flexible packet processing, which enables both OS fingerprinting and security policy enforcement on the switch. P40f fingerprints OS type by analyzing TCP header and option fields in TCP SYN packets, a powerful technique used by p0f [3], a popular passive OS fingerprinter implemented in software. P40f provides OS fingerprinting that matches the interface's packet processing rate, whether it is 10, 40, or even 100 Gbps. In addition, P40f can take direct action, such as allow, drop, or redirect, on packets based on their OS information directly in the data plane.

Implementing and running an OS fingerprinter in a programmable data plane is challenging. Switches impose more strict constraints on packet parsing and processing when compared to machines with general-purpose CPUs. For example, the number, types, and lengths of fields in a TCP option differ from option to option, but switch parsers cannot process headers that contain a variable number of fields or that contain variable-length fields in the beginning or middle of the header. This makes fine-grained TCP option processing difficult. The p0f tool also requires performing division between some TCP option fields, but hardware switches cannot perform such complex arithmetic operations correctly.

P40f presents the following contributions to overcome the challenge and provide OS fingerprinting functionality for practical use in real enterprise networks.

Complete, versatile TCP option parser. P40f implements a complete, versatile, and efficient TCP option parser in the data plane, dealing with any combination of TCP options with

```
ver:ittl:olen:mss:wsize,scale:olayout:quirks
*:64:0:*:20,10:mss,sok,ts,nop,ws:df,id+
```

Figure 1: The first line is the format of a p0f v3.x TCP signature. The second line is an example signature for "Linux 3.11 and newer".

a variable number of fields and variable-length fields. P40f's TCP option parser also leaves the original TCP header intact while doing so. P40f can parse up to ten TCP option fields in any order, and keeps track of the order, too.

Modular design. P40f only occupies in the ingress pipeline, and can tag the OS type inforamtion as metadata to a packet. Thus, an operator can easily add a custom P4 code that operates in the egress pipeline to create any network application that uses the OS type information.

P40f is an early prototype that demonstrates the power of programmable data planes. P40f compiles and runs in Intel's Tofino-based switches [9], and we open-source our code [10].

II. Background: P0f Overview

P40f is based on p0f [3], a popular passive OS fingerprinter. The software is written in *C*, and it compiles and runs on servers with general-purpose CPUs. The p0f software monitors a network interface and analyzes packets that appear on the interface. To perform OS fingerprinting on a packet, p0f first extracts information from the packet's TCP/IP headers. Then it compares this information to entries in a fingerprint database file to match the packet to an operating system label. Unsurprisingly, the software faces performance issues when the incoming packet rate becomes too high.

A p0f TCP *signature* is a string that specifies and enumerates the values, which are in the packet, needed to identify an OS or application. Figure 1 shows the format of a signature in the fingerprint database and an example signature for the OS label "Linux v3.11 and newer". A signature has nine colon-delimited fields; Table I summarizes them.

There are two different *types* of signatures: *specific* or *generic*. The generic signatures match broader groups of operating systems, are considered "last-resort," and thus are given lower priority than specific signatures. An example of a generic signature is "Mac OS X" while a specific one is "Mac OS X 10.x".

III. Passive OS Fingerprinting in P4

P40f is an implementation of p0f in P4 [11], a language for specifying how to parse and process packets. Our work compiles and runs in a real data plane at *line rate*. Figure 2 illustrates P40f's architecture and packet's path through the switch. As a packet enters the data plane, P40f first extracts information from a TCP SYN packet's TCP/IP headers, populates a set of custom metadata, and then compares the metadata against a list of p0f signatures. The p0f signatures are translated and installed as match-action rules in a table in the data plane, along with the desired actions per OS label. Therefore, there is no need for control-plane interaction at run-time.

Field	Description
`ver`	Signature for IPv4 ('4'), IPv6 ('6'), or both ('*').
`ttl`	Initial TTL used by the OS. Mostly 64, 128, or 255.
`olen`	Length of IPv4 options or IPv6 extension headers.
`mss`	TCP max segment size. Supports wildcard value.
`wsize`	TCP window size. Value is a fixed integer, a multiple of MSS or MTU, a multiple of an integer, or a wildcard.
`scale`	TCP window scaling factor. Fixed value or '*'.
`olayout`	Exact layout of TCP options, including of bytes of padding after EOL option. Consists of comma-delimited strings.
`quirks`	Implementation quirks found in IP and TCP headers and in TCP options. Consists of comma-delimited strings. Examples include "don't fragment bit set" ('df+').

Table I: Main fields of a p0f v3.x TCP signature.

Type (kind)	Length	Data	Purpose
0	N/A	N/A	End of options list
1	N/A	N/A	No operation
2	4	MSS	Maximum segment size
3	3	Window	Window scale
4	2	N/A	Selected ACK permitted
5	10, 18, 26, 34	Blocks ACKed	Selective ACK
8	10	Timestamps	TCP timestaps

Table II: The list of available TCP options.

A. Parsing Variable-Length TCP Options

Aside from information contained in the standard IP and TCP header fields, the list of TCP options in a packet act as a strong fingerprint. Moreover, the order of the existing TCP options is also a strong indicator of a particular OS. Thus, we need a parser design that can: (1) successfully identify and parse every possible TCP option, (2) parse not just one but a series of options, and (3) save the order of options.

Predefined parser states: As mentioned before, the number of TCP options appearing in a packet is variable. Fortunately, the number of *available* TCP options is fixed, and each TCP option is identifiable by its option type, or kind. Also, the TCP option's length is fixed per option kind, too. Table II summarizes the available TCP options. Therefore, P40f creates a unique parser state for each TCP option kind, and each parser state knows how many bits to parse. One exception is the Selective ACKnowledgement (SACK) option kind; this option has four different possible lengths. As the P4 language and a real PISA switch hardware are not amendable to writing a program with parameters, we create four different parser states for the SACK option type, one for each length. Thus, P40f has ten different types of predefined parser states.

Sequential branching as packets fly by: With a high-level programming language for general-purpose CPUs (*e.g.*, *C*), it is straightforward to write code that can parse a series of TCP options: if/elif/else statements within a for/while loop will do.

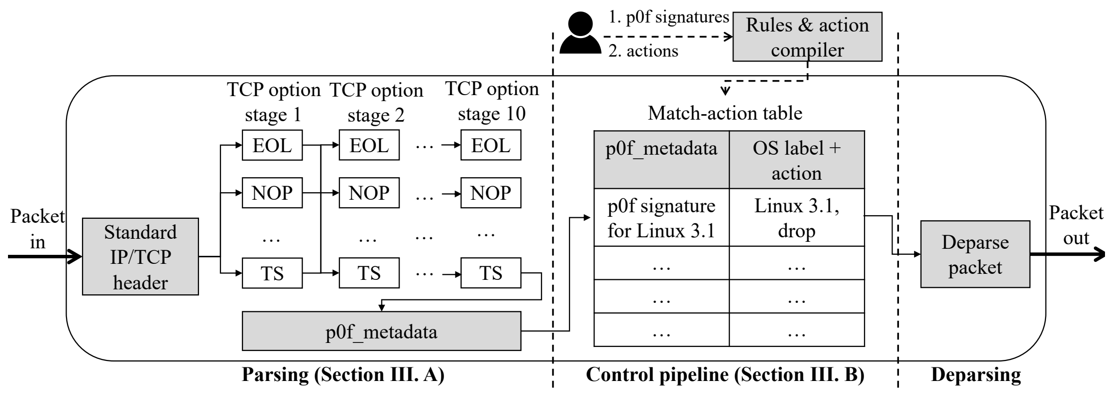

Figure 2: P40f's architecture and how a TCP SYN packet traverses the pipeline.

However, as a language for a feed-forward packet processing pipeline, the P4 language does not allow loop statements. Thus, we unfold this logic in P40f: we design P40f's parser as a sequence of parsing stages, where each parsing stage contains a set of parser states. In each parsing stage, there are ten predefined parser states, which were described above. As illustrated in Figure 2, a TCP SYN packet enters the first parsing stage and branches along in sequence until no TCP option is left or it reaches the maximum number of parsing stages. In our current prototype, we allow up to ten TCP options, or parsing stages, in sequence; there is no p0f signature that has more than ten TCP options. As a packet branches along, the order of parsed TCP options is saved, too.

Knowing when to stop with a parser counter: Per the TCP protocol standard, a special TCP option kind, *end of options list* (Table II), indicates that there are no more TCP options left in the header. Unsurprisingly, however, this standard is not always followed in various TCP/IP stack implementations in different end devices. Therefore, we implement another mechanism to identify the end of TCP options list. In particular, we calculate the length of all TCP options present in a packet and save it as a counter. This counter value can be calculated by subtracting the standard TCP header length without any TCP option, which is 20 bytes, from *data offset* value in the TCP header, which indicates the total TCP header length, including the TCP options fields. Once the counter value is set, P40f decrements this counter whenever it parses an option kind, according to the option's size. When this counter value finally reaches 0, P40f knows that all TCP options have been parsed. When the counter never reaches 0 or becomes a negative value, we consider this as a failed parse. Thus, we discard the packet and report it accordingly.

Computing P0f signature fields: In the switch, we maintain one metadata field for each p0f signature field. These metadata fields make up a structure called *p0f_metadata*. The metadata fields are populated during parsing and in the control pipeline, which then act as keys to the match-action table for OS fingerprinting. Our current prototype populates ten p0f metadata fields, including the first seven rows in Table I and three quirks that are heavily used: don't fragment bit set (df+), non-zero IPID (id+), and TCP timestamp specified as zero (ts1-).

B. Inferring OS Labels and Applying Actions

All p0f signatures are translated into rules in a table in P40f, where the match keys are fields in the signature. These match keys are compared with the packet's *p0f_metadata* once the metadata fields have been fully populated. However, some p0f signature field values are relative to another field value, which sometimes raises an issue when translating to table rules in P40f. There are also different priorities assigned to signatures based on their type. We discuss how P40f deals with these aspects, and then describe the possible actions an operator can assign to each rule.

Maximum segment size and window size: In p0f signatures, the TCP window size (p0f signature field *wsize*) is often expressed as a multiple of the TCP maximum segment size (*e.g.*, *wsize*=*mss* x2). This becomes an issue when matching on this particular field in a programmable data plane. In particular, the challenge arises when the *mss* value itself is wildcarded in the p0f signature. We can do a wildcard matching (*i.e.*, $\star$) for the *mss* value, but we cannot do the same for the *wsize* value because of the x2 condition. One idea is to enter the integer that is multiplied to the *mss* value as a match key, which is 2 in our example, then make the programmable data plane perform a division operation, $wsize/mss$, in the pipeline to match on this value. Current data planes, however, cannot perform an exact multiplication or division operation; both are expensive operations that cannot be done at line rate, currently.

To overcome this limitation, we investigate the most popular maximum segment sizes (*mss*) observed in the Internet. Based on prior work [12], [13], we select the top ten most used *mss* values, which covers around 90% of observed IPv4 TCP traffic: 1460, 1400, 1452, 1360, 1370, 1412, 1440, 1300, 1420, and 1380. For each *mss* value, we calculate the matching *wsize* according to each p0f signature, and then install a corresponding table rule. Thus, for a p0f signature with the (*wsize*=*mss* x2) condition, we will have ten rules installed (*i.e.*, (1) mss=1460, wsize=2920, (2) mss=1400, wsize=2800, and so on). Note that this technique only applies to signatures with wildcard matching on the *mss* value, thus the increase in the number of rules in limited.

Generic vs. specific rules: As mentioned in Section II, there are *specific* and *genetic* rules in p0f signatures. The original p0f software tool matches on the generic rules only after there is no match with the specific rules. We follow this process in P40f as well, using the *priority* field in the match-action table rules. In particular, the generic signatures are installed with a lower priority compared to the specific signatures.

Fuzzy matching: The original p0f software does fuzzy matching: if the key fields in a p0f signature are matching, p0f relaxes or even ignores some fields. A prime example is the Time-To-Live or *ttl* value: if all other fields are matching, p0f allows the *ttl* value to be between (*max*(0, *ttl*-35), *ttl*). This makes sense since each hop decrements the *ttl* value, thus the value depends on where the packet trace was captured. We achieve the same behavior in P40f by using a range match type for the *ttl* field for rules in the match-action table.

Supported actions on packets: P40f currently supports three policy actions: *drop_pkt*, *drop_ip*, and *redirect*. The *drop_pkt* action drops any packet matched to an OS label. The *drop_ip* action drops the packet as well as any subsequent packets received from the source IP address this packet came from. The *redirect* action redirects any packet matched to this OS label to a specified destination IP address. This destination IP is given as a parameter in the policy file.

IV. P40f Prototype

P40f prototype consists of 1,819 lines of P4 code, and it compiles and runs in Intel's Tofino-based switches [9]. In this section, we describe the resource footprint of the prototype.

Number of stages. When compiled, the processing pipeline of the P4 code spans around two-thirds of the available number of stages in the Tofino-1 switch. Yet, the program does not consume much resources in each stage. Given that hardware switches can execute multiple independent tables in parallel in a stage, production-grade P4 compilers (*e.g.*, Intel P4Studio) will easily allow other packet-processing functionality to run in parallel with P40f. Other parallel P4 functionality can likely share the stages with P40f and also additionally use the remaining one-third of the stages in full while running simultaneously with P40f on the Tofino switch. Moreover, P40f fits into the ingress pipeline, leaving the egress pipeline for any other packet-processing functionality.

Table size and memory space. The P40f installs a total of 278 rules, which are derived from the p0f signatures, in the match-action table. Our PISA-based hardware, the Edge-core Wedge 100BF-32X with Intel's Tofino chip [14], can typically hold up to tens of thousands of rules, thus the overhead is negligible. If desired, the P40f prototype can also have an internal data structure for keeping an up-to-date counter per OS type (helps produce results in Section V). For this purpose, we can use the Tofino chip's a stateful memory called register arrays. The memory required for OS counter register arrays is negligible: the p0f v3.09b database contains 45 OS labels and each counter is 64 bits wide, which totals to 360 bytes. This is less than 1% of the total available on our hardware.

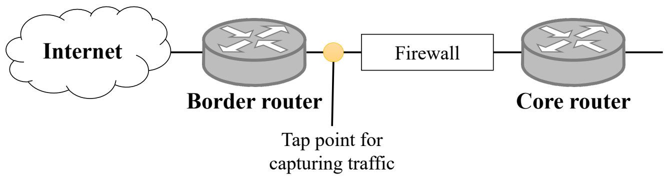

Figure 3: The edge of the university campus network. Traffic was captured at the yellow dot.

V. Preliminary Evaluation

We ran our P40f prototype against our campus network traffic to demonstrate that operators can use P40f for network-management tasks. Figure 3 shows where the packet traces were captured on our campus network. A three-hour packet trace was captured on August 19, 2020 between 8:00am-11:00am local time. There are around 1.5 millions packets in total, with an average of 138,802 packets per second. We applied a one-way hash on all internal campus IP and MAC addresses to remove any personally identifiable information (PII), and payloads were not collected. All of our data collection and evaluation processes were reviewed and approved by our institutional review board. We perform OS fingerprinting at the granularity of the source and destination IP pair. This is because a host's IP address can change over time due to DHCP or NAT. Table III summarizes the OS fingerprinting results against internal IP addresses, or clients with IP addresses assigned by our campus network. Table IV summarizes the result against external IP addresses from the Internet.

P40f generally matches p0f's output for internal hosts. As shown in Table III, P40f closely matches p0f's fingerprinting result for packets originating from internal hosts. This shows that P40f can perform fingerprinting pretty accurately even with the sacrifices needed to make the program fit in the Tofino hardware. Specifically, P40f overcounts Linux systems but undercounts Windows systems. Although the gaps are not significant, further investigation is desirable. On the other hand, P40f fingerprints Mac OS X systems with perfect accuracy. This is likely because Mac OS X systems have a very distinctive signature, which is well-defined in the signatures.

OS fingerprinting is harder against external hosts. In Table IV, P40f's result still generally matches the trend of p0f's, but with higher variability when compared to fingerprinting internal hosts. In particular, P40f is still good at fingerprinting Mac OS X systems, but there are noticeable gaps in Linux and Window categories. We leave this as future work. For both p0f and P40f, we detect a lot of SYN scan activity by the NMap tool, but P40f counts more NMap SYN scans than p0f. The NMap signature specifies that the NMap scanning tool uses a random TTL value between 0 and 64. Further investigation is required, but we suspect P40f handles this signature

OS Label	p0f-v3.09b		P40f	
	Count	%	Count	%
Linux	11412	3.05	12769	3.40
2.2.x-3.x	9558	2.56	9978	2.66
3.11+	1406	0.38	2473	0.66
3.1-3.10	332	0.09	114	0.03
3.x	39	0.01	23	0.01
Android	21	0.01	2	0.00
2.4.x	20	0.01	20	0.01
2.2.x-3.x (barebone)	15	0.00	145	0.04
2.2.x-3.x (no timestamps)	11	0.00	11	0.00
2.6.x	5	0.00	2	0.00
2.4.x-2.6.x	5	0.00	1	0.00
Windows	11753	3.14	10874	2.90
NT kernel	10202	2.73	9546	2.54
NT kernel 5.x	920	0.25	798	0.21
7 or 8	560	0.15	499	0.13
XP	65	0.02	31	0.01
NT kernel 6.x	6	0.00	0	0.00
Mac	23917	6.39	23917	6.38
OS X	23634	6.32	23634	6.30
OS X 10.x	171	0.05	171	0.05
OS X 10.9+ (iPhone/iPad)	112	0.03	112	0.03
Other	47	0.01	47	0.01
FreeBSD	37	0.01	37	0.01
FreeBSD 9.x+	9	0.00	9	0.00
NMap SYN scan	1	0.00	1	0.00
Unclassified	326918	87.40	327513	87.31
Total	374047	100%	375120	100%

Table III: SYN packets matching each OS label in the three-hour campus traffic trace for internal hosts (outgoing traffic).

incorrectly, possibly installing a rule with a wider range. We plan to fix this in our next version. Deeper investigation also shows that a large chunk of the unclassified packets are due to SYN flooding; such packets can be identified by finding TCP SYN packets without any TCP option. The presence of such activities in packets originating from the Internet towards our campus network is not surprising. Besides, the trace was captured before our firewall, as shown in Figure 3.

In general, it is hard to fingerprint external hosts with perfect accuracy due to the diversity of client types and more adversarial behavior. Yet, the ability to run passive OS fingerprinting against incoming traffic at line rate is a big win.

VI. Conclusion and Future Work

We present P40f, a work-in-progress tool that can perform passive OS fingerprinting directly in the data plane. With P40f running on a programmable switch, network operators can define and enforce security policies in the data plane by OS type against incoming and outgoing traffic without relying on external components. We prototype P40f in P4 and validate P40f's output against p0f with simulated packet traces. We then used the prototype to characterize both incoming and outgoing traffic from a real campus network.

We plan to improve P40f's fingerprinting accuracy by further investigating the mismatches we observed in our evaluation against our campus trace. We plan to improve P40f's response to SYN flood attacks and NMap scan activity. We also plan to find and use a more updated set of p0f signatures.

OS Label	p0f-v3.09b		P40f	
	Count	%	Count	%
Linux	1280209	14.28	1231089	13.56
2.2.x-3.x (barebone)	778527	8.68	681735	7.51
3.11 and newer	402081	4.48	424058	4.67
2.2.x-3.x	33986	0.38	66210	0.73
3.1-3.10	31730	0.35	26488	0.29
2.4.x	15277	0.17	14889	0.16
2.6.x	13272	0.15	12692	0.14
2.2.x-3.x (no timestamps)	3326	0.04	3370	0.04
2.4.x-2.6.x	1147	0.01	917	0.01
3.x	827	0.01	675	0.01
Android	28	0.00	23	0.00
2.0	8	0.00	32	0.00
Windows	563295	6.28	440887	4.86
7 or 8	466222	5.20	388341	4.28
XP	81245	0.91	42603	0.47
NT kernel	15086	0.17	9277	0.10
NT kernel 5.x	680	0.01	646	0.01
NT kernel 6.x	61	0.00	16	0.00
7 (Websense crawler)	1	0.00	4	0.00
Mac	1816	0.02	1816	0.02
OS X	1514	0.02	1514	0.02
OS X 10.x	295	0.00	295	0.00
OS X 10.9+ (iPhone/iPad)	7	0.00	7	0.00
Other	256666	2.86	453532	5.00
NMap SYN scan	256326	2.86	453199	4.99
FreeBSD 9.x+	221	0.00	220	0.00
FreeBSD 8.x	68	0.00	68	0.00
FreeBSD	50	0.00	44	0.00
OpenBSD 4.x-5.x	1	0.00	1	0.00
Unclassified	6864591	76.56	6951554	76.57
Total	8966577	100%	9079010	100%

Table IV: SYN packets matching each OS label in the three-hour campus traffic trace for external hosts (incoming traffic).

References

[1] G. F. Lyon, *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. USA: Insecure, 2009.

[2] Z. Durumeric, E. Wustrow, and J. A. Halderman, "Zmap: Fast internet-wide scanning and its security applications," in *USENIX Conference on Security*, 2013, pp. 605–620.

[3] M. Zalewski, "p0f v3.09b)," http://lcamtuf.coredump.cx/p0f3/, 2014.

[4] P. Auffret, "SinFP, unification of active and passive operating system fingerprinting," *Journal in Computer Virology*, vol. 6, no. 3, pp. 197–205, Aug 2010.

[5] L. Florio and K. Wierenga, "Eduroam, providing mobility for roaming users," in *EUNIS Conference*, 2005. [Online]. Available: https://www.terena.org/activities/tf-mobility/docs/ppt/eunis-eduroamfinal-LF.pdf

[6] J. Barnes and P. Crowley, "K-p0F: A high-throughput kernel passive OS fingerprinter," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2013, pp. 113–114.

[7] DPDK, "The data plane development kit," https://www.dpdk.org, 2019.

[8] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *ACM SIGCOMM Conference*, 2013, pp. 99–110.

[9] Intel, "Intel Tofino," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html/, 2019.

[10] S. Bai, H. Kim, and J. Rexford, "P40f code public repository," https://github.com/Princeton-Cabernet/p4-projects/tree/master/P40f-tofino.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[12] S. Alcock and R. Nelson, "An analysis of TCP maximum segment sizes," https://wand.net.nz/sites/default/files/mss_ict11.pdf.

[13] G. Huston, "APNIC: TCP MSS values - what's changed?" https://blog.apnic.net/2019/07/31/tcp-mss-values-whats-changed/, 2019.

[14] EdgeCore, "Edge-core Wedge 100BF-32X with Tofino," https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335.