# Passive OS Fingerprinting on Commodity Switches

Sherry Bai, Hyojoon Kim, Jennifer Rexford
Princeton University

## ABSTRACT

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 vulnerabilities and for administering OS-related security policies that block, rate-limit, or redirect traffic. Passive fingerprinting has distinct advantages over active approaches: passive fingerprinting does not generate active probes that not only introduce additional network load but could also trigger alarms and get blocked by network address translators and firewalls. 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 also self-learning; P40f collects information about traffic that cannot be fingerprinted so that new fingerprints can be learned in the future. We present our prototype implemented with P4 language along with experiments we ran against packet traces from a campus network.

## 1 INTRODUCTION

Information about the operating systems running on end hosts is important for managing enterprise networks. In particular, a host's operating system can be an indication for whether traffic from that host poses a security risk. For example, hosts running outdated operating systems may be vulnerable to security exploits that may cause them to be compromised, which can cost an organization hundreds of thousands of dollars [9]. OS information allows administrators to evaluate which threats are most likely to impact their network and to act accordingly, including:

- **Device inventory**: Administrators can take inventory of the devices connected to the network, as well as what OSes run on these devices, at any given time.
- **Access control**: Administrators can define OS-based access control policies, such as blocking devices running vulnerable OSes from connecting to the Internet.
- **OS upgrades**: Users running devices with outdated OSes can be encouraged to upgrade the OS, by directly contacting the user or by redirecting the host's web traffic to a web page with the necessary instructions.
- **Spam detection**: Machines running certain OSes (*e.g.*, Windows) are more likely than others to be spam botnet drones [38]. OS information could be used as a feature in detecting spam in email from external hosts.

Yet, while OS information about both internal and external hosts is important for network management, this information is difficult to obtain. First, while administrators can ask their users to provide information about their devices, there is no way to obtain this information from external hosts sending traffic to the enterprise. Second, obtaining accurate OS information about internal hosts may be difficult. For example, most bring-your-own-device (BYOD) networks (*e.g.*, eduroam on college campuses [13]) do not require host registration. Even in networks that require device registration, user-inputted information about a device may be incorrect or out-of-date. Administrators may need to verify this information or update the information as it changes.

Fortunately, operating system information can be obtained via other means, such as OS fingerprinting—identifying a host's OS through observations of network-level behavior. For example, because different OSes implement TCP differently, the idiosyncrasies found in TCP packets can reveal the sending host's operating system. These idiosyncrasies may include the ordering of TCP options and the relationship between maximum segment size and window size [44].

### 1.1 A Case for Passive OS Fingerprinting

Many popular OS fingerprinting tools rely on *active probing*. These tools send probes designed to elicit unusual or distinctive responses from target hosts to reveal OS-specific quirks. For example, Nmap [26] and ZMap [10] send TCP SYN packets to hosts, and then analyze the resulting SYN-ACK responses. However, active fingerprinting has a number of disadvantages. First, tools such as Nmap [26] exchange multiple packets with each host, leading to longer scan times and additional network load. Additionally, some hosts might block or just ignore such probes, which makes probing useless. Furthermore, active probes can trigger false alarms in firewalls and intrusion detection systems, especially when the number of probes is large [17]. Such middleboxes need the correct configuration and exception policies in place, which is not necessarily easy to coordinate when an organization has multiple middleboxes deployed around its network, each with different administrators (*e.g.*, departmental firewalls). Active fingerprinters also have difficulty sending probes to hosts behind network address translators (NATs). Finally, active fingerprinters cannot detect hosts that only connect briefly to the network (*i.e.*, between scans).

In contrast, *passive* OS fingerprinting tools [2, 44] monitor existing network traffic to identify a host's OS in real time, while avoiding the need for periodic scans, extra traffic, and bypassing of NATs and firewalls. Passive fingerprinting tools are used in several network-monitoring applications. For example, the pfSense firewall [30] uses the passive fingerprinting software p0f [44] to filter connections by operating system [29]. The OpenBSD Stateful Packet Filter (pf) [19]—a packet filter that resides in the kernel—can choose to pass a packet with or without modifications, silently block a packet, or explicitly reject a packet with a response. A modification to Packet Filter source code [14, 15] allows it to use p0f signatures to filter SYN packets based on OS labels.

## 1.2 OS Fingerprinting in the Data Plane

Despite the advantages of passive OS fingerprinting, existing software tools are not sufficient for practical use. Software cannot keep up with large amounts of traffic on high-speed links. Even the k-p0f tool designed for high throughput cannot monitor a gigabit link saturated with SYN and SYN-ACK packets without experiencing a 38% degradation in throughput [4]. Additionally, out-of-band monitoring systems cannot easily take action on traffic (e.g., to block, redirect, or rate-limit packets) based on the operating system. These limitations could be avoided if *network devices* could perform OS fingerprinting and take direct action on packets.

Fortunately, emerging Protocol Independent Switch Architecture (PISA) programmable switches [3, 7] provide flexible packet processing that could enable OS fingerprinting and security policies. However, to operate at line rate, these switches impose constraints on packet parsing and processing. 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. Fingerprinting tools like p0f also use the result of dividing maximum segment size by window size, but hardware switches typically cannot perform complex arithmetic operations such as division or modulo.

We present P40f, a passive fingerprinter that runs directly in the data plane. P40f presents following contributions:

**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 a variable number of fields and variable-length fields. P40f's TCP option parser also leaves the original TCP header intact while doing so. Normally, such a capability would require performing complex operations such as integer division (*e.g.*, for extracting TCP window size), which is not possible to perform in PISA switches. We utilize an enhanced "loop" technique to accurately parse various TCP options and implement a binary search in the data plane to replace complex operations (Sections 3.1 and 3.2).

**OS-based security policy enforcement in data plane.** P40f allows network administrators to express and enforce a security policy based on a host's operating system information directly in the data plane. Traditionally, network administrators need to feed live traffic to a powerful and expensive packet analyzer that can perform deep packet inspection to even just identify a host's OS type. Enforcing a policy to take action on packets based on host OS information in real time is even harder if the network is using a third-party middlebox for such purpose. P40f provides an interface for network operators to express OS-based security policies, which can allow, drop, or redirect packets based on host OS types (Section 3.3).

**Collecting data to learn new OS signatures.** TCP SYN packet-based OS fingerprinting is powerful; over 98% of outgoing SYN packets can be labeled in our campus network. Yet, some packets might be missed. P40f can forward unidentified traffic to an external system to drive the creation of new signatures. For example, the HTTP user-agent field is known to be a good host OS identifier. P40f forwards unidentifiable SYN packets with destination port 80 (HTTP) to an external analyzer that can perform OS fingerprinting using the HTTP user-agent field. P40f implements a Bloom filter to efficiently keep track of all flows associated with such packets in the resource-constrained data plane, and export only the first HTTP request packet of such flows for offline analysis (Section 3.4).

Figure 1 shows P40f's architecture. P40f is prototyped in P4 [6], a language for specifying how to parse and process packets. P40f is based on the popular p0f tool [44]. Our P40f system first extracts information from a TCP SYN packet's TCP/IP headers, then compares these fields against rules from the fingerprint database. P40f's fingerprinting capability is not affected by the traffic being encrypted (*e.g.*, SSH, HTTPS) because P40f utilizes field values in the TCP/IP header, not the payload. In addition, P40f allows administrators to specify actions to take on packets that match a particular OS label. If the OS associated with an HTTP connection cannot be identified using the TCP header, P40f forwards the SYN packet and the associated HTTP request to a software for analysis to enable the discovery of new OS signatures.

To illustrate the value of P40f for network administrators, we present the results of running P40f on packet traces of both incoming and outgoing traffic on a university campus network. We discuss the types of operating systems observed for both internal and external hosts, and also compare P40f's

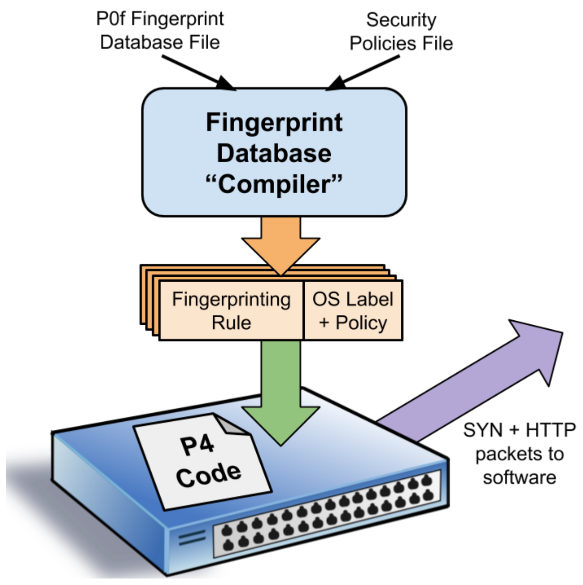
P0f Fingerprint Database File
Security Policies File
Fingerprint Database "Compiler"
Fingerprinting Rule
OS Label + Policy
P4 Code
SYN + HTTP packets to software

Figure 1: P40f architecture

output to user-agent strings observed in HTTP traffic in these traces. Because user-agent strings often contain information about the OS running on the HTTP client, we can use this information to validate whether the operating systems reported by P40f are accurate.

**Roadmap:** Section 2 presents an overview of p0f, with emphasis on the TCP signatures used to infer the OS of the sending host. Section 3 describes how P40f performs OS fingerprinting in the data plane. We evaluate our prototype of P40f in Section 4. We present a measurement case study using P40f in Section 5. Section 6 discusses related work, and Section 7 concludes the paper.

## 2 BACKGROUND: P0F OVERVIEW

P40f is based on p0f [44], a popular passive OS fingerprinter implemented in software. P0f monitors a network interface and performs passive OS fingerprinting on packets sent on this interface. To perform OS fingerprinting on a TCP SYN packet, p0f first extracts information from the packet's TCP/IP headers. P0f then compares this information to entries in a fingerprint database file to match the packet to an operating system label. A p0f TCP *signature* is a string that summarizes the information needed to identify the OS or application that sent a particular TCP packet. Signatures are organized into signature groups that are each given an OS or application label. Figure 2a shows the format of a signature in the fingerprint database, as given in the p0f README [43]. Figure 2b shows an example signature group with two signatures.

```
sig = ver:ittl:olen:mss:wsize:scale:olayout:quirks:pclass
```

**(a) Format of a p0f v3.x TCP signature**

```
label = s:unix:Linux:3.11 and newer
sig   = *:64:0:*:20,10:mss,sok,ts,nop,ws:df,id+:0
sig   = *:64:0:*:20,7:mss,sok,ts,nop,ws:df,id+:0
```

**(b) Signature group for "Linux 3.11 and newer"**

Figure 2: P0f signature format and examples

A signature consists of nine colon-delimited fields, summarized in Table 1. The field `olayout`, which captures exact ordering of TCP options, is particularly important, especially for distinguishing between different classes of operating systems (e.g., Linux vs. Windows): almost all values of `olayout` in the p0f v3.09b database are unique to one OS class.

Each signature group corresponds to an OS label. A p0f v3 OS label contains four fields: *type*, *class*, *name*, and *flavor* [43]. The *type* field specifies whether the signature is *specific* or *generic*: generic signatures match broader groups of operating systems, are considered "last-resort," and thus are given lower priority than specific signatures. The *class* field specifies the OS architecture family to which the label belongs: examples of classes include "unix", "win" (Windows), and "cisco." The *name* field specifies the name of the specific OS to which this OS label corresponds: examples of names include "Linux" or "OpenBSD." The *flavor* field contains information further qualifying the OS label, such as the OS version.

## 3 PASSIVE OS FINGERPRINTING IN P4

PISA-based programmable switch generally consists of some number of parsers, match-action pipeline stages, and deparsers. To perform OS fingerprinting on a packet, P40f first extracts packet headers, including TCP header options, in the *parser* of the switch. In the parser and match-action pipeline, P40f collects information about p0f signature fields and stores it in metadata (`p0f_metadata`). P40f then uses this metadata as a key in a *lookup table* occupying one of the final stages of the pipeline. After the lookup table match, the policy action associated with the matched OS label is executed on the packet. Finally, P40f *deparses* the packet headers and forward the packet to its next destination. Figure 3 summarizes the path of the packet through the switch.

In Section 3.1, we discuss fine-grained parsing of TCP options, which is nontrivial because of the variable length, number, and ordering of options between TCP packets. In Section 3.2, we discuss collection of p0f signature field information in `p0f_metadata`. In Section 3.3, we discuss the fingerprinting rules in the lookup table, including policy actions that may be contained in a fingerprinting rule. In

| Field | Description |
|---|---|
| `ver` | IP version. Value is '4' for IPv4, '6' for IPv6, or wildcard ('*'). |
| `ittl` | Initial TTL used by the OS. |
| `olen` | Length of IPv4 options or IPv6 extension headers. |
| `mss` | TCP maximum 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. Supports wildcard value ('*'). |
| `olayout` | Exact layout of TCP options, including of bytes of padding after EOL option (if present). 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+') and "sequence number is zero" ('seq-'). |
| `pclass` | Payload size classification. Value is '0' for zero payload size, '+' for nonzero payload size, or wildcard value ('*'). |

**Table 1: Fields of a p0f v3.x TCP signature.**

Section 3.4, we discuss collecting data about packets that cannot be identified within the switch such that p0f signatures can be learned and verified over time.

## 3.1 Parsing Variable-Length TCP Options

TCP options contain information critical for P40f's OS fingerprinting. P0f signature fields such as `mss`, `scale`, and `quirks` all rely on information contained in TCP option fields. The `olayout` signature field requires capturing the exact ordering of TCP options in a packet, and certain `quirks` signature fields require detection of multiple options of the same type.

Fine-grained parsing of TCP options, however, is difficult because a TCP packet can contain a variable number of options, each of variable length and containing variable configurations of fields. Performing such parsing in a switch at line rate is particularly challenging. In P4, each header and its constituent fields must be explicitly defined for the header to be extracted in the parser, but the variability of TCP options makes it challenging. Dapper [16] approaches this problem by creating a parsing "loop". Dapper first identifies the type of option being parsed using the option's "kind" byte, then passes control to a sub-parser for this specific option type. However, Dapper defines a fixed deparsing order, meaning that packet options may be modified after switch processing. As a result, the ordering of options in the deparsed packet may be different from that of the original packet, and duplicate options in the original packet can be lost when the packet is sent to its next destination, which is undesirable. A TCP options parser written by Andy Fingerhut [12] also uses a parsing "loop," first reading the option kind byte and then passing control to a subparser. Fingerhut's parser resolves Dapper's ordering issue by using a P4 header stack, which can store options in their exact original order, including multiple options of the same type. However, this parser does not adhere to the TCP standard [37] because it assumes that all options have both kind and length fields. Furthermore, the parser does not support common TCP options recognized by p0f, such as the `timestamp` option (kind 8).

P40f's parser extends on Fingerhut's TCP options parser by adding support for options without length fields, as well as for the remaining option types recognized by p0f. Additionally, because the switch can only access the"front" (most recently extracted) option in a header stack, all collection of TCP options data must be done while each option is parsed. Therefore, information about `mss`, `scale`, `olayout`, and some `quirks` are all collected by the P40f parser.

## 3.2 Computing P0f Signature Fields

In order to match a packet to an OS label, P40f collects information about p0f signature fields defined in Table 1 from each packet. In the switch, we maintain one metadata field for each p0f signature field; these metadata fields make up a struct called `p0f_metadata`. `p0f_metadata` is populated during parsing and in the match-action pipeline of the switch.

Most `p0f_metadata` fields can be extracted directly from the packet headers; for example, the *ver* field maps directly to the version field in a packet's IP header. However, the TCP window size (p0f signature field `wsize`) is hard to extract; this is because a `wsize` field value can be expressed as either a fixed integer or a multiple of the TCP maximum segment size (*e.g.*, `mss*2`). Switches typically cannot perform integer division, so we cannot simply divide the packet's TCP window size value by the MSS value.

To overcome this limitation, P40f performs a binary search instead over a range of reasonable values (0 to 64) in order to find an integer value of TCP window size divided by MSS, or `wsize_div_mss`. Performing binary search over this range requires six iterations at most. At each iteration, we maintain the invariant that `wsize_div_mss` falls between some lower and higher bound. If no value has been found after six iterations, TCP window size is not a multiple of MSS. In that case,

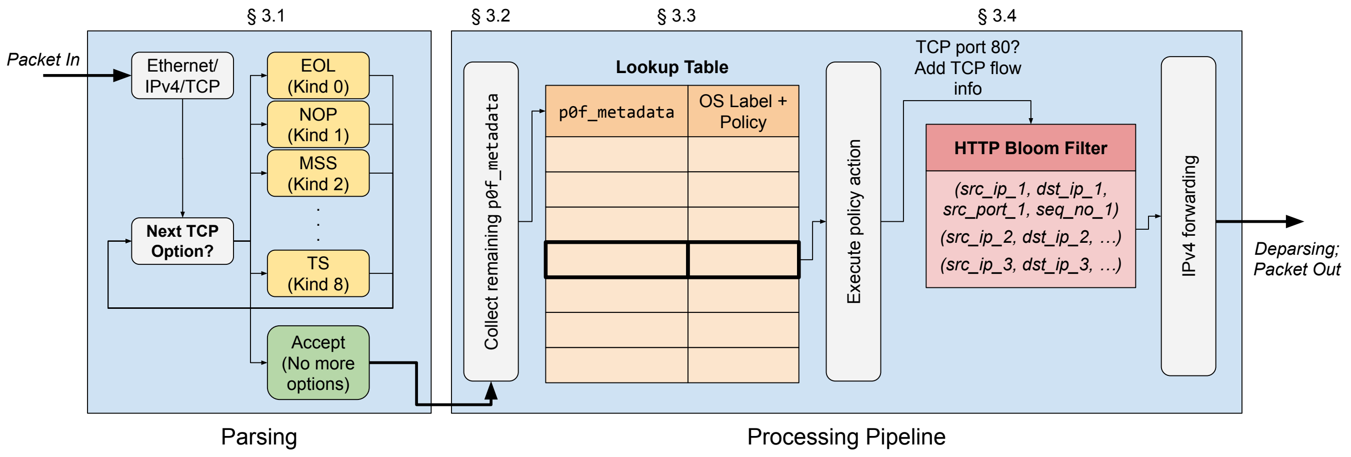

**Figure 3: Diagram summarizing P40f's processing of a SYN packet.** `p0f_metadata`, or match field information, is collected during TCP options parsing (Section 3.1) and at the beginning of the processing pipeline (Section 3.2). This metadata is then used as a key to look up the appropriate policy to execute (Section 3.3). If the SYN packet is a web packet and has been chosen to be sampled, then we add its TCP flow information to the HTTP Bloom filter, then send a clone of the packet to software for analysis (Section 3.4).

```
action binary_search_iter() {
    mid = (lo + hi) >> 1;
    mid_mss = (lo_mss + hi_mss) >> 1;
    // Compare wsize to midmss
    if (wsize < mid_mss) {
        hi = mid;
        hi_mss = mid_mss;
    } else if (wsize > mid_mss) {
        lo = mid;
        lo_mss = mid_mss;
    } else {  // wsize == mid  mss
        wsize_div_mss = mid;
    }
}
```

**Figure 4: One iteration of binary search for calculating `wsize_div_mss`.** We maintain the invariant that `wsize_div_mss` falls within the range $[lo, hi]$, which is initialized to $[0, 64]$. If no value has been found after six stages, `wsize` is not divisible by MSS.

wsize_div_mss is set to 0, so the packet cannot match any rule corresponding to a signature with wsize expressed as mss*n. Any signature with wsize expressed as a fixed integer translates to a rule with wildcard wsize_div_mss. Figure 4 shows the pseudocode for one iteration of binary search in P40f.

## 3.3 Inferring OS Label and Applying Policy

The match-action pipeline contains a lookup table containing fingerprinting rules. After collecting all `p0f_metadata` information, the switch uses `p0f_metadata` to look up an appropriate rule containing an OS label for and policy action to be executed on the packet.

Each rule contains nine *match fields*, which are used by the switch to match each packet to a particular rule. Each *match field* maps one-to-one to one of the fields in `p0f_metadata`, thus also maps to each p0f signature field in Table 1. The switch uses `p0f_metadata` as a key for the lookup table. A rule also contains three other components: an *OS label ID*, a policy *action*, and a *priority* value. If a packet matches multiple rules, the switch tie-breaks between the rules based on their *priority* values. The switch then increments a counter associated with the *OS label ID* of the matched rule, which is saved in the register memory. Such counters can be used to provide valuable information to operators such as the proportion of each particular OS that initiated a connection to the network. There are around 45 total OS labels in the p0f database and each counter is eight bytes wide in the current prototype, which totals to 360 bytes. This is much less than 1% of the total available on PISA-based hardware such as the Edge-core Wedge 100BF-32X with the Barefoot Tofino chip [11]. Finally, the switch executes the policy *action*.

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 for the `redirect` action in the policy file.

Fingerprinting rules are generated by a Python script that takes as input a p0f fingerprint database file and a security policies file. The script generates at least one rule for each p0f signature in the fingerprint database. The rule is then given an OS label ID based on the OS label for its corresponding p0f signature. If the OS label has been assigned a policy action in the policies file, the rule is given this action.

The format and and example of a policy in the policy file is as follows:

```
[target] -> [action name] [action parameters]
s:unix:Linux:3.11 and newer -> redirect 10.0.1.2
```

The *target* of the policy is the OS label or group of OS labels to which the policy applies. The *action name* is the type of action applied to this label. The *action parameters* are any parameters the policy action might take: for example, the `redirect` policy takes the IP address to which packets should be redirected as a parameter. In the above example, the policy states that all SYN packets that match Linux 3.11 and newer should be redirected to IP address `10.0.1.2`.

## 3.4 Exporting Data to Learn New Rules

Some SYN packets may not match any of the signatures in the p0f signature database, thus may not be identifiable in the switch running P40f. However, if the SYN packet forms part of an HTTP connection, then HTTP packets sent on this connection may contain a user-agent header field, which can be used to identify the host OS. Even if the switch is able to match such a SYN packet to an OS label, we can use HTTP user-agent fields to verify that the OS label is correct.

P40f captures unidentifiable SYN packets that form part of a web connection (destination port 80), as well as the first data packet observed on this connection, and forwards copies of these packets to the software for analysis. P40f forwards the first data packet observed on a web connection to the software because this packet is likely to be an HTTP request; therefore, the packet's HTTP header may contain a user-agent field that can identify the OS running on the sender. P40f also samples a small proportion (*e.g.*, 10%) of SYN packets that can be labeled by the switch and sends copies of these packets and their corresponding first data packets to the software as well. P40f still allows the TCP handshake to proceed by forwarding the original packet to its intended destination.

When sending a SYN packet to the software, P40f records a three-tuple (source IP address, destination IP address, source port) of the packet and its sequence number. Any subsequent web packets with the same tuple and with sequence number incremented by one are forwarded to the software as well. Any such packet is likely to be an HTTP request with a user-agent header field since TCP port 80 is associated with HTTP by convention. However, non-HTTP connections might use TCP port 80 as well. Thus, the software checks if a received packet is HTTP or not.

P40f needs to keep track of (source IP address, destination IP address, source port) tuples and sequence numbers to forward subsequent packets to the software. However, the amount of memory for the data structure might increase and overwhelm the hardware switch. We resolve this issue by using a Bloom filter, which allows P40f to test if an element is in a set using a small, fixed amount of memory. The four values *source IP adddress*, *destination IP address*, *source port*, and *sequence number + 1* are extracted from each SYN packet and added to the Bloom filter as a single element. For every TCP packet that passes through the switch, this set of four values is checked against the Bloom filter. If the Bloom filter contains the set, the switch clones the packet and forwards the clone to the software. Our Bloom filter implementation is based on a well-known $P4_{14}$ implementation of a counting Bloom filter found in the SIGCOMM 2016 P4 tutorials [8].

Such resource-effective Bloom filter comes with a cost; lookups to the Bloom filter can result in false positives. This means that the switch may forward data packets to the software even if they do not correspond to previously-seen unmatched SYN packets. It is the responsibility of the software to ignore non-SYN packets that do not correspond to previously-received SYN packets. Fortunately, Bloom filter lookups never result in false negatives.

## 4 EVALUATION OF PROTOTYPE

The P4 component of the P40f prototype consists of 1,037 lines of code. In Section 4.1, we describe the resource footprint of the prototype and show that P40f can be run in parallel with other P4 applications. In Section 4.2, we describe validation of the P40f prototype against p0f.

### 4.1 Resource Usage

**Number of stages.** When compiled using the p4c compiler for the BMv2 software switch [35], the processing pipeline of the P4 code consists of 15 tables. Figure 5 shows the tables used by P40f as well as the packet flow through the tables. Seven tables are used for obtaining all p0f signature information, including the six tables for binary searching for `wsize_div_mss`. One table is required for the p0f signature lookup table. Five tables are required for deciding and sending a packet to the software that does fingerprinting based on HTTP user agent. This includes sampling, adding to HTTP Bloom filter, lookup, and cloning packets. Lastly, two tables are required for IPv4 forwarding. One table queries a

Bloom filter to check if the source IP address has previously been given an OS label for which the `drop_ip` policy applies. If so, the packet must be dropped. Another table performs a longest-prefix-match to determine the next interface on which to forward.

A programmable hardware switch typically contains 10-20 stages, and each stage has around 10-20 parallel tables. P40f's P4 program has 15 sequential tables, thus will fit in a hardware switch that has 15 or more stages. Also, while P40f requires 15 stages, each stage does not use many resources. Given that hardware switches can execute multiple independent tables in parallel in a stage, smarter P4 compilers from production-grade Software Development Environment (*e.g.*, Barefoot P4Studio) will easily allow other P4 applications to run in parallel with P40f.

**Register memory space.** P40f prototype uses a constant amount of memory in the form of registers and packet metadata. The memory required for OS counter registers is negligible because the number of OS labels in the fingerprint database is unlikely to be larger than in the order of hundreds: the p0f v3.09b database contains just 45 OS labels. Similarly, the Bloom filters used for lookup of TCP connection information and IP address lookup require only a small, fixed amount of stateful memory. Packet metadata also requires only a small amount of memory.

## 4.2 Validation

We validated that both the P4 program and the rule generator script are correct by generating test packets using a Python script. The script produces at least one packet for each signature in the p0f v3 database. We compiled P40f with the p4c compiler and installed the P4 program on the BMv2 software switch [34]. We then installed the generated rules onto the switch using the P4 Runtime API. We then sent the test packets through the switch. We also used p0f to identify these same test packets. We verified that the OS labels assigned to the packets by the switch and by p0f were the same for each test packet. The only packets for which P40f and p0f assigned different labels were those that P40f labeled `s:!:NMap:OS detection`. We believe that this discrepancy is due to a bug in p0f related to detecting the `ack+` quirk, in which the ACK number in the TCP header is non-zero, but the ACK flag is not set.

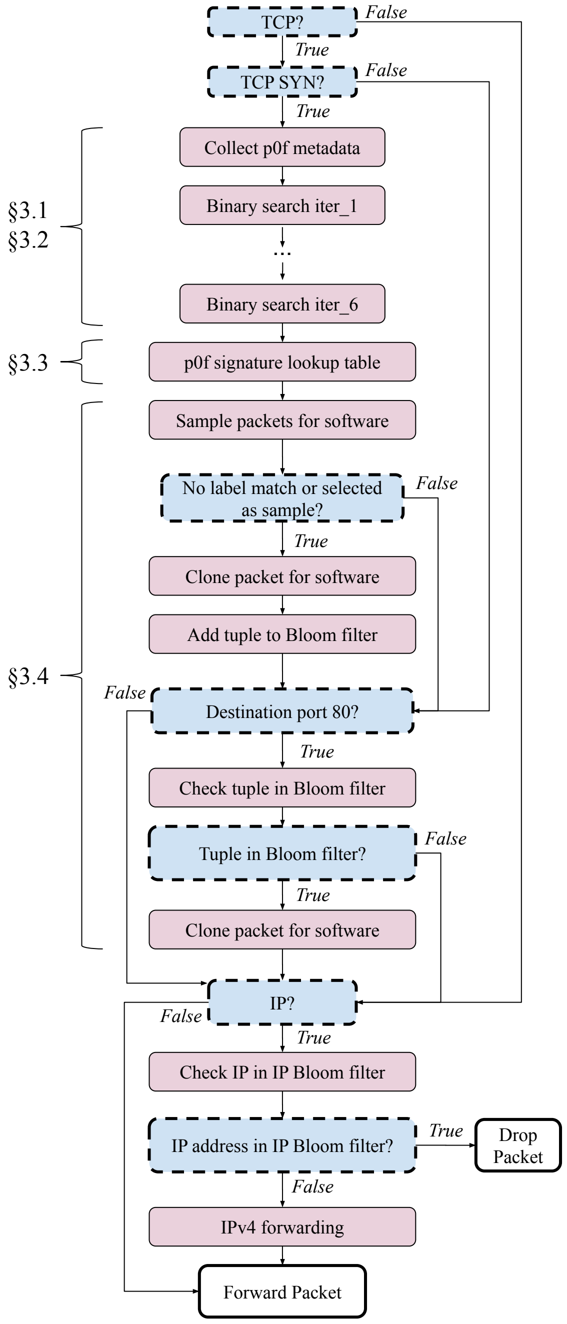
TCP?
False
True
TCP SYN?
False
True
Collect p0f metadata
Binary search iter_1
...
Binary search iter_6
§3.1
§3.2
§3.3
p0f signature lookup table
Sample packets for software
No label match or selected as sample?
False
True
Clone packet for software
Add tuple to Bloom filter
§3.4
False
Destination port 80?
True
Check tuple in Bloom filter
Tuple in Bloom filter?
False
True
Clone packet for software
IP?
False
True
Check IP in IP Bloom filter
IP address in IP Bloom filter?
True
Drop Packet
False
IPv4 forwarding
Forward Packet

**Figure 5: P40f Stage resource usage.** Each red box with straight lines corresponds to a single table. Each blue box with dotted lines corresponds to a conditional branch.

## 5 CAMPUS MEASUREMENT STUDY

In this section, we present a measurement study on a campus network using our prototype of P40f. Figure 6 shows the diagram of the edge of the campus network and indicates where packet traces were captured. The purpose of this study is to demonstrate the benefit to network administrators by having the capability to fingerprint host OS directly in the data plane.

For privacy reasons, evaluation scripts were run on packet data by a campus network administrator working in the university's office of information technology department. No

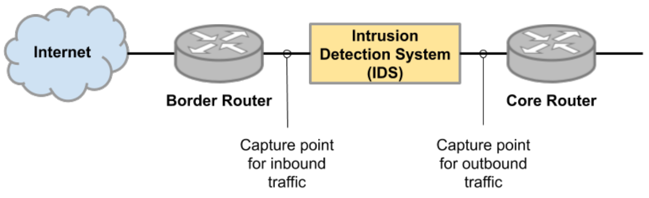
Internet
Border Router
Intrusion Detection System (IDS)
Core Router
Capture point for inbound traffic
Capture point for outbound traffic

**Figure 6: The edge of the university campus network.** Inbound traffic was captured between the border router and the IDS, while outbound traffic was captured between the core router and the IDS.

other individuals were able to view personally-identifiable information such as unanonymized IP addresses or packet payloads. All evaluation script outputs were checked and verified to be free of such information before further analysis. To mitigate risk, all of our data collection and evaluation processes were reviewed and approved by our institution's review board.

In Section 5.1, we discuss the presence of TCP options in the SYN packets observed in these traces: we observe that a lack of TCP options may be indicative of an adversarial sender. In Section 5.2, we use P40f's OS fingerprinting to characterize operating systems running on external and internal hosts of the campus network. In Section 5.3, we use both incoming and outgoing packet traces to analyze the user-agent field found in HTTP packets originated from outside and inside the network, respectively.

## 5.1 SYN Packets With No TCP Options

As discussed in Section 2, both the ordering and contents of TCP options are critical features of a p0f signature. For example, we observe that no TCP SYN signature in the p0f v3.09b fingerprint database contains an empty *olayout* field. Thus, a packet without TCP options cannot be labeled by P40f. For both incoming and outgoing packet traces, we track the number of SYN packets with and without TCP options. We also track how many of these SYN packets saw a corresponding ACK packet. To do this, we count the number of SYN packets for which we observed a subsequent ACK packet with the same three-tuple (source IP address, destination IP address, source port) and with a sequence number one greater than the sequence number of the SYN.

In a one-minute capture of incoming packets (from Internet to campus), we observed 286,215 SYN packets, and 72% of them (220,991) contained no TCP options. We observed that 17,226 SYN packets saw corresponding ACK packets, but among them, only six packets (0.0348%) contained no TCP options. In a one-minute capture of outgoing packets, we observed 105,288 SYN packets, and two of them (0.00190%) contained no TCP options. 87,087 SYN packets saw ACK packets, and among these packets, only two packets (0.00230%) contained no TCP options.

We find that while the proportion of outgoing SYNs with no TCP options is very small (0.00190%), the majority of incoming SYNs (77.2%) contain no TCP options. This suggests that P40f's ability to fingerprint external SYN packets is limited. However, we also observe that among incoming SYN packets that see a following ACK, the proportion of packets with no TCP options is small (0.0348%). This suggests that while most incoming SYN packets contain no TCP options, most of these SYN packets are not sent to establish a TCP connection; rather, they may be the result of adversarial behavior such as port scanning. Thus, while P40f may not be able to perform fingerprinting on packets with no TCP options, an absence of TCP options is itself an indication that the sender may be adversarial.

Since over 99% of outgoing SYN packets contain TCP options, P40f is more suitable for fingerprinting hosts that are present in the network. As network administrators are more interested in tracking information of internal hosts in their network than external hosts, P40f still promises great benefit for managing an enterprise network.

## 5.2 Operating Systems Running on Hosts

In this section, we use P40f's OS fingerprinting to characterize external and internal hosts of the network. To characterize external hosts, we run P40f on SYN packets in the one-minute incoming packet trace. To characterize internal hosts, we run P40f on SYN packets in the one-minute outgoing packet trace.

We used Tcpreplay [42] to replay TCP packets with only the SYN flag set (`tcp[13] == 2`) through the BMv2 software switch running P40f. There are 284,594 such packets in the incoming trace and 100,297 such packets in the outgoing trace. We note that these counts are slightly smaller than the numbers of SYN packets reported in Section 5.1 (286,215 and 105,288 packets). This is because the TCPreplay filter excluded packets with flags other than the SYN flag (e.g., PSH or URG) set. While non-SYN flags correspond to the p0f signature quirks `urgf+`, `pushf+`, and `ack-`, we observe that no signature in the p0f v3.09b database contains any of these quirks. Therefore, any packet with non-SYN flags set would not have been successfully matched to an OS label, so the characterization results are not affected.

In the incoming packet trace, 242,936 SYN packets (85.4%) could not be labeled. Since 77.2% of SYN packets in the incoming trace do not contain TCP options (Section 5.1), such a high rate of unlabeled packets is not unexpected. On the other hand, only 1,113 SYN packets (1.11%) could not be labeled in the outgoing packet trace.

| OS Label | Inbound (%) | Outbound (%) |
|---|---|---|
| **Linux** | **52.01%** | **23.15%** |
| 2.2.x-3.x, generic | 31.39% | 13.98% |
| 3.11 and newer | 10.83% | 7.44% |
| 3.1-3.10 | 6.97% | 0.88% |
| 3.x | 1.04% | 0.58% |
| 2.4.x | 1.03% | 0.02% |
| 2.6.x | 0.59% | 0.25% |
| 2.4.x-2.6.x, generic | 0.14% | 0.00% |
| Android | 0.02% | 0.00% |
| **Windows** | **40.91%** | **34.99%** |
| 7 or 8 | 28.82% | 11.50% |
| NT kernel, generic | 8.96% | 18.13% |
| XP | 2.60% | 0.05% |
| NT kernel 5.x | 0.51% | 5.30% |
| NT kernel 6.x | 0.02% | 0.00% |
| **Mac OS** | **6.46%** | **41.79%** |
| OS X, generic | 6.23% | 41.46% |
| OS X 10.9 or newer | 0.12% | 0.09% |
| OS X 10.x | 0.10% | 0.25% |
| **Other** | **0.62%** | **0.06%** |
| FreeBSD 9.x or newer | 0.24% | 0.04% |
| FreeBSD, generic | 0.21% | 0.01% |
| Solaris 10 | 0.13% | 0.02% |
| FreeBSD 8.x | 0.04% | 0.00% |

**Table 2: SYN packets matching each OS label in the one-minute packet trace of incoming and outgoing campus traffic.** Percentages are given out of the number of successfully-labeled SYN packets.

Table 2 summarizes the distribution of OS labels for all the SYN packets that can be labeled and marked by P40f. While only 6.46% of incoming SYNs were found to be sent from hosts running Mac OS, 41.8% of outgoing SYNs were labeled as Mac OS. The prominence of Mac OS in the outgoing trace reflects that the campus network consists mostly of client devices employed for personal use: Mac OS X runs primarily on Apple laptops, desktops, and smartphones. This result also corroborates conventional wisdom and reports suggesting that the proportion of academic clients who use Macintosh devices is greater than that of the general population [22, 27].

In contrast, more than half (52.0%) of incoming connections are initiated by Linux hosts, compared to just 23.1% of outgoing connections. One of the most accessed sites on this campus network is a mirror site for Linux ISO images and software packages. A large proportion of hosts accessing this site are presumably Linux servers.

## 5.3 Comparison With HTTP User-Agent

For both incoming and outgoing packet traces, we tracked each TCP connection to destination port 80 for which the SYN was observed in the packet trace. If the first data packet sent on the connection was an HTTP packet, we extracted the user-agent field for examination. There were 21,058 such packets in the incoming packet trace and 59,438 such packets in the outgoing packet trace. We parse a p0f OS label from each user-agent string in the same way that the database-updating software parses user-agent fields (Section 3.4).

| OS Label Group | Incoming | Outgoing |
|---|---|---|
| Windows | 80.21% | 37.86% |
| Linux | 6.56% | 9.81% |
| iOS | 5.15% | 12.17% |
| Android | 4.13% | 13.33% |
| Mac OS | 3.95% | 26.84% |

**Table 3: HTTP user-agent in the incoming and outgoing packet traces that match each OS label group.** Percentages are given out of all successfully-labeled user-agents for each packet trace.

Table 3 shows the percentage of HTTP user-agents matching each OS category for both incoming and outgoing traces. While we could not parse an OS label from approximately 73.1% of user agents in the incoming packet trace and 67.2% of user-agents in the outgoing packet trace, we argue that only one parsable user agent is needed from a given host in order to generate a new p0f signature for that host's OS. Because the user agent is dependent on the application rather than the operating system, HTTP traffic from a single host may contain many different user agents.

For each HTTP packet examined, we also extracted information about p0f signature fields from the corresponding SYN, similar to how the P4 program obtains `p0f_metadata` (Section 3.2). We used this data to build new TCP packets using ScaPy [40], then played these packets through the BMv2 switch with the P40f installed. To evaluate how well P40f identifies OS labels by just using TCP options, we compare the result of P40f's OS fingerprinting on each generated SYN packet to the user-agent in the corresponding HTTP packet. Specifically, each OS label produced by P40f and each HTTP user-agent was mapped to an OS label group. We then counted the number of instances of each (P40f OS label group, user-agent OS label group) pair. Table 5 shows the number of instances for each pair in the incoming packet trace, and Table 4 shows the number of instances for each pair in the outgoing packet trace. We report only the pairs for which the user-agent could be parsed.

| P40f OS | User-Agent OS Label | | | | |
|---|---|---|---|---|---|
| Label | Windows | Linux | Mac OS | Android | iOS |
| Windows | 7,296 | 133 | 0 | 0 | 0 |
| Linux | 0 | 1,750 | 2 | 2,588 | 24 |
| Mac OS | 0 | 1 | 135 | 0 | 145 |
| FreeBSD | 2 | 0 | 0 | 0 | 0 |
| Unknown | 54 | 16 | 65 | 3 | 9 |

**Table 4: Number of instances observed in the outgoing trace for each P40f OS label group and user-agent OS label group pair.**

| P40f OS | User-Agent OS Label | | | | |
|---|---|---|---|---|---|
| Label | Windows | Linux | Mac OS | Android | iOS |
| Windows | 2,729 | 181 | 5 | 0 | 45 |
| Linux | 1,810 | 183 | 84 | 234 | 105 |
| Mac OS | 0 | 1 | 135 | 0 | 145 |
| FreeBSD | 4 | 1 | 2 | 0 | 1 |
| Unknown | 82 | 12 | 6 | 4 | 1 |

**Table 5: Number of instances observed in the incoming trace for each P40f OS label group and user-agent OS label group pair.**

For the outgoing packet trace, there were very few instances in which the OS label reported by P40f and the OS label reported by the user-agent mismatch. For example, only 133 out of 7,429 (1.79%) packets labeled Windows by P40f corresponded to user-agents that reported Linux, and no user-agent reported an OS other than Windows or Linux. However, the incoming trace contains many more instances in which the P40f OS label and HTTP user-agent OS label completely differ. In particular, only 413 out of 2,416 (19.4%) packets labeled by P40f as Linux corresponded to user-agents reporting Linux or Android. Instead, for 1,810 out of 2,416 (74.9%) such packets, the user-agent reported Windows.

We further investigate this case using Table 6. Table 6 shows counts for pairs of P40f and user-agent OS labels for the 1,810 packets labeled "Linux" by P40f but "Windows" by the HTTP user-agent. We observe that only 392 packets (21.7%) were labeled by P40f with a "generic"-type, which is a last-resort OS label. P40f assigns a narrower, non-"generic" label to each of the remaining 1,418 packets (78.3%), so P40f is able to label about 78.3% of packets with relatively high confidence. Thus, it is unlikely that P40f simply performs poorly on traffic from external hosts. Rather, these hosts are likely to be spoofing their user-agents or modifying TCP/IP headers in order to appear to be running a different OS. Some evidence suggests that most malicious HTTP traffic involves the use of incorrect or suspicious user-agents [39]. We observe that 1,110 out of 1,810 packets (61.3%) are classified as "Linux 3.1-3.10" by P40f but as "Windows XP" by the HTTP user-agent. Of these 1,110 packets, 1,103 reported the exact user-agent string "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1." This suggests that a bot running Linux 3.1-3.10 but claiming to run Windows XP could be responsible for the majority of mismatched packets.

| P40f OS Label | User-Agent OS Label | Count |
|---|---|---|
| Linux 3.11 and newer | Windows 10 | 96 |
| | Windows 8.1 | 3 |
| | Windows 8 | 4 |
| | Windows 7 | 54 |
| | Windows XP | 105 |
| | Windows, generic | 1 |
| Linux 3.1-3.10 | Windows 10 | 9 |
| | Windows 8.1 | 1 |
| | Windows 7 | 16 |
| | Windows XP | 1110 |
| Linux 2.4.x | Windows 10 | 9 |
| | Windows 7 | 10 |
| Linux 3.x, generic | Windows 10 | 21 |
| | Windows 7 | 38 |
| Linux 2.2.x-3.x, generic | Windows 10 | 32 |
| | Windows 8.1 | 3 |
| | Windows 8 | 3 |
| | Windows 7 | 33 |
| | Windows XP | 232 |
| Linux 2.2.x-3.x (no timestamps), generic | Windows 10 | 3 |
| | Windows 7 | 14 |
| | Windows XP | 13 |

**Table 6: Number of packets observed for each Linux P40f OS label and Windows user-agent OS label pair.** An OS label starting with *g:* is a "generic"-type or last-resort label; an OS label starting with *s:* is a non-"generic" label.

## 5.4 Using P40f For Management Tasks

P40f provides valuable information about hosts inside or outside a network. In this section, we describe how a network administrator can interact with P40f and use it for real network management tasks.

**Lowering security risks for internal hosts.** It is common that a security vulnerability is tied to a specific operating system or a specific version of an operating system. Also, most, if not all, operating systems have an end-of-life, after which no new updates (including essential security patches) are provided. When a new OS-specific security vulnerability is found or the end-of-life of an operating system is announced, it is the network operator's job to assess the security risk in the network. Such management tasks are becoming more important as bring-your-own-device (BYOD) networks, where network administrators have less control over client devices, become more prevalent (*e.g.*, eduroam on campuses [13]). P40f can help the network operator handle such tasks.

For example, if a new security vulnerability is announced for the Apple Mac OS X (*e.g.*, CVE-2018-4243 [32], CVE-2018-4249 [33]), the operator can check with the OS statistics reported by P40f and immediately know that at least 41.79% of connections that comes from internal hosts are at risk (Table 2). Linux 2.4 reached its end-of-life on April 9, 2012 [23] and Microsoft Windows XP reached its end-of-life on April 8, 2014 [28]. An operator can detect that 0.02% of connections are from hosts running Linux 2.4 and 0.05% of connections are from hosts still running Windows XP in the campus network (Table 2, outbound). An operator can use the P4 Runtime API [36] to extract such information from the switch's match-action table statistics or from the register memory.

As these internal hosts are at higher risk, a network administrator may want to contact the owner of an vulnerable host. To do this, the network administrator can set a security policy in P40f that applies the `redirect` action on packets that match the `p0f_metadata` signatures of the operating systems in question (Section 3.3). P40f will then mirror matched packets and send them to the specified output port, which is connected to a collector that performs further analysis. The collector can extract host-specific information such as source and destination IP addresses, MAC addresses, timestamp, TCP/UDP port numbers, and so on. By joining this information with other sources of information such as authentication (login credentials) and DHCP (MAC and IP addresses) logs maintained by the network, it is possible to pinpoint the user and the location of the host at risk. For example, in case of eduroam, once the operator has a list of IP addresses that are identified to have a vulnerable OS type by P40f, she can join that information with the 802.1X authentication logs and DHCP logs from wireless access point controllers to retrieve the list of email addresses of the users who own the client devices. The operator can contact each user of a vulnerable host to gain more knowledge and lower the security risk in the network.

**Tightening security against external hosts.** P40f can be used to tighten a network's security against external hosts and the wider Internet. In Section 5.1, we observed that the majority of incoming SYNs (77.2%) contain no TCP options, but among incoming SYN packets that see a following ACK, the proportion of SYN packets with no TCP options is very small (0.0348%). This suggests that most of such SYN packets with no TCP options represent adversarial behavior such as port scanning or SYN flooding, which in many cases have no TCP options in packets. A network operator can use P40f to identify and even block such adversarial behavior in the data plane itself by taking corrective action against SYN packets with no TCP options.

The mismatch of the OS label when using the HTTP user-agent versus the TCP SYN option parsing is one of the key indicators for detecting malicious behavior with a spoofed OS [18, 24, 31]. It is easier to forge the HTTP user-agent field, which can be done with a simple browser extension, compared to changing the values in the TCP header. For example, based on Table 6, we suspect a malicious external Linux host is claiming to run Windows XP and sends SYN packets to the campus to find web servers that allow communicating with Windows XP clients. We hypothesize that this Linux client is trying to avoid being blocked by a web server by masquerading itself as a Windows client, or to check if the web server implements good security practices such as disallowing clients that has an outdated OS in their user-agent field. In Section 5.3, we observed that this mismatch rate is much higher with incoming traffic, *i.e.*, external hosts trying to connect to the campus' internal web servers, compared to the outgoing traffic. A network operator can use P40f to detect such malicious behavior towards the network by comparing the host OS label reported by P40f and another system with HTTP user-agent information. For example, by joining the OS label and the client IP address reported by P40f with the user-agent and the client IP address logged by the web server, the operator can detect a mismatch and also pinpoint the suspicious client by its IP address.

## 6 RELATED WORK

**TCP options parsing in P4**. Dapper [16] also implements fine-grained TCP options parsing in P4. However, because Dapper is written in $P4_{14}$, this parsing loop causes the exact deparsing behavior to be undefined. Thus, Dapper must explicitly define and enforce an exact order of options for every deparsed packet. On the other hand, P40f is written in $P4_{16}$, which supports the header stack type. A header stack can store parsed options in their original order. This allows packets to pass through the switch unmodified, leaving no marks or unnecessary modification to the original packet after OS fingerprinting has occurred.

**Applications that use p0f**. A number of existing software applications use p0f for OS-based packet filtering. The network firewall pfSense [30] can use p0f to filter connections initiated within the network by OS [29]. A modification [14, 15] to the OpenBSD Stateful Packet Filter (pf) [19] also allows pf to use p0f signatures to filter SYN packets by OS. Because pfSense and pf are both implemented as a tool that runs on x86 systems, OS fingerprinting and filtering packets involve the overhead of traversing the network stack. Such process is expensive and limits overall network throughput. P40f, on the other hand, is able to block or modify packets based on OS directly in the data plane, allowing OS fingerprinting and packet filtering to be performed at line rate.

**Other approaches to TCP/IP fingerprinting**. ZMap [10] is an active OS fingerprinting software tool optimized for scanning the entire IPv4 address space. ZMap is able to achieve short scan times through fast sending of probes, no storing of per-connection state, and no retransmission. ZMap was developed for large-scale networks. It is less suitable for BYOD networks because active fingerprinters may not be able to detect devices that connect and disconnect between scans. Furthermore, each scan places additional load on the network in the form of probes.

Some prior work has focused on statistical rather than signature match-based approaches to TCP/IP fingerprinting. Beverly [5] uses a naïve Bayesian classifier to identify the operating system of a host based on features such as TTL, window size, SYN size, and DF (don't fragment). Beverly finds that the Bayesian classifier is able to make an identification even when data from packet headers is ambiguous. Hershel [41] is another probabilistic classification method for single-packet passive OS fingerprinting that uses features derived from TCP/IP header information. However, both Bayesian classifier and Hershel involve a large amount of computational overhead, which prevents their use in network monitoring settings that require OS fingerprinting and packet filtering in real-time. P40f can perform OS fingerprinting on ongoing network traffic stream with low overhead and without impacting network performance.

**Non-TCP/IP Approaches**. The use of DHCP options by a DHCP client is specific to the client vendor, operating system, and device. This means that OS fingerprinting can also be performed by inspecting DHCP options in packets involved in a DHCP exchange. ArubaOS [1] is a network operating system for Aruba Networks' wireless local area network (WLAN) controllers. ArubaOS's DHCP fingerprinting feature can be enabled when the controller is placed in the path between the DHCP client and server. Administrators can also set access control policies based on the fingerprinted OS. Fingerbank [20] is a database containing identification information for a variety of operating systems and devices. Each device entry contains a DHCP fingerprint, MAC address information, and/or an HTTP user agent. Users can query the database API using one or more of these attributes to produce a device match. PacketFence [21] is a software tool for network access control that uses Fingerbank to perform OS and device identification. Kollmann describes how DHCP-based OS fingerprinting can be done in practice [25]. However, DHCP fingerprinting can only be used against DHCP clients. Thus, DHCP-based fingerprinting cannot be used in the absence of a DHCP server on the network or against clients that do not use the DHCP server that the network operator controls. In contrast, P40f performs OS fingerprinting on TCP packets, which make up a substantial amount of both incoming and outgoing traffic on most networks. P40f can thus be used to perform OS fingerprinting in settings and on hosts for which DHCP fingerprinting is not applicable or available.

## 7 CONCLUSION AND FUTURE WORK

We present P40f, a 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 at 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 used the prototype to characterize both incoming and outgoing traffic from a real campus network.

We plan to improve and extend P40f as we aim to deploy it in real networks.

**Run in real hardware**. P40f's P4 program can currently run on the BMv2 software switch [34], a software simulator for a P4-programmable switch. We are currently working on compiling the code with the compiler for the Barefoot Tofino hardware switch [3]. We plan to deploy P40f on a Barefoot Tofino switch and run against mirrored traffic from a real campus production network in real-time.

**More built-in actions.** More sophisticated policies may involve rate-limiting traffic sent from hosts associated with undesirable OSes. Rate-limiting can act as an incentive for users of outdated or vulnerable operating systems to upgrade or change their OSes. Rate-limiting may also help reduce the impact of certain security attacks by vulnerable OSes, such as SYN floods or other denial-of-service attacks. We can implement a token bucket in the data plane by maintaining registers that hold the state of the token bucket at any given time. This state can include the *timestamp* associated with the token bucket state and the *number of tokens* in the bucket at this timestamp. The number of tokens can be updated accordingly by packets from undesirable OSes.

## REFERENCES

[1] Aruba Networks. 2019. ArubaOS network operating system. https://www.arubanetworks.com/products/networking/arubaos/.
[2] Patrice Auffret. 2010. SinFP, unification of active and passive operating system fingerprinting. *Journal in Computer Virology* 6, 3 (01 Aug 2010),

197–205. https://doi.org/10.1007/s11416-008-0107-z

[3] Barefoot Networks. 2019. Barefoot Tofino. https://www.barefootnetworks.com/technology/.

[4] Jason Barnes and Patrick Crowley. 2013. K-p0F: A High-throughput Kernel Passive OS Fingerprinter. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '13)*. IEEE Press, Piscataway, NJ, USA, 113–114. http://dl.acm.org/citation.cfm?id=2537857.2537875

[5] Robert Beverly. 2004. A Robust Classifier for Passive TCP/IP Fingerprinting. In *Passive and Active Network Measurement*, Chadi Barakat and Ian Pratt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 158–167.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Computer Communication Review* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[7] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM '13)*. ACM, New York, NY, USA, 99–110. https://doi.org/10.1145/2486001.2486011

[8] Sean Choi. 2016. heavy_hitter. https://github.com/p4lang/tutorials/blob/846f059ddd9c53157ea9cc2ec7c0b2d5359f2df0/SIGCOMM_2016/heavy_hitter/p4src/heavy_hitter.p4.

[9] Catalin Cimpanu. 2019. Georgia county pays a whopping $400,000 to get rid of a ransomware infection. https://www.zdnet.com/article/georgia-county-pays-a-whopping-400000-to-get-rid-of-a-ransomware-infection/.

[10] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2013. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *Proceedings of the USENIX Conference on Security (SEC'13)*. USENIX Association, Berkeley, CA, USA, 605–620. http://dl.acm.org/citation.cfm?id=2534766.2534818

[11] EdgeCore. 2019. Edge-core Wedge 100BF-32X with Barefoot Tofino Chip[Online]. https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335.

[12] Andy Fingerhut. 2017. tcp-options-parser2. https://github.com/jafingerhut/p4-guide/blob/f2b3fc6d02ddd0acec6afd7a38543c040defcb0b/tcp-options-parser/tcp-options-parser2.p4.

[13] Licia Florio and Klaas Wierenga. 2005. Eduroam, providing mobility for roaming users. In *Proceedings of the EUNIS Conference*. https://www.terena.org/activities/tf-mobility/docs/ppt/eunis-eduroamfinal-LF.pdf

[14] Mike Frantzen. 2003. PF filter decisions based on source OS type. https://groups.google.com/d/msg/bit.listserv.openbsd-pf/_cCxtG06YC4/_fXS5WsQynIJ. Post to Usenet group bit.listserv.openbsd-pf.

[15] Mike Frantzen. 2003-16. pf.os. https://github.com/openbsd/src/blob/3395fe1155e42ac7e12ee105544ca6877801749f/etc/pf.os.

[16] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. 2017. Dapper: Data Plane Performance Diagnosis of TCP. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. ACM, New York, NY, USA, 61–74. https://doi.org/10.1145/3050220.3050228

[17] Lloyd G. Greenwald and Tavaris J. Thomas. 2007. Toward Undetected Operating System Fingerprinting. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT '07)*. USENIX Association, Berkeley, CA, USA, Article 6, 10 pages. http://dl.acm.org/citation.cfm?id=1323276.1323282

[18] Martin Grill and Martin Rehák. 2014. Malware detection using http user-agent discrepancy identification. In *2014 IEEE International Workshop on Information Forensics and Security (WIFS)*. IEEE, 221–226.

[19] Daniel Hartmeier. 2002. Design and Performance of the OpenBSD Stateful Packet Filter (Pf). In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 171–180. http://dl.acm.org/citation.cfm?id=647056.713848

[20] Inverse. 2019. Fingerbank. https://fingerbank.org/.

[21] Inverse. 2019. PacketFence. https://packetfence.org/.

[22] jamf. 2016. 2016 Survey: Managing Apple Devices in Higher Education. https://www.jamf.com/resources/e-books/2016-survey-managing-apple-devices-in-higher-education/.

[23] Sean Michael Kerner. 2012. Linux 2.4 Hits the End of the Line. http://www.internetnews.com/blog/skerner/linux-2.4-hits-the-end-of-the-line.html.

[24] Amit Klein. 2012. How Fraudsters are disguising PCs to fool device fingerprinting. https://securityintelligence.com/how-fraudsters-are-disguising-pcs-to-fool-device-fingerprinting/. (2012).

[25] Eric Kollmann. 2007. Chatter on the Wire: A look at DHCP traffic. http://chatteronthewire.org/download/chatter-dhcp.pdf. (2007).

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

[27] Leila Meyer. 2015. Report: Apple Device Adoption Increasing in Higher Education. https://campustechnology.com/articles/2015/12/09/report-apple-device-adoption-increasing-in-higher-education.aspx.

[28] Microsoft. 2014. Support for Windows XP ended. https://www.microsoft.com/en-us/windowsforbusiness/end-of-xp-support.

[29] Netgate. 2019. List of pfSense Features. http://web.archive.org/web/20190203121739/https://www.pfsense.org/about-pfsense/features.html.

[30] Netgate. 2019. pfSense. https://www.pfsense.org/.

[31] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2013. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 541–555.

[32] NIST. 2018. Apple Vulnerability: CVE-2018-4243. https://nvd.nist.gov/vuln/detail/CVE-2018-4243.

[33] NIST. 2018. Apple Vulnerability: CVE-2018-4249. https://nvd.nist.gov/vuln/detail/CVE-2018-4249.

[34] p4lang. 2019. Behavioral Model Repository. https://github.com/p4lang/behavioral-model.

[35] p4lang. 2019. P4_16 Prototype Compiler. https://github.com/p4lang/p4c.

[36] P4 Language Consortium. 2017. P4 Runtime. https://p4.org/p4-runtime/.

[37] J. Postel. 1981. *Transmission Control Protocol*. RFC 793. RFC Editor. http://www.rfc-editor.org/rfc/rfc793.txt

[38] Anirudh Ramachandran and Nick Feamster. 2006. Understanding the Network-level Behavior of Spammers. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM '06)*. ACM, New York, NY, USA, 291–302. https://doi.org/10.1145/1159913.1159947

[39] Christian Rossow, Christian J. Dietrich, Herbert Bos, Lorenzo Cavallaro, Maarten van Steen, Felix C. Freiling, and Norbert Pohlmann. 2011. Sandnet: Network Traffic Analysis of Malicious Software. In *Proceedings of the Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS '11)*. ACM, New York, NY, USA, 78–88. https://doi.org/10.1145/1978672.1978682

[40] ScaPy. 2019. ScaPy: Packet crafting for Python2 and Python3. https://scapy.net.

[41] Zain Shamsi, Ankur Nandwani, Derek Leonard, and Dmitri Loguinov. 2016. Hershel: Single-Packet OS Fingerprinting. *IEEE/ACM*

Sherry Bai, Hyojoon Kim, Jennifer Rexford
Princeton University

*Transactions on Networking* 24, 4 (Aug 2016), 2196–2209. https://doi.org/10.1109/TNET.2015.2447492

[42] TcpReplay. 2019. Tcpreplay: Pcap editing and replaying utilities. http://tcpreplay.appneta.com.

[43] Michał Zalewski. 2012. p0f v3: passive fingerprinter. http://lcamtuf.coredump.cx/p0f3/README.

[44] Michał Zalewski. 2014. p0f v3 (version 3.09b). http://lcamtuf.coredump.cx/p0f3/.