A Robust Classifier for Passive TCP/IP Fingerprinting

Robert Beverly

MIT Computer Science and Artificial Intelligence Laboratory
rbeverly@csail.mit.edu

Abstract. Using probabilistic learning, we develop a naive Bayesian classifier to passively infer a host's operating system from packet headers. We analyze traffic captured from an Internet exchange point and compare our classifier to rule-based inference tools. While the host operating system distribution is heavily skewed, we find operating systems that constitute a small fraction of the host count contribute a majority of total traffic. Finally as an application of our classifier, we count the number of hosts masquerading behind NAT devices and evaluate our results against prior techniques. We find a host count inflation factor due to NAT of approximately 9% in our traces.

1 Introduction

Previous measurement studies analyzing Internet packet header traces demonstrate the wealth of information, for instance traffic characteristics and host behavior, available to a passive monitor. In this work we use simple probabilistic learning methods to perform a maximum-likelihood inference of a host's operating system from packet headers. Drawing upon earlier TCP/IP "fingerprinting" techniques [1] we exploit the subtle differences in network stack implementations that uniquely identify each operating system. Whereas previous tools rely on an exact match from an exhaustive list of TCP settings, we develop a naive Bayesian classifier that provides a continuous degree of identification confidence without deep-packet inspection.

Rule-based approaches fail to identify as many as 5% of the hosts in traces we collect from an Internet exchange point, likely due to users modifying their TCP parameters or employing stack "scrubbers" [2]. In contrast our classifier can intelligently disambiguate under uncertainty.

While fingerprinting is often regarded as a security attack, we argue that our work is motivated by a number of positive and practical applications. Passively determining operating systems is valuable in intrusion detection systems [3], serving operating system specific content, providing security against unauthorized access, compiling network inventories, building representative Internet models and measuring NAT (Network Address Translation) [4] deployment.

This paper presents results from using our classifier to measure the distribution of hosts, packets and bytes per operating system in our exchange point trace captures.

As an application of our classifier we improve upon previous approaches [5] to infer the frequency and behavior of Internet clients behind NAT devices. Understanding the prevalence of hosts masquerading behind NAT devices has important implications [6][7][8] to the Internet address registries, developers of end-to-end applications and designers of next generation protocols.

The remainder of the paper is organized as follows. We present related work in Section 2. Section 3 describes our inference algorithm and methods to train the classifier. In Section 4 we give our measurement results and distribution of different operating systems. Section 5 details approaches to, and results from, NAT inference. We conclude with suggestions for further research and a summary of major findings.

2 Related Work

Several TCP/IP fingerprinting tools exist employing both active and passive techniques. Active methods involve repeatedly probing a remote host with specially crafted packets to induce operating-system-unique responses. Passive methods rely on an ability to observe a remote host's traffic and identify the machine based on normal traffic flow. Passive fingerprinting is non-intrusive and undetectable, but requires an acceptable monitoring point. Conversely, active tools generate traffic, but can be run anywhere in the network.

The most widely used active tool is the freeware `nmap` [9] program. `nmap` identifies a remote host by finding open TCP and UDP ports, sending specially formed packets and matching the responses to a database of over 450 signatures.

Our research focuses on large-scale measurement using passive fingerprinting. The freeware `p0f` tool [1] provided the original impetus for this work. `p0f` is a rule-based tool containing approximately 200 signatures gathered by the Internet community. The `p0f` signatures contain the following fields: IP time to live (TTL), presence of the IP don't fragment (DF) bit, initial TCP window size, SYN packet size and TCP options. Each signature maps distinctive fields of the IP and TCP header to different operating systems. For each passively observed TCP SYN packet, `p0f` searches the signature space for a match. Unfortunately, we find that `p0f` fails to identify as many as 5% of the hosts in traces we collect from an Internet exchange point.

Bellovin presents a technique for counting hosts behind NAT devices by observing patterns in IP ID sequences [5]. We utilize many of these techniques when performing our NAT measurement, but find several difficulties which we discuss in Section 5.

Because of these difficulties, the only Internet-wide measurements of NAT usage are from a study that attracted clients to a game server [10]. This study found NAT in use among 17% to 25% of the game playing Internet population. The study did not attempt to determine the number of machines masquerading behind each NAT device.

3 Methodology

In this section we detail our inference algorithm, address several practical difficulties and explore different methods of training the classifier.

Passive fingerprinting leverages the fact that different operating systems implement differing TCP/IP stacks, each of which has a unique signature. Even between versions or patches of an operating system there exist subtle differences as developers include new features and optimize performance [11].

Our implementation makes decisions based solely on the TTL, DF, window size and SYN size information which are collectively distinct. The classifier examines the initial TCP SYN packets, but determines the probabilistic likelihood of each hypothesis, i.e. operating system, and selects the maximum-likelihood hypothesis. Thus we always determine a best guess regardless of whether there is an exact signature match.

Before discussing specifics of our implementation, we briefly explain our process of inferring the original TTL of a packet as sent from the source host and mitigate complicating factors with the initial window size.

The IP TTL is decremented by one at each router along the forwarding path. Because our monitoring point is inside the network at an arbitrary distance away from different hosts, a packet's originating TTL as it was sent from a host must be inferred from the observed TTL. We use the well-known heuristic of selecting the next highest power of 2 as the originating TTL. If the observed TTL is greater than 128 we infer an original TTL of 255 and if less than 32 we infer 32.

The initial TCP window size can be fixed in the case of simple stacks or may be a function of the MTU (Maximum Transmission Unit) or MSS (Maximum Segment Size). The MSS is defined as the payload size a packet can carry for the interface MTU. Determining the MSS is complicated in our implementation as we assume visibility into only the first 40 bytes of each packet[1], thereby excluding observation of TCP MSS options. In addition, we assume asymmetrical traffic such that we cannot determine a negotiated MSS. An empirical study of the data to find the most common largest packet sizes revealed five common MSS values: 1460, 1380, 1360, 796 and 536 bytes. The first four are the result of common MTUs and 536 bytes is the default MSS [12]. A further complication arises when the MSS is effectively reduced by the size of the TCP and IP options. We infer the size of the options by taking the size of the SYN packet minus 40 bytes. For each of these five common MSS values, and its equivalent with options, we check if there exists an integer multiple of the window size. If so, the conditional probability of that window size is used.

We wish to evaluate the probability of each operating system hypothesis H_i given the observed data $P(H_i|D)$. Our four observations are: $d_{TTL=ttl}$, $d_{WSS=wss}$, $d_{SYN=syn}$, $d_{DF=\{0,1\}}$. A prediction is made using all the hypotheses weighted by their probabilities. Bayes' rule provides a convenient mechanism to obtain the posterior probability of each hypothesis from causal data. We employ the

[1] In fact our exchange point monitor captures only the first 40 bytes

common simplifying strategy of assuming that each piece of data $d_j \in D$ is statistically independent to obtain the naive Bayesian classifier:

$$P(H_i|D) = \frac{\prod_j P(d_j|H_i)P(H_i)}{P(D)} \tag{1}$$

Clearly the fields are not independent of each other, but analysis has shown naive classifiers can perform close to optimal in practice [13]. Note that for classification, the denominator $P(D)$ is the same for each hypothesis and therefore does not need to be explicitly computed for comparisons.

We experiment with two methods to train the classifier. In the first we use the p0f signature file and take all hypotheses prior probabilities $P(H_i)$ as uniform over the hypothesis space. The p0f signatures are the product of a community effort and are highly accurate.

In our second training method, we collect all TCP SYN packets into a web server. We correlate each SYN packet with an HTTP log entry that contains an explicitly stated browser and operating system. To eliminate bias toward any operating system, we gather logs from a non-technical web server.

Web logs provide an attractive automated method to continually train the classifier without maintaining and updating a database of signatures. However, there are two potential sources of error when training the classifier with web logs. The first source of error is due to hosts connecting through proxies. The web log entry contains the host's operating system, but the captured TCP SYN is from the proxy. Secondly, some browsers provide false information in an attempt to provide anonymity or ensure compatibility. With a large enough training set, we maintain that these errors are statistically insignificant. We present measurements comparing results from our classifier trained with signatures and web logs in the next section.

Our classifier has two main advantages. It gives appropriate probabilistic weight to the value of each piece of data based on the training set, making it robust to user-tuned TCP stacks. Secondly, the classifier produces a maximum-likelihood guess along with a degree of confidence even in the face of incomplete or conflicting data.

The major limitation of our classifier is evaluating its accuracy without a large packet trace of known hosts. Traditionally, the performance of a classifier is measured by taking a known data set, training on some fraction of it and measuring the accuracy of the classifier on the remainder of the data. Unfortunately no data set is available on which to measure the accuracy of our classifier. We have a public web page available that captures each visitor's incoming packets, makes an inference and allows the user to correct an incorrect inference. We plan to measure the accuracy of our classifier once we gather a sufficiently large set of results from our web site.

In lieu of formal verification of the classifier, we gather packet traces from machines of all major operating systems. In addition to testing the stock TCP settings, we modify the socket buffer size, turn on window scaling and enable selective acknowledgments where possible. In many cases the rule-based tool could not identify our machines with modified stacks. We ensure that our tool correctly infers all configurations of machines in our test set.

4 Measurement Results

This section gives the results of our tool on Internet traces. We use our classifier to analyze approximately 38M packets from an hour-long mid-week trace collected at a United States Internet exchange at 16:00 PST in 2003. Within the trace, we analyze only the TCP packets which account for approximately 30.7M packets. Understanding the prevalence of operating systems in the wild allows Internet models to incorporate a realistic distribution of TCP features and provides insight into potential homogeneity implications, e.g. genetic host diversity in the face of epidemic network attacks.

For brevity of presentation, we group operating systems into six broad classifications.[2] We measure the number of unique hosts of each operating system type and per-operating system packet and byte distributions. We compare results from our Bayesian classifier trained with the p0f signatures (denoted Bayesian in the following tables), the classifier trained with web logs (denoted WT-Bayesian) and p0f (denoted rule-based).

Recall that our tool classifies the first TCP SYN of the three way handshake, i.e. the source of the host initiating communication. Thus in contrast to measurements of server operating system popularity [14], our results are mostly representative of client activity. However we will capture particular server initiated traffic, for instance SMTP.

Table 1 displays the operating system distribution among roughly sixty-thousand unique hosts in the trace. Windows based operating systems as a group dominate among all hosts with a greater than 92% share. Although we do not present the results here as they are not generally applicable, we also classify traces gathered from our academic department's border router. By comparison we find only 40% popularity among Windows operating systems in our department's traffic.

We then measure the number of packets attributable to each operating system and again cluster the results into broad classes. For each packet, we determine the operating system of the source based on a prior SYN inference. Surprisingly, Table 2 shows that as a fraction of total packets, the Linux operating system contributes substantially: approximately 19% compared to Window's 77%.

Table 3 depicts the per-operating system distribution of approximately 7.2G bytes. Unexpectedly, Linux operating systems are the largest traffic contributor, with an average of over 2MB of traffic per Linux host in the hour measurement interval. The proportion of byte traffic between Windows and Linux was roughly equal with other operating systems contributing negligibly.

To understand the large discrepancy between operating system counts and the traffic contribution due to each operating system, we process the traces to

[2] Apple's OS-X TCP stack is indistinguishable from BSD, so the results for Mac are under represented, while the BSD results are over represented.

Table 1. Inferred Operating System Distribution (59595 Unique Hosts)

	Bayesian	WT-Bayesian	Rule-Based
Operating System	Percent	Percent	Percent
Windows:	92.6	94.8	92.9
Linux:	2.3	1.6	1.7
Mac:	1.0	2.1	1.0
BSD:	1.6	0.0	1.6
Solaris:	0.4	0.5	0.2
Other:	2.1	1.1	1.0
Unknown:			1.6

Table 2. Inferred Operating System Packet Distribution (30.7 MPackets)

	Bayesian	WT-Bayesian	Rule-Based
Operating System	Percent	Percent	Percent
Windows:	76.9	77.8	77.0
Linux:	19.1	18.7	18.8
Mac:	0.8	1.5	0.8
BSD:	0.8	0.1	1.6
Solaris:	0.7	1.3	0.5
Other:	1.7	0.6	0.2
Unknown:			1.3

Table 3. Inferred Operating System Byte Distribution (7.2 GBytes)

	Bayesian	WT-Bayesian	Rule-Based
Operating System	Percent	Percent	Percent
Windows:	44.6	45.2	44.7
Linux:	52.6	52.3	52.4
Mac:	0.5	0.9	0.5
BSD:	0.7	0.1	1.2
Solaris:	0.7	1.1	0.6
Other:	0.9	0.4	0.1
Unknown:			0.7

find the largest flows. The top ten flows contribute approximately 55% of the total byte traffic. Of these ten, we classify five as Linux, two as Windows, one as BSD. The remaining two flows are not classified, and hence do not contribute to the totals in Table 3, because the monitor did not observe an initial SYN from the source.

The largest traffic source is a software distribution mirror running Linux that is continually transferring data with clients and other mirrors. Four Linux machines continually crawling web pages at a rate of one query every 2 to 3 ms are among the top ten byte flows. We also find email SMTP servers, web caches and P2P applications in the ten largest flows. Thus we conclude that Linux is the largest byte traffic contributor, primarily due to server applications.

5 NAT Inference

Next we describe techniques, including using our tool, to detect the number of hosts masquerading behind a NAT-enabled device. To provide an unbiased trace to evaluate different techniques, we create synthetic NAT traces with a known fraction of masquerading hosts. We evaluate the performance and discrimination power of existing techniques, such as IP ID pattern matching, and our classifier in detecting IP masquerading. Finally we present the results of our NAT inference on the exchange point traces.

5.1 Techniques for Counting NAT Hosts

Previous research toward passively counting hosts behind NAT devices centers on two methods. The first technique [15] requires a monitor before the client's first-hop router, an impossibility for analysis of traces from deep within the network. By assuming the ability to monitor packets at the client attachment point, e.g. a switch or aggregation device, the TTL of packets a monitor observes are not yet decremented by a router and are a standard power of two. For packets with a TTL exactly one less than expected, this method can guess a NAT device is present.

Bellovin provides a more sophisticated approach by observing patterns in IP IDs [5]. Bellovin's algorithm relies on the IP IDs from individual hosts being implemented as sequential counters. By building sequences of IDs that match within reasonable gap and time bounds, one can infer the actual number of machines in a trace.

However as Bellovin notes, matching sequential IP IDs may fail. The purpose of the IP ID field is to allow fragmentation and reassembly by guaranteeing a sufficiently unique number for the lifetime of the packet. There is no defined semantic to the field and hence no reason why it should be a counter. For example, BSD-based operating systems now implement the IP ID as a pseudo-random number rather than a counter. In addition, if the don't fragment bit is set, reassembly is not necessary and hence some NAT devices reset the IP ID to zero. Both issues render IP ID sequence matching ineffective.

A few subtleties remain even assuming non-zero, sequential IDs. Sequence matching actually relies on incorrect NAT behavior: a NAT should rewrite IDs to ensure uniqueness but often does not. Of most relevance to the data from our monitoring point, the potential for misidentification increases when monitoring deep in the network. As the amount of traffic increases, the number of IP stacks producing IDs starting from a randomly distributed point increases. Hence the number of ID collisions increases as a function of observation point depth.

5.2 Evaluation of Sequence Matching

In order to evaluate the performance of Bellovin's method on our traces, we implement our own version of the algorithm. We maintain separate pools of normal and byte-swapped sequences since the IP ID counter may be in network or host byte order. All arithmetic is modulo 2^{16} as follows. The algorithm adds each new packet to the best matching sequence among the two pools. An ideal match is a sequence whose most recent ID is one less than the packet's ID. A packet's ID and time must be within prescribed thresholds gaps in order to match a sequence. If the gap between a new packet's ID and the last ID of all sequences is greater than the threshold, or the time gap is too great, we create a new sequence. After constructing all of the sequences, we coalesce them. We repeatedly combine sequence pairs with the successively smallest gap up to a threshold. All of our thresholds are the recommended values based on Bellovin's empirical results. Finally, sequences with fewer than 50 packets are discarded. The total number of sequences remaining in the normal and byte-swapped pools estimate the number of hosts in the trace.

Because ID collisions overwhelm the algorithm and produce obviously incorrect results, we take Bellovin's suggestion and augment the IP ID technique to discriminate based on IP address. We build, coalesce and prune sequences separately for each unique address.

To evaluate the performance of our algorithms and their suitability for measuring NAT in our live traffic traces, we create synthetic NAT traffic. We capture anonymized traffic from the border router of our academic building where no NAT is present. We then process the trace to reduce the number of unique addresses by combining the traffic of n addresses into one, where n is the "NAT inflation factor."

Using the campus trace of approximately 2.4M packets, we create a synthetic NAT trace with a NAT inflation factor of 2 so that there is the same traffic as in the original trace, but half the number of unique hosts. Using per IP address sequence matching, we obtain a NAT inflation factor of 2.07. Our experimental result well approximates our control data set and gives merit to the approach.

Finally, we apply Bellovin's per IP address sequence matching to our Internet exchange trace. We find an inflation factor of 1.092, lower than our intuitive sense, but providing an intuition for the lower bound.

5.3 Evaluation of Fingerprint Matching

Next, we explore a second technique to count the number of hosts masquerading behind NAT devices. We use our classifier to detect and count multiple packet signatures originating from a single IP address. We assume affects due to DHCP [16] and dual-boot systems are negligible over the hour long span of our trace.

Again, we validate our scheme against the synthesized NAT trace. Using the classifier we obtain a NAT inflation factor of 1.22, significantly less than the correct factor of 2. The per host IP ID sequencing scheme performs closer to ideal than using the classifier as the discriminator on our control trace. Counting NAT hosts on the basis of the unique signatures from each IP address is likely to undercount because the classifier cannot distinguish between multiple machines running the same operating system.

Nonetheless, the classifier finds a 20% host count increase due to machines masquerading behind NAT in the synthetic trace, suggesting that it is a useful metric. Combining several discriminators and techniques, for instance IP address, operating system and IP ID sequence matching, will likely provide even higher accuracy in measuring NAT host counts.

The inferred NAT inflation factor using our classifier to distinguish unique hosts belonging to a single IP address is 1.020. This result, combined with the 9% inflation factor found using ID sequence matching, provides a measurement-based lower bound to understanding NAT prevalence in the Internet.

6 Future Work

Our classifier currently implements simplistic learning based only on the first packet in the TCP handshake. A more interesting and challenging problem is passively identifying the TCP stack variant, e.g. tahoe, and properties of the connection. Methods similar to our classifier could be used to train and evaluate packet streams for more complex inferences.

While we have confidence in our results, we require a large known data set to evaluate and improve the accuracy of our classifier. Once we collect a sufficiently large set of responses from our public web page that gathers packets and user responses, we plan to measure error in our classifier.

Finally, passively counting the number of machines using NAT enabled devices to masquerade behind a single address remains an open problem. One approach we are currently pursuing as an additional data point is measuring the Gnutella peer-to-peer network. The packet header for Gnutella query hits contains a "push" bit so machines behind NAT devices can indicate to other peers that they are not directly reachable. Measuring the prevalence of push queries provides another method of approximating NAT penetration among one user group.

7 Conclusions

We presented a statistical learning technique for passive TCP/IP fingerprinting without deep-packet inspection. Whereas rule-based approaches fail to identify as many as 5% of the hosts in our traces, our classifier provides a continuous degree of identification confidence in the face of incomplete data. We evaluated several approaches to training the classifier, including an automated method with web logs. Analysis of packet traces from an Internet exchange revealed strong dominance of commercial operating systems among all hosts. Surprisingly, we found that the freeware Linux operating system contributes a significant fraction of the byte and packet traffic.

We experimented with a new technique for determining the number of hosts masquerading behind NAT devices by leveraging our classifier. While our technique found many of the additional hosts due to NAT in a synthetic control data set, existing techniques such as Bellovin' IP ID sequence matching performed better. Among the hosts in our exchange point trace, we found a NAT inflation factor of approximately 9% by Bellovin's method and 2% using our classifier. Our results provide a measurement-based lower bound to understanding NAT prevalence in the Internet.

Acknowledgments

We would like to thank Mike Afergan and Kevin Thompson for reviewing initial versions of the paper. Thanks also to the Advanced Network Architecture group and Karen Sollins for her support of this work.

References

1. Zalewski, M.: Passive OS fingerprinting tool (2003) http://lcamtuf.coredump.cx/p0f.shtml.
2. Smart, M., Malan, G.R., Jahanian, F.: Defeating TCP/IP stack fingerprinting. In: Proc. of the 9th USENIX Security Symposium. (2000)
3. Taleck, G.: Ambiguity resolution via passive OS fingerprinting. In: Proc. 6th International Symposium Recent Advances in Intrusion Detection. (2003)
4. Egevang, K., Francis, P.: The IP network address translator (NAT). RFC 1631, Internet Engineering Task Force (1994)
5. Bellovin, S.: A technique for counting NATted hosts. In: Proc. Second Internet Measurement Workshop. (2002)
6. Hain, T.: Architectural implications of NAT. RFC 2993, Internet Engineering Task Force (2000)
7. Senie, D.: Network address translator (NAT)-friendly application design guidelines. RFC 3235, Internet Engineering Task Force (2002)
8. Holdrege, M., Srisuresh, P.: Protocol complications with the IP network address translator. RFC 3027, Internet Engineering Task Force (2001)
9. Fyodor: Remote OS detection via TCP/IP stack fingerprinting (1998) http://www.insecure.org/nmap.

10. Armitage, G.J.: Inferring the extent of network address port translation at public/private internet boundaries. Technical Report 020712A, CAIA (2002)
11. Paxson, V.: Automated packet trace analysis of TCP implementations. In: SIGCOMM. (1997) 167–179
12. Braden, R.: Requirements for internet hosts – communication layers. RFC 1122, Internet Engineering Task Force (1989)
13. Langley, P., Iba, W., Thompson, K.: An analysis of bayesian classifiers. In: National Conference on Artificial Intelligence. (1992) 223–228
14. Netcraft: Web server survey (2004) http://www.netcraft.com.
15. Phaal, P.: Detecting NAT devices using sflow (2003) http://www.sflow.org/detectNAT.
16. Droms, R.: Dynamic host configuration protocol. RFC 2131, Internet Engineering Task Force (1997)