Hershel: Single-Packet OS Fingerprinting

Zain Shamsi, Ankur Nandwani, Derek Leonard and Dmitri Loguinov*
Department of Computer Science and Engineering
Texas A&M University, College Station, TX 77843 USA
{zain, abn9457, dleonard, dmitri}@cse.tamu.edu

ABSTRACT

Traditional TCP/IP fingerprinting tools (e.g., nmap) are poorly suited for Internet-wide use due to the large amount of traffic and intrusive nature of the probes. This can be overcome by approaches that rely on a single SYN packet to elicit a vector of features from the remote server; however, these methods face difficult classification problems due to the high volatility of the features and severely limited amounts of information contained therein. Since these techniques have not been studied before, we first pioneer stochastic theory of single-packet OS fingerprinting, build a database of 116 OSes, design a classifier based on our models, evaluate its accuracy in simulations, and then perform OS classification of 37.8M hosts from an Internet-wide scan.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement Techniques

General Terms

Design, Measurement, Algorithms, Theory, Security

Keywords

OS Fingerprinting; OS Classification; Internet Measurement

1. INTRODUCTION

With the explosive growth and distributed nature of computer networks, it has become progressively more difficult to manage, secure, and identify Internet devices. One of the approaches that greatly helps in understanding the composition of internal and external networks is *OS fingerprinting*, which is a process that determines the operating system of remote hosts based on peculiarities of their network-level behavior. This differentiation is possible due to certain freedom in selection of default stack parameters, ambiguities in IETF RFCs [7], [32], [33], non-compliant TCP/IP implementations, and lacking standardization for responses to malformed requests.

Over the last 20 years, these observations have led to a variety of methods [2], [3], [4], [6], [8], [16], [22], [25], [41], [42], [43], [45], [46], [50], [52], [53], [54] that perform classification using application-layer traffic, TCP/IP/UDP headers, ICMP packets, or some combination thereof. These algorithms are useful not only in network security (i.e., detection of outdated/unpatched hosts), but also market analysis [28] and Internet characterization [17], [23], [29], [31], [45]. However, their usage, scalability, and accuracy at very large scale (i.e., millions of destinations) have not been explored before. We aim to address this issue below.

1.1 Methodology and Objectives

The Internet has been the target of numerous measurement studies, with the trend recently shifting from covering a small subset of destinations [31], [36] to scanning the entire IP space [11], [17], [23], [37]. This allows researchers to enumerate live hosts, detect vulnerabilities, and shed light on deployment of new protocols. Over the years, network scanning has become progressively faster – from 4 months [37] down to 30 days [17], then one day [23], and now 45 minutes [13]. In conjunction with these studies, low-overhead OS fingerprinting can allow significantly better understanding of the systems researchers interact with and improve our general knowledge about the Internet.

OS fingerprinting consists of two approaches – *passive* and *active*. The former [21], [54] monitors ongoing communication (inbound and/or outbound) with remote hosts, but does not generate traffic of its own. Unless each studied device voluntarily connects to the measurement server, this technique is difficult to use for classifying each IP on the Internet. The latter approach, which is the topic of this paper, actively sends packets to targets and infers their operating system from the collected responses.

One important aspect that differentiates between the active methods is the potential maliciousness of probing traffic, where certain nonsensical combinations of TCP flags (e.g., SYN-FIN-RST-ACK) or intrusive actions (e.g., trying to delete the root directory in HTTP fingerprinting [42]) may harm or crash the target. Additionally, these packets are easily detected and dropped by IDS [44], which leads to complaints against research institutions using these methods and possibly reduced accuracy of the results.

The second aspect is the amount of outbound traffic required by the classifier, which ranges from a single SYN probe [3], [50] to lengthy multi-packet exchanges [25], [29],

*Supported by NSF grants CNS-1017766 and CNS-1319984.

SIGMETRICS'14, June 16–20, 2014, Austin, Texas, USA.
Copyright 2014 ACM 978-1-4503-2789-3/14/06 ...$15.00.
http://dx.doi.org/10.1145/2591971.2591972.

[42], [46], [52]. Ideally, fingerprinting should be performed with no extra overhead to scan traffic, which rules out techniques [29], [52] that expect to reach the target on multiple open ports, using different protocols (e.g., ICMP, TCP, UDP), and elicit responses on closed ports. While LAN environments can tolerate high traffic rates and may allow multi-protocol access to each host, these conditions are generally difficult to satisfy when scanning the entire Internet.

The third aspect is the ability of the underlying estimator to correctly identify the target OS under realistic network conditions and without using retransmission. Since prior single-packet techniques [3], [50] were mainly developed for local use, they are not well provisioned to overcome high amounts of fluctuation and loss in temporal features. They also lack resilience to OS tuning, which can be applied by end-users in hopes of optimizing network performance or obfuscating the default parameters of the stack. Either way, the modified OS features may exhibit little correlation to those originally present at the host, which cripples estimation accuracy of existing tools.

1.2 Contributions

Given the many open issues in wide-scale fingerprinting and lacking performance analysis in the literature, our first goal is to formalize the estimation problem in single-packet OS classification and study the pitfalls of existing techniques. We then develop a low-overhead framework we call *Hershel*[1] for overcoming the various randomization effects (i.e., queuing delays, packet loss, manual tuning) and apply it as proof-of-concept in a measurement study that classifies every visible webserver on the Internet.

2. RELATED WORK

OS fingerprinting has roots in *banner grabbing*, which relies on application-layer protocols (e.g., HTTP, SSH, SMTP, FTP, telnet) to provide a textual description of the OS as part of the communication sequence. While this worked well 20 years ago, banner grabbing today faces many impediments, including high overhead, administrator ban on OS-identifying strings in responses, generic software (e.g., Apache) that can run on multiple platforms without exposing the underlying OS, and purposefully incorrect banners that aim to mislead the various fingerprinting tools.

2.1 Multiple Packets

The second wave of OS classification started in 1997 with the release of nmap [29], which pioneered TCP/IP tricks that would elicit different responses from different implementations. By default, it sends 1032 probes to the target, including a vertical port scan and certain malformed packets that trigger popular IDS such as Snort [39]. Nmap ideally expects the target to accept a TCP connection, send ICMP port unreachable on a closed UDP port, and respond to a ping. Under bandwidth-optimized settings for OS classification, nmap requires no fewer than 38 different types of probes; however, due to mandatory retransmission, this in practice corresponds to well over 100 packets per host.

Due to its popularity, nmap has received a great deal of attention in the literature, which includes usage of neural networks to differentiate between versions of a same OS [41],

[1]William J. Hershel is known for inventing forensic usage of fingerprints in 1858.

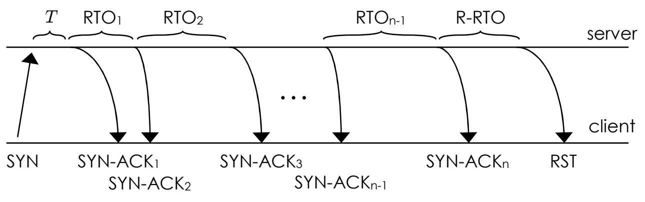


Figure 1: Retransmission timeouts (RTOs) between SYN-ACK packets.

detection of unknown devices [24], and techniques for reducing the number of sent probes [15]. Additional work includes fuzzy matching [52], automatic generation of OS features that aid fingerprinting [8], [38], application of formal testing methods to the detection problem [16], and classification using lengthy observations (up to 100K packets) of Initial Sequence Numbers (ISNs) from the TCP header [25].

2.2 Single Packet

RING [50] and Snacktime [3] are the only tools that perform classification using a single outbound probe. As shown in Figure 1, each measurement consists of a SYN packet, server processing delay T needed to accept the connection, and a stream of n SYN-ACK responses from the target OS, followed by an optional TCP reset (RST) with its own RTO. RING uses the $n-1$ values in the RTO vector and presence of the final RST packet in classification. Snacktime ignores the RST feature, but instead uses the default TCP window size and TTL carried in the SYN-ACKs, which allows it to differentiate between 25 operating systems [3]. We analyze its classification process in more detail later in the paper. A simplified version of Snacktime and extension to 98 signatures was offered in [19], [23]; however, no accuracy analysis, modeling, or verified improvement was provided.

Another tool with a related capability is p0f [54]. In addition to passive fingerprinting, it can actively generate SYN packets and profile remote network stacks based on a set of fixed features from the SYN-ACKs (i.e., window size, TTL, IP flags, and TCP options); however, it does not leverage the RTOs and by default is quite verbose (i.e., sends eight copies of the same SYN per target). The current version can differentiate between 18 operating systems.

2.3 Common Defenses

There exist many approaches to thwart remote OS fingerprinting. The most basic tools tweak Windows registry [10] or implement plugins [4], [5], [40] for the Unix packet-mangling module Netfilter [27]. Their objective is to modify the fixed features of departing packets to no longer resemble those of the underlying host. A similar direction is to deploy network honeypots [35], [49] or standalone systems [51] that spoof arbitrary operating systems and their services. Placing obfuscation into the network gives rise to intermediate devices known as *fingerprint scrubbers* [34], [43].

While these techniques can effectively deal with static header fields, they are not well suited for distorting the temporal features of departing packets, which requires expensive buffering of packets and per-flow state. Additionally, lack of technical support and possibility for various side-effects (e.g., disabling SACK in TCP may lead to significantly lower throughput) raise questions about deployment of these tools in production systems and/or at large-scale.

3. STOCHASTIC MODEL

We assume a single-packet scanner similar to Snacktime in Figure 1. While this approach has minimal intrusiveness, lowest transmission overhead, and non-malicious operation, it also exhibits several fundamental challenges. These arise due to the complex ways in which the RTOs can be modified by packet traversal across wide-area networks, scarcity of information about the target host contained in the samples, and user tuning of features, all of which has a strong influence on one's ability to detect the underlying OS.

It should be noted that straightforward application of machine-learning methods [47] to our problem is difficult. Experimentation with support vector machines, neural networks, and decision trees has led to the realization that they perform poorly when the measured samples contain missing data (i.e., the RTO vector is corrupted by packet loss). Statistical imputation [14] is a common technique for dealing with these problems; however, it requires knowing *which* features are missing and ability to accurately reconstruct the *remaining* (non-missing) features. In our case, lost packets go completely unnoticed and additionally modify the following RTOs to produce feature vectors that have little resemblance to the original (see below).

Our contribution in this section is to formalize single-packet OS fingerprinting, set forth clear goals for the classifier, study the impact of network delay and loss on the measured samples, analyze the existing methods, and outline the assumptions under which the classification problem is tractable.

3.1 Objectives

Assume a database $\mathcal{D} = (1, 2, \dots, M)$ of $M \geq 1$ known operating systems, where each OS j has some vector-valued fingerprint y_j collected during a-priori measurement of the OS. The fingerprint consists of multiple features, which we partition into those modified only by the network (e.g., RTOs) and those only by the user (e.g., TCP window size). Suppose the former are described by some vector δ_j and the latter by another vector u_j. While the length of δ_j normally depends on j, that of u_j is constant across all operating systems.

As both vectors undergo random modification before being observed by the scanner, the response of OS j to probe traffic is some random variable that is a function of y_j. Given an observation $x = (\delta; u)$ from an Internet host, a typical estimation problem is to find the most likely OS $s(x)$ that could have produced that vector:

$$\begin{aligned} s(x) :&= \operatorname*{argmax}_{j \in \mathcal{D}} \; p(y_j|x) = \operatorname*{argmax}_{j \in \mathcal{D}} \frac{p(x|y_j)p(y_j)}{p(x)} \\ &= \operatorname*{argmax}_{j \in \mathcal{D}} \; p(x|y_j)p(y_j), \end{aligned} \tag{1}$$

where notation $p(x|y)$ refers to the probability (or conditional density, if more convenient) of x given y. Observe that the probability $p(x)$ that some OS in $\mathcal{D}$ has produced x is constant for a given observation and can be omitted from the optimization. If the fraction of Internet hosts $p(y_j)$ running OS j is unknown, it is common to set each value to $1/M$, which removes this term from the optimization as well.

The more interesting component of (1) is the probability $p(x|y_j)$ that OS j has produced the observation, or equivalently that y_j has become distorted into x. Before investigating this metric further, observe that network and user modifications to the OS features can be treated as independent, from which it follows that:

$$p(x|y_j) = p(\delta|\delta_j)p(u|u_j). \tag{2}$$

This means that the two terms can be dealt with separately, which we do in the rest of the section.

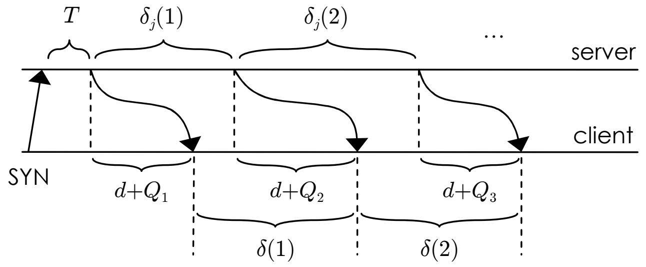


Figure 2: Effect of jitter on observed RTOs.

3.2 Network Features: Jitter

For single-packet techniques [3], [50] in Figure 1, the vector of temporal features δ_j consists of individual RTOs generated by network stack j. Classification based on δ_j is possible not only because some devices deviate from TCP algorithms (e.g., exponential timer backoff), but also because RFCs that govern TCP retransmission [7], [32], [33] do not specify the initial RTO or how many SYN-ACKs must be generated. As a result, a wide variety of unique RTO patterns exists.

For the time being, assume loss-free conditions. During the measurement of sample x, suppose d is the sum of propagation and transmission delays along the path from the server back to the scanner. Note that d is a constant due to the fixed size of SYN-ACKs. Now define Q_m to be a random queuing delay of the m-th packet in the return path. As shown in Figure 2, the RTO vector δ_j undergoes distortion that is independent of the forward path, server think time T, and propagation delay d:

$$\delta(m) = \delta_j(m) + Q_{m+1} - Q_m, \quad m = 1, 2, \dots, |\delta_j| \tag{3}$$

Defining OWD (one-way delay) jitter $J_m = Q_{m+1} - Q_m$ and considering that the gap between subsequent SYN-ACKs is sufficiently large (i.e., at least several seconds), it follows that back-to-back packets arriving from the server are not likely to encounter the same busy period of the queues they traverse. In that case, sequence $Q_1, Q_2, \dots$ consists of independent and identically distributed (iid) random variables. Furthermore, since the number of hops and congestion of the path is not affected by j, the distribution of each Q_m does not depend on the OS being profiled. This leads to:

$$p(\delta|\delta_j) = \begin{cases} \prod_{m=1}^{|\delta|} f(\delta(m) - \delta_j(m)) & |\delta| = |\delta_j| \\ 0 & \text{otherwise} \end{cases}, \tag{4}$$

where $f(.)$ is the PDF (probability density function) or PMF (probability mass function) of OWD jitter, depending on whether J_m is treated as continuous or discrete. It should also be noted that $Var[J_m] = 2Var[Q_m]$, while $f(.)$ is zero-mean and symmetric. For certain models of OWD, jitter can be obtained in closed-form. For example, exponential Q_m produces the Laplace distribution with the same parameter λ and Gaussian $N(\mu, \sigma^2)$ becomes $N(0, 2\sigma^2)$.

	RTO$_1$ (sec)	Y_{j1}	RTO$_2$ (sec)	Y_{j2}	W_j
Observation δ	3.0		24.0		
Fingerprint δ_1	3.0	6	12.0	0	6
Fingerprint δ_2	2.9	1	23.9	1	2

Table 1: Snacktime example.

We next contrast (4) with the RTO classifier in Snacktime [3], which is a tool that is the closest to our objectives and most advanced in single-packet OS fingerprinting. For each RTO, this method first computes the number of matching digits (limited to 6 decimal places of precision) between the sample and all known fingerprints j:

$$Y_{jm} = \max(\lceil -\log_{10}(\max(|\delta(m) - \delta_j(m)|, 10^{-6}))\rceil, 0). \quad (5)$$

It then assigns score W_j to OS j using the sum of these weights across all available RTOs:

$$W_j = \sum_{m=1}^{|\delta|} Y_{jm}. \quad (6)$$

For the example in Table 1, which exemplifies the common pitfalls of Snacktime, (6) scores six for the first OS and two for the second OS, indicating that jitter combination $(0, 12)$ is more likely than $(0.1, 0.1)$. Taking the log of (4), our model can also be reduced to optimizing a summation:

$$\log p(\delta|\delta_j) = \sum_{m=1}^{|\delta|} \log f(\delta(m) - \delta_j(m)); \quad (7)$$

however, it differs from (6) in two important ways. First, the log is applied to the distribution function $f(.)$ rather than the jitter itself. Second, there is no loss of precision due to rounding to the nearest integer or capping the jitter at 10^{-6}. Nevertheless, while (4) is a good starting point, it does not work in real networks due to the lacking robustness against packet loss. This is our next topic.

3.3 Network Features: Loss

The main problem with (4) is that loss-free conditions are impossible to satisfy during Internet scans. Besides congestion, routing loops, and various checksum violations, the RTOs may be altered by the target server crashing or shutting down during the measurement, which affects the tail of the RTO vector and appears similar to packet loss. Since single-packet fingerprinting *by definition* cannot retransmit SYN probes, OS detection must be performed using only the features available in observation x, which calls for more sophistication in the model.

To exacerbate the situation, packet loss creates more dramatic changes to the RTO vector than delay jitter. For example, consider a scenario with $\delta_j = (3, 6, 12)$, where all delays are given in seconds. Even with a relatively large $E[Q_m] = 100$ ms, delay jitter remains small compared to each RTO. On the other hand, the loss of a single packet produces one of four dissimilar combinations – $(3, 6)$, $(3, 18)$, $(6, 12)$, or $(9, 12)$ – while that of two packets leads to six additional options – (3), (6), (9), (12), (18), or (21). The RTO swing in these cases is significantly higher, which makes mapping x to the correct OS more challenging.

We now examine how to model the combined probability that loss and jitter transform δ_j into observation δ. This will allow us to solve such dilemmas as whether $\delta = (3, 18)$ is a more likely match to $(3, 6, 12)$ with one lost packet or to some other signature $(2.6, 17.9)$ without any loss. To deal with these cases, we propose to generalize the concept of RTO. First, let τ_j be a vector of $|\delta_j| + 1$ packet-transmission timestamps from OS j:

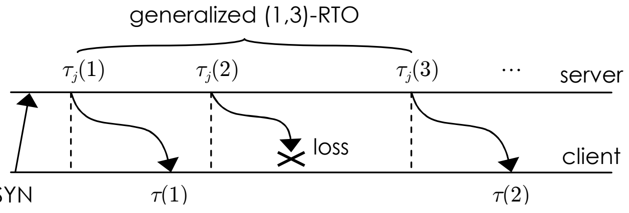


Figure 3: Generalized RTOs under packet loss.

$$\tau_j(m) = \begin{cases} 0 & m = 1 \\ \tau_j(m-1) + \delta_j(m-1) & m \geq 2 \end{cases} \quad (8)$$

and τ be the corresponding random vector observed in x after the packets have traversed the network. Then, a generalized $(m, m+k)$-RTO is the distance $\tau_j(m+k) - \tau_j(m)$, which is illustrated in Figure 3 for $m = 1$ and $k = 2$. Note that $k = 1$ produces the usual RTO and that all timestamps are given using local clocks (i.e., τ_j at the server and τ at the client).

Now suppose set $\Gamma(\tau, \tau_j)$ contains all subsets of size $|\tau|$ of integer sequence $(1, 2, \ldots, |\tau_j|)$. We can view each $\gamma \in \Gamma(\tau, \tau_j)$ as a mapping of received packets in τ to their position in the original vector τ_j, i.e., $\gamma(m) = k$ means that the m-th received SYN-ACK was initially in position k. For the example in Figure 3, we have $\gamma(1) = 1$ and $\gamma(2) = 3$. Assuming no reordering of SYN-ACKs, which is reasonable given at least several seconds between them, each γ is a vector of strictly increasing integers.

Armed with these definitions, we get:

$$p(\tau|\tau_j) = \begin{cases} \sum_{\gamma \in \Gamma(\tau,\tau_j)} p(\gamma) p(\tau|\tau_j, \gamma) & |\tau| \leq |\tau_j| \\ 0 & \text{otherwise} \end{cases}, \quad (9)$$

where the number of summation terms equals the number of ways to select $|\tau|$ objects from $|\tau_j|$ available options and (9) is non-zero only if the number of received packets does not exceed that in the fingerprint. This is in contrast to (4), where the two vectors had to have equal length.

Again leveraging the large spacing between server responses, we can treat congestion events affecting SYN-ACKs as independent, which allows one to approximate packet loss as an iid Bernoulli process with some probability q. Since each loss combination is equally likely, we trivially get:

$$p(\gamma) = q^{|\tau_j| - |\tau|}(1-q)^{|\tau|}, \quad (10)$$

which can be moved outside the summation in (9). To deal with $p(\tau|\tau_j, \gamma)$, which is the probability to observe τ from OS j under loss pattern γ, notice that the gap between each adjacent pair of received packets is determined by the generalized RTO:

$$\tau(m) - \tau(m-1) = \tau_j(\gamma(m)) - \tau_j(\gamma(m-1)) + J'_m, \quad (11)$$

where $m \geq 2$ and generalized jitter J'_m is given by:

$$J'_m = Q_{\gamma(m)} - Q_{\gamma(m-1)}. \quad (12)$$

Rearranging the terms in (11), define the m-th jitter sample under pattern γ as:

$$R_{jm}^{\gamma} = \tau(m) - \tau(m-1) - \tau_j(\gamma(m)) + \tau_j(\gamma(m-1)). \quad (13)$$

Noticing that J'_m has the same distribution as J_m yields:

$$p(\tau|\tau_j, \gamma) = \prod_{m=2}^{|\tau|} f(R_{jm}^{\gamma}). \quad (14)$$

We thus get for $|\tau| \leq |\tau_j|$:

$$p(\tau|\tau_j) = q^{|\tau_j|-|\tau|}(1-q)^{|\tau|} \sum_{\gamma\in\Gamma(\tau,\tau_j)} \prod_{m=2}^{|\tau|} f(R_{jm}^{\gamma}), \quad (15)$$

which replaces $p(\delta|\delta_j)$ in (2).

We finish this subsection by noting that Snacktime only considers operating systems with the same number of RTOs as the measured sample x, which automatically results in a mismatch for observations that have sustained loss.

3.4 User Features

OS tuning is common practice in the current Internet, with numerous online guides recommending optimizations to network settings [30], [48] and automated software offering tuning capabilities to the TCP/IP stack to achieve better performance [12]. A number of fixed header parameters in general-purpose kernels (e.g., Unix, Windows) can be changed through registry or using command-line tools. Unlike jitter-induced noise, where small distortions are generally more likely that large ones, the main difference with OS tuning is that *there may be no correlation between the manually selected values of the user and those installed in the OS by default.* For example, TCP window size may be more likely to jump from 8192 to 65535 than to 8193.

While accurate modeling of manual modification and human psychology is difficult, it makes sense for the analysis to at least take into account whether a given feature under user control has been changed. Suppose that π_m is the probability of such modification in feature m and assume that user tuning is applied independently to each available parameter. Defining $I_{jm} = \mathbf{1}_{\{u(m)=u_j(m)\}}$ to be an indicator of the event that the m-th measured feature matches the original of OS j, we get:

$$p(u|u_j) = \prod_{m=1}^{|u|} \Big[(1-\pi_m)I_{jm} + \pi_m(1-I_{jm})\Big]. \quad (16)$$

Besides user interference, vector u_j may be modified by intermediate devices along the path (e.g., NAT, IDS, fingerprint scrubbers [10], [34], [40], [43], [51]), whose actions can be clumped under the same umbrella of (16). Since buffering packets for periods of time comparable to RTO (i.e., $3-6$ seconds) and per-flow state are expensive, it is often safe to assume that these devices do not alter the RTO pattern in significant ways and thus leave enough features by which the OS can still be identified. This underscores the importance of having a robust RTO estimator.

The Snacktime algorithm for scoring user-modified features can be generalized as a sum of weights assigned to each match:

$$W'_j = \sum_{m=1}^{|u|} w_m I_{jm} = \sum_{I_{jm}=1} w_m, \quad (17)$$

which is added to the RTO score W_j in (6) for a final result. One open issue, however, is selection of proper weights, which need to be somehow correlated with feature volatility. Our model is much simpler since π_m directly provides this probability. To better understand the difference between (16) and (17), assume that $\pi_m > 0$ for all m and write:

$$\log p(u|u_j) = \sum_{I_{jm}=1} \log(1-\pi_m) + \sum_{I_{jm}=0} \log \pi_m. \quad (18)$$

For $\pi_m \approx 1$, we get $\log \pi_m \approx 0$, the second term of (18) disappears, and our model reduces to Snacktime with weights $w_m = \log(1-\pi_m)$. However, in more realistic cases of $\pi_m \ll 1$, the second term of (18) becomes non-negligible and serves the role of balancing non-matching features against those that do match. Snacktime has no such mechanism.

3.5 Final Result

We now consolidate the various models into one formula. Combining (15) and (16) in (2) and (1), dropping terms that do not depend on j, and performing straightforward manipulations, we get:

$$s(x) = \operatorname*{argmax}_{j\in\mathcal{D}:|\tau|\leq|\tau_j|} \Big\{ p(y_j) q^{|\tau_j|-|\tau|} \sum_{\gamma\in\Gamma(\tau,\tau_j)} \prod_{m=2}^{|\tau|} f(R_{jm}^{\gamma}) \times \prod_{I_{jm}=1} (1-\pi_m) \prod_{I_{jm}=0} \pi_m \Big\}. \quad (19)$$

Although (19) maximizes the OS-detection probability under the assumptions stated throughout this section, its performance with a-priori-unknown q, π_m, $f(.)$, and $p(y_j)$ is an open question that needs to be examined. We perform this later in the paper. In the meantime, we outline the various remaining issues.

3.6 Limitations

First, the SYN packet may be lost and never reach the target. Since there is no way to verify this, the host will automatically be considered non-responsive and will be excluded from fingerprinting. Not much can be done to overcome this problem unless SYN retransmission is allowed. If we relax the single-packet assumption, the estimator will face the problem of determining which of the SYNs triggered which SYN-ACK response, without which the RTOs cannot be computed correctly. This problem can be solved in the future by encoding the retransmission attempt into the source port of the SYN.

Second, our model allows only the *network* to modify the RTOs; however, this may not hold if users manage to alter SYN-ACK spacing during OS tuning. This is not of wide-spread concern as few optimization guides target the RTO pattern. With enough effort, scrubbers and obfuscation tools can disrupt inter-SYN-ACK delays; however, we do not consider development of end-to-end methods to combat such approaches a fruitful objective. A related problem arises with middleboxes and caches that accept the connection on behalf of the server [18], in which case any fingerprinting tool is bound to classify only the visible side of the TCP stream (i.e., the OS of the middlebox).

Third, Hershel's accuracy may deteriorate if the network jitter process J_m becomes non-iid or deviates from the predicted bounds, e.g., due to significant kernel scheduling la-

Operating system	Win	TTL	DF	Reset				MSS	OPT	SA-RTO	R-RTO
				RST	RA	RN	RW				
Windows 7	8192	128	1	1	0	1	0	1460	MNWST	3, 6	12
Linux 2.6	5792	64	1	0	–	–	–	1460	MSTNW	3.8, 5.9, 12.1, 24, 48.2	–
Linux 2.0	32736	64	0	0	–	–	–	1414	M	3, 6, 12, 24, 48, 96	–
Mac OS 10.3	33304	64	1	1	1	1	32768	1460	MNWNNT	2.92, 6, 12, 24	30
NetBSD 4.0.1	32768	64	1	0	–	–	–	1460	MNWNNTSNN	2.92, 6, 12, 24	–
VxWorks 5.4.2	8192	64	0	1	1	1	8192	512	MNW	5.58, 24	45
Juniper Netscreen	8192	64	0	1	0	0	8192	1380	M	1.67, 2, 2, 2, 2, 2, 2, 2	2

Table 2: Sample signatures (M = MSS, N = NOP, W = window scale, S = SACK, T = timestamp).

Operating system	All enabled	Drop S	Drop T	Drop W	Drop ST	Drop SW	Drop WT	Drop all
Linux 2.6	MSTNW	MNNTNW	MNNSNW	MST	MNW	MNNT	MNNS	M
Windows XP/2003	MNWNNTNNS	MNWNNT	MNWNNS	MNNTNNS	MNW	MNNT	MNNS	MNW
Windows 7/2008	MNWST	–	–	MST	–	–	–	–
FreeBSD 8.2	MNWST	MNWNNT	–	–	–	–	MSE	M
Solaris 10	NNTMNWNNS	NNTMNW	–	–	–	–	–	–

Table 3: Examples of transformations applied by the OS to TCP options (dashes indicate impossible cases).

tency during CPU overload. Similar issues may surface if network loss depends on j, users modify different operating systems with different probability, or there is correlation in loss events within a single stream of SYN-ACKs. Solving these problems requires a per-OS set of parameters $(q_j, f_j(.), \pi_{jm})$ and multi-dimensional covariance matrices for joint distributions of RTOs and individual features modified by tuning, all sampled under realistic load conditions. Needless to say, these are difficult to come by, but we may consider this direction in future work.

4. CLASSIFIER

Our next contribution is to enhance Snacktime's feature vector, describe a working classifier based on the theory developed in the previous section, bring awareness to RTO randomization performed by certain OSes, and explain how to collect signature databases under these conditions.

4.1 Features

Snacktime uses only two non-RTO features – TCP advertised window size and TTL; however, additional parameters are readily available from the SYN-ACKs. Following Table 2, these include the Do Not Fragment (DF) flag in the IP header, four different fields from the RST packet (more on this below), the Maximum Segment Size (MSS) declared by TCP, the order in which the OS assembles the option fields (OPT), SYN-ACK RTOs (SA-RTO), and the RST RTO (R-RTO). Some of these features are self-explanatory, but others require additional elaboration.

First, it should be noted that the initial TTL cannot be reconstructed exactly at the receiver. We use the common technique of rounding this value up to the nearest "likely" boundary, which includes four values used by the OSes in our database $\mathcal{D}$ – 32, 64, 128, and 255. Second, the reset features are quite rich. In Table 2, the binary flag RST is 1 for the fingerprints that contain a reset packet, RA indicates whether the RST has the ACK bit set, RN is 1 if the ACK sequence is non-zero, and RW records the window of the reset packet. RST features represent peculiarities of internal stack operation and cannot be modified via OS tuning. However, fingerprint scrubbers, NAT/IDS, and kernel recompilation can still change them.

Third, as seen in the table, support for TCP options differs between the operating systems since no specific subset is required to be implemented [20]. More importantly, users have the freedom to disable them as needed. As certain options are considered security risks (e.g., timestamps), they may be disabled by default, although users can still re-enable them. Certain devices (e.g., printers) do not allow OPT tweaking at all, while newer versions of popular operating systems tend to support fewer choices. For example, even though Windows 7/2008 provides registry keys to disable TCP timestamps, the modification does not work. Similarly, SACK can be disabled only if the entire TCP stack is offloaded to the NIC [26].

What makes OPT a good feature is not the specific string, but rather *the order in which non-padding options appear*. This is illustrated in Table 3, where we progressively disable various combinations of options and observe the resulting SYN-ACK packets. For example, Windows XP supports four options MWTS. Turning off W produces MTS interspersed by NOPs as padding. Simplicity of implementation and lacking reasons to reorder the options suggests that this phenomenon likely exists in other stacks.

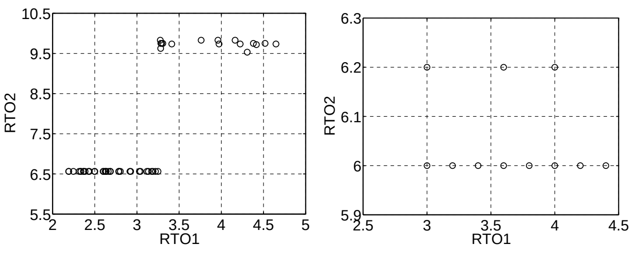
As a result, OPT requires a more advanced classification logic than straight comparison. Specifically, a match is registered if the observed sample x contains a *feasible* string, which we examine by taking an intersection of non-NOP options between x and each fingerprint, followed by verification that the order of the resulting letters is the same. For example, MTW is a match to Linux, VxWorks, and Juniper in Table 2, but not the other OSes.

Fourth, the reset RTO (R-RTO) helps in resolving additional ambiguities, such as between Mac OS 10.3 and NetBSD 4.0.1 in Table 2, which otherwise have identical SA-RTO patterns. Additionally, we expand Snacktime's default measurement time limit from 65 seconds to 120, the latter of which is the MSL (Maximum Segment Lifetime) of TCP [33]. For instance without considering the 96-second RTO of Linux 2.0 in Table 2, it might be hard to differentiate it from Linux 2.6.

Table 4 summarizes the features used in our classification and compares them to those in nmap, p0f, and Xprobe [29], [46], [50], [52], [54]. We have four novel features and one match type (subset) never used in fingerprinting before.

Feature	Description	Appeared In
Win	Receiver window	[3], [29], [46], [52], [54]
TTL	Time-to-live field	[3], [29], [46], [52], [54]
DF	Do Not Fragment	[29] [46] [52], [54]
SA-RTO	RTO sequence	[3], [46], [50]
RST	True if RST packet	[50]
MSS	Max segment size	[29], [46], [54]
OPT	TCP options (exact)	[29], [54]
RA	ACK bit in RST	New
RN	ACK seq $\neq 0$ in RST	New
RW	Window in RST	New
OPT	TCP options (subset)	New
R-RTO	RTO of RST packet	New

Table 4: Enhanced feature vector.



(a) Windows Server 2003 (b) Linux 2.6

Figure 4: RTO randomness in TCP/IP scheduler.

4.2 Stochastic Timers

Table 2 shows SA-RTOs from a single captured sample of the OS; however, it turns out that many kernels naturally exhibit significant RTO variation, sometimes by as much as 50%. Two examples are shown in Figure 4 using a 2D scatter plot of the first two SA-RTOs. For Server 2003 in subfigure (a), there are two distinct patterns – the lower left corner, with RTO_1 distributed in $[2.2, 3.3]$ and RTO_2 frozen at 6.56, and the upper section, with RTO_1 scattered in $[3.3, 4.6]$ and RTO_2 in $[9.5, 9.8]$. Furthermore, the two scenarios are not equally likely as the bottom one occurs 68% of the time. This shows that the temporal model must take into account not just the possible RTO regions, but also their likelihoods.

A similar picture emerges for Linux 2.6 in subfigure (b). The mass of the RTO is now concentrated on 11 distinct points, where RTO_1 ranges from 3 to 4.4 seconds and RTO_2 from 6 to 6.2. Again, the popularity of individual points is non-uniform, swinging from 2% to 16%. Note that both cases in Figure 4 have been collected from idle hosts over a single-hop network consisting of one switch, *which makes this behavior part of the fingerprint itself rather than an artifact of the sampling environment.*

Possible reasons for this fluctuation are the absence of per-connection RTO timers during the SYN-ACK phase and discretization of retransmission delays. What these examples show is that internal OS operation is a complex stochastic system that requires measuring the RTO *distribution* (rather than a single snapshot) during creation of the signature database. This is necessary because such large variations are not taken into account by the jitter model, which normally assumes OWDs on the order of tens or hundreds of milliseconds, with similarly sized jitter.

Our approach is to treat RTOs as random variables, unlike prior work that has always considered them deterministic. Specifically, suppose OS j has w_j unique types of behavior, each occurring with probability β_{jr}, where $r = 1, 2, \dots, w_j$. We call each of these types a *subOS* and assign it a separate RTO vector τ_{jr}, which updates (14) to:

$$p(\tau|\tau_j,\gamma) = \sum_{r=1}^{w_j} \beta_{jr} p(\tau|\tau_{jr},\gamma). \tag{20}$$

A simpler technique is to measure each host w times and let each obtained RTO vector τ_{jr} be a subOS with $\beta_{jr} = 1/w$. In that case, (20) becomes:

$$p(\tau|\tau_j,\gamma) = \frac{1}{w}\sum_{r=1}^{w}\prod_{m=2}^{|\tau|} f(R^{\gamma}_{jrm}), \tag{21}$$

where R^{γ}_{jrm} is the generalized jitter of the m-th RTO under subOS r of OS j and loss-pattern γ. Note that summations involving $\Gamma(\tau, \tau_j)$ remain the same since all subOSes within a given OS send a fixed number of SYN-ACKs. They also exhibit deterministic user features, which keeps (16) unchanged.

4.3 Fingerprint Database

In order to produce an accurate fingerprint τ_j, the OS must be measured in some isolated testbed with low end-to-end delays and idle conditions at the server. To avoid loss-related bias, each host must be sampled multiple times to determine the longest vector of RTOs it produces, which should then be used to collect w loss-free samples for the database. Following these guidelines, we installed a variety of commodity operating systems in our lab, determined the proper size of their RTO vectors, and collected $w = 50$ clean fingerprints from each. We also captured a number of embedded devices found in our department LAN.

The final step was to perform a scan of the university network for additional signatures not already in the database. Once found, these devices were fingerprinted in a similar fashion, producing w clean samples per OS. When the owner of the device could not be contacted and the server's webpage did not provide enough detail, we used nmap to identify the device. While Snacktime ships with 25 signatures and [23] uses 98, our database contains 116 network stacks. We can distinguish not only between different operating systems (e.g., Windows, Linux, FreeBSD), but also sometimes identify their versions and patches (e.g., Windows Server 2003 with and without SP1, MacOS 10.3 vs MacOS 10.4).

4.4 Hershel

Our classification method, which we call *Hershel*, builds upon (19) and (21), where we treat all $w = 50$ subOSes as deterministic. Common sense suggests that users, scrubbers, and network devices are not likely to directly tweak individual RST features RA, RN, and RW; instead, these fields (if modified at all) will be simultaneously replaced with another set that comes from a different OS. We thus combine all four RST values in Table 2 into one atomic feature for classification purposes. This makes vector u_j consist of six fields – Win, TTL, DF, aggregated RST, MSS, and OPT.

RTO vectors τ and τ_j includes timestamps of all SYN-ACKs and the first RST (if present). To account for resets that might be injected by firewalls/IDS after they time out the connection, (9) and (16) require a revision. Specifically, if the measured vector τ contains a reset, but $|\tau_j|$ does not, the RST is removed from τ prior to computing (9). To

OWD	μ	Snacktime		Hershel $\lambda = 2, q = 0.038$						Hershel	Hershel
		RTO	+Win/TTL	RTO	+Win/TTL	+DF	+OPT	+MSS	+RST	$\lambda = 10$	$q = 0.1$
$q_{real} = 0, \pi_{real} = 0$ ($\chi = 100\%$)											
Pareto	0.5	12.6	58.3	22.1	86.2	88.5	96.2	99.72	99.72	94.62	99.69
Exp	0.5	12.8	58.3	21.9	86.9	89.4	96.5	99.92	99.94	96.21	99.82
Uniform	0.5	13.0	58.4	21.7	87.4	89.8	96.8	99.99	99.99	98.50	99.99
Pareto	0.1	16.3	62.9	33.1	94.9	96.7	99.0	99.99	99.99	99.69	99.99
$q_{real} = 3.8\%, \pi_{real} = 0$ ($\chi = 84\%$)											
Pareto	0.5	10.0	49.0	21.4	85.1	87.7	96.1	99.69	99.69	94.68	99.66
Exp	0.5	10.1	49.0	21.5	85.6	88.1	96.3	99.76	99.82	96.21	99.80
Uniform	0.5	10.3	49.0	21.7	86.4	89.0	96.7	99.96	99.96	98.50	99.96
Pareto	0.1	13.1	53.2	31.6	93.6	95.6	98.8	99.96	99.96	99.66	99.97
$q_{real} = 3.8\%, \pi_{real} = 10\%$ ($\chi = 49\%$)											
Pareto	0.5	10.0	44.4	21.4	77.7	78.6	91.4	94.93	95.37	90.13	95.25
Exp	0.5	10.1	44.4	21.5	78.3	79.1	91.6	95.02	95.55	91.78	95.34
Uniform	0.5	10.3	44.4	21.7	78.9	79.7	91.9	95.20	95.63	93.97	95.57
Pareto	0.1	13.1	48.5	31.6	87.3	87.7	95.0	96.54	96.92	96.67	96.87
$q_{real} = 10\%, \pi_{real} = 10\%$ ($\chi = 34\%$)											
Pareto	0.5	6.9	33.4	20.1	76.2	77.1	91.2	94.84	95.22	90.01	95.14
Exp	0.5	7.0	33.4	20.1	76.8	77.7	91.5	94.98	95.43	91.76	95.20
Uniform	0.5	7.2	33.4	20.1	77.4	78.3	91.7	95.13	95.51	93.82	95.46
Pareto	0.1	9.3	36.8	29.4	85.3	85.7	94.5	96.38	96.71	96.46	96.67
$q_{real} = 50\%, \pi_{real} = 50\%$ ($\chi = 0.13\%$)											
Pareto	0.5	0.82	2.49	10.4	28.1	35.6	53.7	56.65	59.95	58.95	60.23
Exp	0.5	0.83	2.49	10.5	28.4	35.9	53.8	56.74	60.12	60.40	60.31
Uniform	0.5	0.84	2.49	10.6	28.6	36.5	54.0	56.89	60.25	60.79	60.46
Pareto	0.1	1.11	2.95	14.4	32.0	40.5	56.8	59.45	62.68	64.84	63.06

Table 5: Classification accuracy (percent) in simulations of 2^{18} samples.

account for the mismatch in the RST feature, (16) gets multiplied by π_4. In the opposite case, i.e., $|\tau_j|$ contains a RST, but $|\tau|$ does not, it is important to avoid mistaking packet loss for changes in the RST feature and improperly penalizing $p(u|u_j)$ with π_4. Next, if both vectors contain a reset packet, (16) gets hit with either π_4 or $1 - \pi_4$ depending on the match in (RA, RN, RW). Finally, if neither vector has a RST, then (16) enjoys multiplication by $1 - \pi_4$.

5. SIMULATIONS

Our contribution in this section is to explain how to select the parameters of the model, and examine Hershel's accuracy in simulations in comparison to Snacktime.

5.1 Parameters

For lack of a better assumption, we suppose that all OSes are equally likely to appear in the trace and set $p(y_j) = 1/M$ to be a uniform PMF. While it is possible to consider multiple iterations and refine this value after each pass, the resulting system sometimes exhibits instability and divergence into inferior states. We thus leave stability analysis for future work and only perform a single iteration in our evaluation below.

We use $\pi_m = 0.01$ for RST and OPT, while keeping $\pi_m = 0.1$ for the other features. The rationale is that RST behavior and option ordering can be changed only through kernel source-code modifications and usage of aggressive intermediate devices, neither of which we believe is that common in today's Internet compared to stack tuning. For queuing delay, we use a simple exponential distribution with CDF $1 - e^{-\lambda x}$ whose mean is set to 0.5 seconds (rate $\lambda = 2$). This produces Laplace jitter density:

$$f(z) = \frac{\lambda}{2} e^{-\lambda|z|}. \tag{22}$$

Note that usage of $\lambda = 2$ is fairly pessimistic, with the majority of paths likely exhibiting significantly smaller delays. For example, this model assumes 82% of the paths produce over 100 ms queuing delays, 37% over 500 ms, and 14% over 1 second. For packet loss, we use Google's study [9] to set $q = 3.8\%$, which was their highest rate of SYN-ACK loss.

5.2 Results

Our next goal is to examine Hershel's robustness in the presence of OWD jitter, packet loss, and random feature modification by the user. We also aim to assess the sensitivity of results to our choices of default parameters above. We simulate a FIFO queue between the server and the client with a given delay distribution. Each packet is dropped by the router with some probability q_{real} and each feature is independently modified with another probability π_{real}. Since these are per-packet and per-feature metrics, it also makes sense to examine the fraction $\chi = E[(1-q_{real})^{|\tau_j|}](1-\pi_{real})^6$ of all generated samples that do not have any loss or feature modification, where the expectation is taken over all j.

The distribution of popularity $p_{real}(y_j) \sim j^{-\alpha}$ is set to Zipf with shape parameter $\alpha = 1.2$, which approximates the fact that some OSes are much more popular than others. We do not attempt to make our assignment of index j to each physical OS such that its $p_{real}(y_j)$ closely follows that in the Internet (which is unknown anyway); instead, the simulation simply verifies performance of the proposed estimator when the OS frequency is highly non-uniform. For that purpose, random ordering of OSes in the database is sufficient.

Table 5 shows classification accuracy for several scenarios of interest. We examine three types of OWD with mean μ in the first column – Pareto $1-(1+x/\beta)^{-\alpha}$ with $\alpha = 3$ and $\beta = \mu(\alpha - 1)$, exponential with rate $1/\mu$, and uniform in $[0, 2\mu]$. We use the original Snacktime since the simplified version from [19] performs worse. Using just the RTOs, Snacktime in the table starts at 13%, but then deteriorates below 1%

RTOs	Hosts	Sigs	Group
3	9,639,810	27	all
2	9,070,991	16	windows, embedded
5	7,834,027	23	linux, embedded, other
4	5,066,940	16	unix, embedded
1	2,669,222	1	Dell printer
0	1,992,196	0	–
6	540,042	9	linux, embedded, other
19	202,733	2	embedded
18	162,442	0	–
17	110,335	0	–

Table 6: Top RTO counts (99% of total).

Window	Hosts	Sigs	Group
5,792	10,143,772	4	linux
16,384	7,051,858	6	windows, embedded, other
8,192	4,266,370	17	windows, embedded
65,535	3,551,640	9	windows, other
5,760	2,643,274	0	–
5,840	981,136	3	embedded
16,000	781,225	5	embedded
4,096	775,473	5	embedded
1,024	758,230	4	embedded
2,800	677,211	1	TP-Link router

Table 7: Top window sizes (87% of total).

near the bottom. This amounts to essentially guessing across the 116 available options (i.e., $1/116 = 0.86\%$). Augmented with Win/TTL, Snacktime begins at a more healthy 58%, but then quickly reduces to single digits.

The next six columns show Hershel with its default $\lambda = 2$. Classifying just based on the RTO vector, Hershel doubles Snacktime's accuracy in the first three scenarios (i.e., the first 12 rows of the table), triples it in the next one, and improves by an order of magnitude in the last one. As additional features are added, Hershel becomes even better, with significant gains seen at the Win/TTL and OPT boundaries. This shows that unlike DF, option strings form an orthogonal dimension to Win/TTL. The MSS improves the result further by 3% and the RST packet by an additional $0.5-3\%$, with the impact mostly limited to high-loss cases.

Staying with $\lambda = 2$, observe that Hershel is quite insensitive to selection of $f(z)$. Specifically, classification accuracy improves *not* when λ equals $1/\mu$ or the PDF of real delay matches (22), but *as μ gets smaller or the tail of the delay gets lighter*. This can be seen by contrasting the two Pareto cases ($\mu = 0.1$ and $\mu = 0.5$) and comparing Pareto, exponential, and uniform cases (all with $\mu = 0.5$). As the difference between the last three scenarios is quite small, we conclude that the distribution of network jitter, as opposed to its mean, generally has a minor effect on accuracy. Therefore, keeping the Laplace model (22) for the experiments in the next section appears reasonable.

To shed additional light on selection of parameters, the next column of the table re-runs Hershel with all available features and $\lambda = 10$. While this slightly improves the $\mu = 0.1$ case, this happens only under 50% packet loss and at the expense of significant reduction in accuracy in other rows, which suggests that $1/\lambda$ should *overestimate*, rather than *underestimate*, the real network delay. To this end, our previous conservative choice $\lambda = 2$ seems quite appropriate. The last column of the table reverts to $\lambda = 2$ and demonstrates that the model is insensitive to selection of q. We thus keep $q = 3.8\%$ for the Internet classification below.

6. EXPERIMENTS

Our contribution in this section is to apply Hershel to a wide-scale Internet scan and provide an assessment of the obtained classification.

6.1 Dataset Properties

We use Internet scan data from [19], which is based on a 2010 survey of webservers in [23]. These IPs were discovered by sending port-80 SYN packets to every address in BGP. The experiment garnered 37.8M samples x that contained at least one SYN-ACK, which we later feed into Hershel.

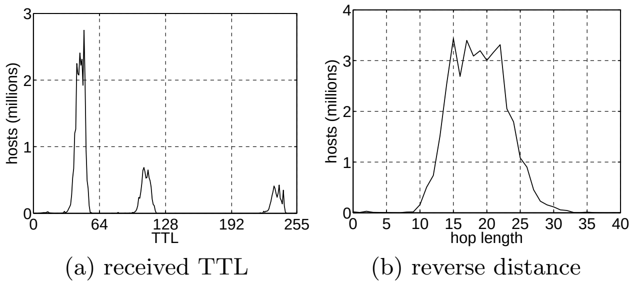


Figure 5: Received TTL and reverse path length.

We start by examining occurrence of various features in the dataset and their mapping to signatures in $\mathcal{D}$. We qualitatively group them into four types – linux, windows, embedded (routers, modems, cameras, hardware gadgets), and other (BSD, Mac, AIX, NetApp, Big-IP, SunOS).

To first step is to ensure that packet loss has not produced totally unworkable temporal features in the dataset. Table 6 shows the number of available RTOs per destination. It is encouraging to see that the top four spots retain enough information for a meaningful match and the most difficult case (i.e., single SYN-ACK) follows in distant sixth place. We next analyze sanity of the remaining features and build intuition for what to expected from Hershel classification.

The scan contains a staggering 3,815 unique window sizes, while our fingerprint collection $\mathcal{D}$ has only 51. The good news is that the distribution of window size is heavily skewed towards mostly well-known values, as seen in Table 7. The most common window is unique to Linux variants, while the most ambiguous is split across 17 operating systems. Interestingly, window size 5760 in position #5, which we later discovered belongs to Ubuntu, is also not present in other fingerprinting databases (p0f, xprobe). We come back to these hosts later in the section and examine how Hershel classifies them. Ideally, unknown devices should be mapped to the same OS family (i.e., Linux in this case).

The TTL values of received packets are plotted in Figure 5(a), covering 251 unique points out of the 255 possible. A vast majority of the hosts are clustered on the values just before the initial TTL defaults 64, 128, and 255. Figure 5(b) shows the distribution of reverse hop length for each host back to the scanner, calculated by subtracting the received TTL from the nearest well-known initial value. This distribution appears reasonable, with less than 0.4% of the mass below 10 or above 30 hops. This suggests the number of non-standard initial TTLs (if any) is small. Table 8 shows the

TTL	Hosts	Sigs	Group
64	26,275,301	70	linux, embedded, other
128	7,129,667	17	windows, embedded, other
255	4,214,927	22	linux, embedded, other
32	190,697	7	embedded

Table 8: Initial TTL distribution.

Feature	Hosts	Sigs	Group
RA= 1, RN= 1	4,368,098	35	embedded, other
RA= 0, RN= 1	1,167,761	11	windows, embedded
RA= 0, RN= 0	367,915	10	embedded
RA= 1, RN= 0	37,113	0	–

Table 9: Breakdown of 5.9M hosts with RSTs.

distribution seen by Hershel and the corresponding number of signatures in $\mathcal{D}$.

A good number of hosts (69%) set the DF flag, indicating they intend to perform path-MTU discovery, which matches 45% of the signatures. Out of 37.8M responsive targets, 5.9M (16%) send at least one reset packet (in addition to the SYN-ACKs), which is consistent with 56 OSes. The reset window (RW) deviates from that in the SYN-ACK for 20.8% of the IPs and 8 fingerprints in $\mathcal{D}$.

Table 9 examines the interplay between RA and RN in reset packets. In the most common scenario, hosts indicate that the ACK sequence is valid and correctly acknowledge values one larger than transmitted by the scanner in the SYN packet (which encodes the destination IP); however, there are also 37K hosts (last row) with broken implementations that indicate a valid ACK, but set the field to zero. None of our signatures exhibit this behavior.

We have 21 unique combinations of options in $\mathcal{D}$; however, the dataset shows 264 different strings, with the top 10 provided in Table 10. Similar to Table 7, a few popular cases account for the majority of IPs and Linux variants hold a clear lead, but now the most ambiguous combination splits across 41 embedded devices. While Akamai currently reports 137K servers [1], it seems reasonable that multiple NICs and IP aliasing can produce 339K samples in last row.

Practically every host (99.5%) supports the MSS option, with Table 11 showing the top 10 cases out of the 1,021 observed in the dataset. The most common MSS 1460 does not provide much information about the OS, but the other values appear useful at partitioning the dataset into small groups. On the downside, general-purpose OSes often set the MSS as a function of the underlying data-link layer (i.e., MSS = MTU – 40), which creates some interesting dilemmas. For example, MSS 1452 in third place can be classified as one of two embedded devices or as home computers with

Options	Hosts	Sigs	Group
MSTNW	13,156,171	8	linux
MNWNNT	6,214,837	18	embedded, other
MNWNNTNNS	5,579,866	12	windows, other
M	5,431,682	41	embedded
MNW	2,656,342	5	linux, embedded, other
MNWST	1,107,935	2	windows, unix
MNWNNTSEE	762,593	4	other
MNNSWNNNT	412,602	0	–
MST	370,699	1	Windows Vista/7
MNNSNW	339,215	1	Akamai linux

Table 10: Top options strings (95% of total).

MSS	Hosts	Sigs	Group
1460	21,969,799	70	all
512	3,523,272	9	embedded
1452	3,512,626	2	embedded
1380	1,633,852	3	windows, embedded
1440	1,472,969	2	linux, embedded
1400	1,074,502	2	embedded
536	620,013	7	embedded
1448	562,961	0	–
1420	431,720	1	Avocent KVM switch
768	419,326	2	embedded

Table 11: Top MSS values (93% of total).

Feature	Fraction	RST possibilities	Fraction
Win	70.3%	Neither has RST	80.9%
TTL	95.2%	Both have RST, match	10.4%
DF	96.2%	Missing RST	4.2%
MSS	70.6%	Both have RST, non-match	3.5%
OPT	99.4%	Bogus extra RST	1.0%

Table 12: Hershel's feature match rate.

1492-byte MTUs commonly seen over PPP links such as DSL. This emphasizes importance of Hershel's probabilistic matching (16) and explains the significantly smaller number of unique MSS values in $\mathcal{D}$ (i.e., only 20).

6.2 Classification

We run Hershel on the scan dataset and obtain a non-zero classification probability for 37.4M devices. Before showing these results, we perform additional sanity checks by examining how often individual features of each IP matched those in the most-likely OS suggested by Hershel.

Starting with the first two columns of Table 12, observe that window size is quite volatile, with 30% of the decisions going to signatures with a different window. This was expected given the willingness of end-users to experiment with this field and the large amount of unique values seen earlier. Additionally, these 30% cover unknown devices whose RTOs and other features may match some OS in $\mathcal{D}$, but not the window size. Hershel remains robust in these cases and simply identifies the closest signature based on the available information. For example, 98.4% of Ubuntu cases with the unknown window 5760 are classified to Linux 2.4/2.6. These 2.6M hosts account for 1/4 of all window mismatch.

TTL and DF both exhibit match rates over 95%, while MSS comes in much lower at 71%. This is not surprising in light of its dependency on the MTU. The OPT string proves extremely reliable, where 77.4% of the cases match exactly and 22% are feasible subsets/supersets of the original. The five possible cases with RST packets are shown in the other two columns of Table 12. Combining the first two rows, we can conclude that 91% of the hosts have a matching RST feature. The next row with missing RSTs allows us to ballpark network packet loss at q_{real} = 4.2%, not too far from the model's 3.8%. The majority of non-matching combinations (RA, RN, RW), responsible for 3.5% in the table, are caused by RW. Some of this behavior was also expected since user tweaking of window size causes some OSes to alter RW as well. Finally, we see 1.0% of the cases with an extra RST packet, which we suspect are injected by firewalls, NAT boxes, and other devices as indication that they have expired the per-flow state.

OS	Count
Linux 2.6 / 2.4	9,610,732
VxWorks embedded systems	4,179,583
Windows Server 2003 SP1 SP2	2,316,590
VxWorks 5.4.2 / Xerox embedded	1,890,585
Linux 2.6 / Debian / CentOS / SonicWall	1,196,143
Embedded Linux / Mikrotik routers	1,190,102
Windows Server 2008 SP1 SP2 R2 / Vista / 7	1,146,609
TP-Link / Iball / Huawei home routers	1,046,985
Windows Server 2003 / 2000 / XP SP1	1,001,343
Cisco / Scientific Atlanta cable modems	827,285

Table 13: Top individual signatures.

Having verified the general soundness of Hershel's output, we show it in Table 13. Linux attracts the most classification decisions, accounting for nearly a quarter of the webservers. This signature is quite unique, which makes accidental lumping of unknown devices or misclassified hosts into this category highly unlikely. In second and fourth place is VxWorks, which is an embedded OS extensively used in routers, modems, cameras, and printers. Interestingly, Windows 2003 is third, well above Server 2008 in seventh position. More Linux, home routers/modems, and Server 2003/XP make up the remaining OSes.

Table 14 groups fingerprints by type. Linux not just takes the first spot, but it dominates all other types of unix combined by a factor of 6. Embedded systems continue in second place, while windows is firmly in third. Interestingly, these results differ quite a bit from those in prior application of Snacktime to this dataset [23], with the most noticeable difference being 9M hosts moving from windows to embedded. This is not surprising as Snacktime's ability to overcome noise, packet loss, and feature corruption is quite weak. Further, as shown above, Microsoft OSes often share the window size and TTL with embedded devices, making this distinction even more difficult for Snacktime.

To shed additional light on this issue, we carry out comparison between the two methods using manual analysis of 1000 random targets. Table 15 shows the result. The first category in the table breaks down 429 hosts on which both methods produce the same exact OS. Out of these, 424 are correct matches, 3 incorrect, and 2 indeterminate. The last option occurs for devices inadequately represented in the database (i.e., no resemblance to any signature) or when multiple OSes appear to be probable. Among 571 disputed hosts, Hershel delivers 476 correct results and Snacktime 9.

Out of the 918 cases for which we can make a decision, Hershel's accuracy is 98% and Snacktime's is 47%. The 9 cases where Hershel is wrong, but Snacktime is right, are caused by bogus RSTs, which Snacktime ignores, but Hershel takes into account. Overall, we find that when the two methods disagree, Hershel is overwhelmingly more accurate.

7. CONCLUSION

We modeled the problem of single-packet OS fingerprinting and developed novel approaches for tackling delay jitter, packet loss, and user modification to SYN-ACK features. Based on this theory, we developed a classification method that significantly increased the accuracy of existing techniques, both in simulation and the real Internet.

Future work involves multi-pass extraction of the jitter distribution, packet loss probability, and OS popularity from the observed samples, which should improve estimation accuracy even further. We also plan to develop better result-verification techniques for wide-scale use.

Group	Count
Linux	13,882,999
Embedded	13,590,803
Windows	7,561,839
Other	2,396,455

Table 14: Common families of operating systems.

Category	Result	Count	Total
Consensus	Both correct	424	
	Neither correct	3	
	Indeterminate	2	429
Disagreement	Hershel correct	476	
	Snacktime correct	9	
	Neither correct	6	
	Indeterminate	80	571

Table 15: Manual verification.

8. REFERENCES

[1] Akamai. [Online]. Available: http://www.akamai.com/html/about/facts_figures.html.

[2] P. Auffret, "SinFP, Unification of Active and Passive Operating System Fingerprinting," *Journal in Computer Virology*, vol. 6, no. 3, pp. 197–205, Nov. 2010.

[3] T. Beardsley, "Snacktime: A Perl Solution for Remote OS Fingerprinting," Jun. 2003. [Online]. Available: http://www.planb-security.net/wp/snacktime.html.

[4] R. Beck, "Passive-Aggressive Resistance: OS Fingerprint Evasion," *Linux Journal*, vol. 2001, no. 89, Aug. 2001.

[5] D. B. Berrueta, "A Practical Approach for Defeating Nmap OS-Fingerprinting," 2003. [Online]. Available: http://nmap.org/misc/defeat-nmap-osdetect.html.

[6] R. Beverly, "A Robust Classifier for Passive TCP/IP Fingerprinting," in *Proc. PAM*, Apr. 2004, p. 158.

[7] R. Braden, "Requirements for Internet Hosts – Communication Layers," *IETF RFC 1122*, Oct. 1989.

[8] J. Caballero, S. Venkataraman, P. Poosankam, M. G. Kang, D. Song, and A. Blum, "FiG: Automatic Fingerprint Generation," in *Proc. NDSS*, Feb. 2007, pp. 27–42.

[9] H. K. J. Chu, "Tuning TCP Parameters for the 21st Century," Jul. 2009. [Online]. Available: http://www.ietf.org/proceedings/75/slides/tcpm-1.pdf.

[10] A. Crenshaw, "OSfuscate," 2008. [Online]. Available: http://www.irongeek.com/i.php?page=security/code.

[11] D. Dagon, N. Provos, C. P. Lee, and W. Lee, "Corrupted DNS Resolution Paths: The Rise of a Malicious Resolution Authority," in *Proc. NDSS*, Feb. 2008.

[12] T. Dunigan, M. Mathis, and B. Tierney, "A TCP Tuning Daemon," in *Proc. ACM/IEEE Supercomputing*, Nov. 2002, pp. 1–16.

[13] Z. Durumeric, E. Wustrow, and J. Halderman, "ZMap: Fast Internet-wide scanning and its security applications," in *Proc. USENIX Security*, Aug. 2013, pp. 605–620.

[14] P. Garcia-Laencina, J.-L. Sancho-Gomez, and A. Figueiras-Vidal, "Pattern Classification with Missing Data: A Review," *Neural Computing and Applications*, vol. 19, no. 2, pp. 263–282, Mar. 2010.

[15] L. G. Greenwald and T. J. Thomas, "Toward undetected operating system fingerprinting," in *Proc. USENIX WOOT*, Aug. 2007, pp. 1–10.

[16] S. Guoqiang and D. Lee, "Network Protocol System Fingerprinting: A Formal Approach," in *Proc. IEEE INFOCOM*, Apr. 2006, pp. 1–12.

[17] J. Heidemann, Y. Pradkin, R. Govindan, C. Papadopoulos, G. Bartlett, and J. Bannister, "Census and Survey of the Visible Internet," in *Proc. ACM IMC*, Oct. 2008, pp. 169–182.

[18] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is It Still Possible to Extend TCP?" in *Proc. ACM IMC*, Nov. 2011, pp. 181–194.

[19] IRL Fingerprinting Dataset. [Online]. Available: http://irl.cs.tamu.edu/projects/sampling/.

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

[21] T. Kohno, A. Broido, and K. C. Claffy, "Remote physical device fingerprinting," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 2, pp. 93–108, May 2005.

[22] E. Kollmann, "Chatter on the Wire: A Look at DHCP Traffic." [Online]. Available: http://myweb.cableone.net/xnih/download/chatter-dhcp.pdf.

[23] D. Leonard and D. Loguinov, "Demystifying Service Discovery: Implementing an Internet-Wide Scanner," in *Proc. ACM IMC*, Nov. 2010, pp. 109–122.

[24] J. Medeiros, A. Brito, and P. Pires, "A Data Mining Based Analysis of Nmap Operating System Fingerprint Database," in *Proc. IEEE CISIS*, Sep. 2009, pp. 1–8.

[25] J. Medeiros, A. Brito, and P. Pires, "An Effective TCP/IP Fingerprinting Technique Based on Strange Attractors Classification," in *Proc. DPM/SETOP*, Sep. 2009, pp. 208–221.

[26] Microsoft Support. [Online]. Available: http://support.microsoft.com/kb/2525390.

[27] D. Napier, "IPTables/NetFilter – Linux's Next Generation Stateful Packet Filter," *SysAdmin Magazine*, vol. 10, pp. 8–16, Nov. 2001.

[28] Netcraft Web Server Survey. [Online]. Available: http://news.netcraft.com/.

[29] Nmap. [Online]. Available: http://nmap.org/.

[30] Oracle, "Operating System Tuning." [Online]. Available: http://docs.oracle.com/cd/E12839_01/web.1111/e13814/os_tuning.htm.

[31] J. Padhye and S. Floyd, "On Inferring TCP Behavior," in *Proc. ACM SIGCOMM*, Aug. 2001, pp. 287–298.

[32] V. Paxson, M. Allman, J. Chu, and M. Sargent, "Computing TCP's Retransmission Timer," *IETF RFC 6298*, Jun. 2011.

[33] J. Postel, "Transmission Control Protocol," *IETF RFC 793*, Sep. 1981.

[34] G. Prigent, F. Vichot, and F. Harrouet, "IpMorph: fingerprinting spoofing unification," *Journal in Computer Virology*, vol. 6, no. 4, pp. 329–342, Nov. 2010.

[35] N. Provos, "A Virtual Honeypot Framework," in *Proc. USENIX Security*, Aug. 2004, pp. 1–14.

[36] N. Provos and P. Honeyman, "ScanSSH - Scanning the Internet for SSH Servers," in *Proc. USENIX LISA*, Dec. 2001, pp. 25–30.

[37] Y. Pryadkin, R. Lindell, J. Bannister, and R. Govindan, "An Empirical Evaluation of IP Address Space Occupancy," USC/ISI, Tech. Rep. ISI-TR-2004-598, Nov. 2004.

[38] D. Richardson, S. Gribble, and T. Kohno, "The Limits of Automatic OS Fingerprint Generation," in *Proc. ACM AISec*, Oct 2010, pp. 24–34.

[39] M. Roesch, "Snort – Lightweight Intrusion Detection for Networks," in *Proc. USENIX LISA*, Nov. 1999, pp. 229–238.

[40] G. Roualland and J.-M. Saffroy, "IP Personality." [Online]. Available: http://ippersonality.sourceforge.net/.

[41] C. Sarraute and J. Burroni, "Using Neural Networks to Improve Classical Operating System Fingerprinting Techniques," *Electronic Journal of SADIO*, vol. 8, no. 1, Mar. 2008.

[42] S. Shah, "An Introduction to HTTP Fingerprinting," May 2004. [Online]. Available: http://net-square.com/httprint_paper.html.

[43] M. Smart, G. R. Malan, and F. Jahanian, "Defeating TCP/IP Stack Fingerprinting," in *Proc. USENIX Security*, Jun. 2000, pp. 229–240.

[44] Snort IDS. [Online]. Available: http://www.snort.org.

[45] G. Taleck, "Ambiguity Resolution via Passive OS Fingerprinting," in *Proc. RAID*, Sep. 2003, pp. 192–206.

[46] G. Taleck, "SYNSCAN: Towards Complete TCP/IP Fingerprinting," *CanSecWest*, Apr. 2004.

[47] S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, 4th ed. Academic Press, 2009.

[48] B. Tierney, "TCP Tuning Guide for Distributed Applications on Wide Area Networks," *USENIX & SAGE Login*, vol. 26, no. 1, pp. 33–39, Feb. 2001.

[49] C. Valli, "Honeyd – A OS Fingerprinting Artifice," in *Proc. Australian Computer, Network and Information Forensics Conference*, Nov. 2003.

[50] F. Veysset, O. Courtay, O. Heen, and I. R. Team, "New Tool and Technique for Remote Operating System Fingerprinting," Apr. 2002. [Online]. Available: http://www.ouah.org/ring-full-paper.pdf.

[51] K. Wang, "Frustrating OS Fingerprinting with Morph," 2004. [Online]. Available: http://hackerpoetry.com/images/defcon-12/dc-12-presentations/Wang/dc-12-wang.pdf.

[52] F. V. Yarochkin, O. Arkin, M. Kydyraliev, S.-Y. Dai, Y. Huang, and S.-Y. Kuo, "Xprobe2++: Low Volume Remote Network Information Gathering Tool," in *Proc. IEEE/IFIP DSN*, Jun. 2009, pp. 205–210.

[53] M. Zalewski, "Strange Attractors and TCP/IP Sequence Number Analysis," Apr. 2001. [Online]. Available: http://lcamtuf.coredump.cx/newtcp/.

[54] M. Zalewski, "p0f v3: Passive Fingerprinter," 2012. [Online]. Available: http://lcamtuf.coredump.cx/p0f3.