# The Limits of Automatic OS Fingerprint Generation

David W. Richardson, Steven D. Gribble, and Tadayoshi Kohno
Department of Computer Science & Engineering, University of Washington
Seattle, Washington, USA
{daverich, gribble, yoshi}@cs.washington.edu

## ABSTRACT

Remote operating system fingerprinting relies on implementation differences between OSs to identify the specific variant executing on a remote host. Because these differences can be subtle and difficult to find, most fingerprinting tools require expert manual effort to construct discriminative fingerprints and classification models. In prior work, Caballero et al. proposed a promising technique to eliminate manual intervention: the automatic generation of fingerprints using an approach similar to fuzz testing [6]. Their work evaluated the technique in a small-scale, carefully controlled test environment. In this paper, we re-examine automatic OS fingerprinting in a more challenging large-scale scenario to better understand the viability of the technique. In contrast to the prior work, we find that automatic fingerprint generation suffers from several limitations and technical hurdles that can limit its effectiveness, particularly in more demanding, realistic environments.

We use machine learning algorithms from the well-known Weka [11] data mining toolkit to automatically generate fingerprints over 329 different machine instances, and we compare the accuracy of our automatically generated fingerprints to Nmap. Our results suggest that overfitting to non-OS-specific behavioral differences, the indistinguishability of different OS variants, the biasing of an automatic tool to the makeup of the training data, and the lack of ability of an automatic tool to exploit protocol and software semantics significantly limit the usefulness of this technique in practice. Automatic techniques can help identify candidate signatures, but our results suggest that manual expertise will remain an integral part of fingerprint generation.

**Categories and Subject Descriptors:** D.4 [Operating Systems]: Security and Protection; D.4.4 [Operating Systems]: Communications Management—network communication; C.2 [Computer-Communication Networks]: Network Protocols

**General Terms:** Design, performance security

**Keywords:** Fingerprinting, machine learning, fuzz testing, active learning, classification, automatic fingerprint generation

## 1. INTRODUCTION

Remote fingerprinting exploits differences between multiple implementations of the same software to identify the software variant executing on a remote host. For example, Nmap [21] uses fingerprinting to identify a remote host's operating system. A fingerprint consists of a set of network queries and a classification model. A fingerprinting tool issues the queries against the remote host, typically by sending carefully crafted network packets, collects response packets, and feeds these responses into the classification model. If the different implementations of the software generate predictably different query responses, the classification model can use the responses to identify the version of the remote software.

Fingerprinting tools typically rely on the manual effort of experts to construct discriminative queries and accurate classification models. As more variants of software propagate and as specific software evolves over time, the effort needed to keep a fingerprint database up-to-date can be prohibitive. In prior work, Caballero et al. proposed the idea of automatic fingerprint generation [6]. Their system, called FiG, uses an approach similar in spirit to fuzz testing to automatically generate candidate queries, execute them against a pool of known test machines, identify useful queries, and derive a classification model using two simple machine learning techniques. They demonstrated the promise of this approach by showing that FiG could reliably differentiate between three specific operating system versions (Windows XP SP2, Linux 2.6.11, and Solaris 9) and between five DNS server implementations.

Though their work presented an important first step, their evaluation did not explore more extensive, fine-grained system diversity or the practical difficulties that fingerprinting tools must overcome in realistic deployments. In this paper, we take the next step by considering whether automatically generated fingerprinting tools are viable in the more challenging environments that fingerprinting tools can face in practice.

We focus on operating system (OS) fingerprinting, specifically *OS classification*, which attempts to identify the OS version running on a remote host. We test whether machine learning algorithms from the well-known Weka [11] data mining toolkit can generate fingerprinting tools that can differentiate between a large number of machine instances in a pool that exhibits both coarse-grained differences, such as Linux vs. Microsoft Windows, as well as fine-grained differences, such as two machines with differing versions of the Linux kernel.

To do this, we implemented a platform that let us run automatic fingerprinting experiments at scale on our own local cluster, as well as within Amazon.com's elastic compute cloud (EC2). Our experiments take advantage of diverse, community-supplied machine instances, including virtual appliances gathered from VMware's marketplace and machine instances from Amazon's instance library.

Using 329 different instances, we evaluated the accuracy of our automatically generated fingerprints for a range of different classification granularities, and we compared their accuracy to Nmap. Finally, we examined the queries and classification rules generated by our automatic tools to determine whether the fields and rules used correspond to true differences in OS implementation, or were erroneous and inappropriate.

Interestingly, our results are mostly negative: somewhat contrary to the early evidence in the prior work, we found that automatic fingerprinting faces significant technical hurdles in practice. Specifically, four issues confounded our ability to automatically generate accurate OS fingerprints at scale:

1. **Overfitting:** response packets can vary for reasons besides differences in OSs' implementations. As examples, the value of TCP window scale option field may depend on how much memory is installed in a machine, the initial TCP sequence number selected by a remote host is typically non-deterministic, and network packet loss may cause some response packets to be lost. To be accurate, fingerprints' classification models must exclude differences caused by these extraneous, non-OS factors, but this is difficult to do automatically, particularly when considering a large pool of training and test machines.

2. **Training bias:** the learning algorithms we used in the construction of classification models tend to latch on to biases in the distribution of hosts in our training pool. If the training set is non-representative of the general population, then the generated tool's accuracy during testing might be very different than in deployment.

3. **Indistinguishablity:** when we try to classify hosts that have fine-grained implementation differences, such Linux kernel versions 2.6.15 and 2.6.22, it is hard to find queries that "tickle" their implementation differences to produce varying responses. As the classification granularity gets more fine-grained, it becomes much more likely that any differences observed in response packets are caused by extraneous factors that lead to overfitting, rather than true operating system implementation differences.

4. **Lack of semantics:** automatic fingerprint generation tools are limited to a mechanical view of packet field, whereas manually derived fingerprints can take advantage of human expertise to understand the deeper semantics of network protocols and packet fields. As such, tools like Nmap can use complex, multi-packet queries, and they can extract attributes out of multiple response packets, such as the greatest common divisor of differences between initial sequence numbers. We expect that it would be challenging to engineer automatic fingerprinting tools that are able to "understand" enough to derive such complex queries and response packet attributes, yet these are some of the most useful.

While an automatically generated fingerprinting tool can be competitive with Nmap at solving OS classification for coarse granularity buckets (Windows vs. Linux) and small machine pools, at fine granularity (Linux 2.6.15 vs. 2.6.22) and large scale, our results suggest that automatic tools fall prey to overfitting and biases in the training data. Automatic tools are prone to confusing unlucky coincidences with stable, genuinely discriminative classification rules and probe packets. At scale and with fine-grained machine differences, such rare coincidences are more likely to be found than useful probes and rules, and our results suggest that manual expertise may ultimately necessary to distinguish between them.

## 2. OVERVIEW

Though fingerprinting applies to a broad range of network software, in this paper we focus on OS fingerprinting as it applies to the well established **OS classification** application. The goal of OS classification is to identify the OS executing on an arbitrary remote machine. To accomplish this, an OS classification tool first constructs a predictive model by probing a set of machines with known operating systems installed on them and finding differentiating features within probe responses. To classify an unknown machine, probe responses from that machine are fed into the constructed model, which outputs its best guess for the unknown machine's operating system.

With OS classification, the set of known machines used to construct the model is referred to as the training set. Training set machines are selected so that they contain a diverse but representative sampling of OSs found in the wild. Probes consist of carefully constructed TCP/IP packets designed to tickle implementation differences between the OSs' network stacks. For OS classification to work well, the probes and predictive model must be constructed to cause different OSs to generate observably different response packet features, but these features must also be stable within different instances of the same OS.

A robust OS classification tool is useful to network administrators and adversaries. For the former, the tool allows them to monitor and enforce policies on types, versions, and patch levels of machines within a network. For the latter, the tool can help identify potential vulnerabilities to exploit in target machines.

OS classification becomes progressively harder at scale. As the number of OS classes to distinguish grows, it becomes more likely that a tool will confuse one minor OS kernel revision for another.

### 2.1 Nmap

One popular remote operating system fingerprinting tool is called Nmap [21]. Nmap's designers have devised 16 hand-crafted network probe packets aimed at eliciting responses containing kernel-specific differences. After probing a machine, Nmap extracts and condenses response features into a summary data structure that is fed into Nmap's classification model.

Nmap was designed to solve the OS classification problem: its classification model consists of a database of thousands of reference summary data structures for known OSs. When an unknown subject machine is probed, its response summary structure is matched against this database, and the OS with the closest matching reference summary in the database is returned. Nmap measures closeness using a set of hand-tuned heuristics, assigning weights to specific summary features and summing up the weights of features that the unknown subject has in common with a given reference record in the database. Because new OS variants and versions are routinely coming online, Nmap's developers face the constant battle of finding new probes and re-examining existing ones to keep the Nmap classification database up-to-date.

### 2.2 Automatic fingerprint generation

Caballero's prior work hypothesizes that the limitations of manual fingerprint generation can be overcome by automatically generating and testing candidate fingerprints, and training a discriminative signature set and accurate classification model [6]. In this way, automatic fingerprint generation is similar to fuzz testing: a large number of network queries are automatically generated and those queries are played against a set of training machines with known ground truth. Simple machine learning is then used to construct a predictive model that reflects observed differences and hopefully generalizes beyond the training set.

In this work, we uncover and examine the challenges with automatic fingerprinting that arise at scale. We focus on two issues. First, when considering a large and diverse population of machines, OS versions, and networks, we will inevitably encounter many practical challenges in collecting accurate and reliable data such as packet loss and non-deterministic data. Can we overcome these problems, and if so, to what degree do our solutions impact fingerprint generation accuracy? Second, classification models must hone in on behavioral differences between operating systems, but to generalize beyond the training set, they must avoid overfitting to sources of behavioral differences other than the OS implementation. Is this possible to do using modern machine learning algorithms, and how well do the resulting tools perform compared to manually generated tools like Nmap?

### 2.2.1 Practical challenges with data collection

A serious limitation of automatic fingerprinting tools is that they have no semantic knowledge about the various TCP and IP fields in network packets. Fields are treated as simple typed attribute/value pairs. This means that a tool is unable to take advantage of the many inter-field and inter-packet relationships that exist in the TCP and IP specifications. This affects a tool's ability to intelligently craft probe packets that are likely to trigger OS-specific implementation differences in a remote host, and its ability to analyze the probes' responses to find more complex relationships that can be used to identify OS variants.

Automatic tools compensate for this lack of semantic knowledge by generating a large number of random probe packets and by using machine learning algorithms on the responses to learn discriminating features. However, we must carefully weigh the tradeoffs between sending too many and too few probe packets: too few probes risks missing valuable response packets, while too many probes can be impractical and make it more likely the tool will encounter noise in the data that confounds the machine learning process.

Another challenge is that some response packets will contain non-deterministic values. Non-determinism can be introduced by the remote host, or by the network itself when packets are dropped, reordered, or modified by middleboxes such as routers or firewalls. In the simplest case, if we see no response from a probe, we cannot be sure if this is because the remote host chose not to respond to it, or because of temporary network or environmental conditions.

### 2.2.2 Challenges with classification models

Once a large number of probe and response pairs are collected from a set of training machines labeled with ground-truth OS versions, the next step in automatic fingerprint generation is to use machine learning techniques to construct a classification model. An ideal model for OS classification would select features that are consistent between instances of the same OS, but which have observable differences between instances of different OSs. The model must hone in on deterministic behavioral differences that are attributable to OS's source code itself. Unfortunately, there are many potential sources of behavioral difference beyond this. In our experiments, we have encountered the following:

**Non-determinism**: non-determinism in the network, operating system, or applications that respond to packets can lead to observable differences. For example, if an OS selects an initial sequence number at random, or is configured to use port knocking, this can lead to observable non-determinism.

**Hidden state**: there are complex state machines within the operating system and applications that can lead to differing behavior over time. For example, OSs that are not configured to use SYN cookies can be vulnerable to SYN flood attacks that cause a SYN queue to fill up, potentially leading to an observable difference dependent on the packet sequence leading up to the probe itself.

**Network and middleboxes**: the nature of the network and network components between the probe machine and remote host can introduce behavioral differences. For example, middleboxes might rewrite fields while routers could modify explicit congestion notification fields during periods of congestion.

**Host hardware**: the hardware on which an OS runs can directly or indirectly affect behavior. For example, the value of the TCP window scale options field selected by a remote host can be affected by the amount of RAM present on the machine.

**Applications**: applications can modify the network behavior of a host, for example, by setting socket variables using the equivalent of Linux's `setsockopt` API.

**System configuration**: even if two hosts are running identical OSs, the machines might be configured differently, leading to differing behavior. For example, a machine might have its firewall enabled, causing it to not respond to some probe packets; as another example, a Linux machine might be configured to change default send or receive window sizes via the `/proc` interface.

**OS source code**: differences in the source code between OS versions can lead to different behavior.

To solve the OS classification problem, a classification model must contain attribute/value pairs whose observed differences are attributable only to operating system source code. If any other kinds of differences pollute the model, then the fingerprinting tool is susceptible to overfitting and potentially will fail to generalize accurately beyond the training data.

Some of these categories can be eliminated with sufficient data; for example, non-determinism can be identified probabilistically by generating multiple, identical probe packets and looking for varying responses. Other categories are much harder to identify automatically, such as hardware or system configuration differences. To be robust against these, the set of training machines must contain instances that reflect exactly these hardware and configuration differences, so that the model generation process can discriminate against them. In fact, if the set of training machines is itself biased to particular configurations and machines, this bias can sneak into the classification models and also affect their accuracy. In essence, the set of training machines must contain as much heterogeneity as the general population itself or it will be prone to overfitting.

Another challenge of classification arises if there are non-representative biases in the training set. Some learning algorithms use the likelihood that a class was encountered during training to weight the probability that a given test instance is a member of that class. If the training set contains a distribution of classes in its machine population than are not seen in the real world, the generated tool's accuracy in the wild might differ substantially than its accuracy during testing.

## 3. IMPLEMENTATION AND TESTBED

In this section, we describe the design and implementation of our automatic fingerprint generation tool and the data sets we use for evaluation. Our tool is composed of a set of modules: a *probe generator* generates a large number of candidate network probes; a *data collector* transmits probes to a machine set and gathers associated response packets; and a *learner* uses training data and machine learning algorithms to generate a fingerprinting tool, including its classification model and final probe set. Figure 1 shows the flow

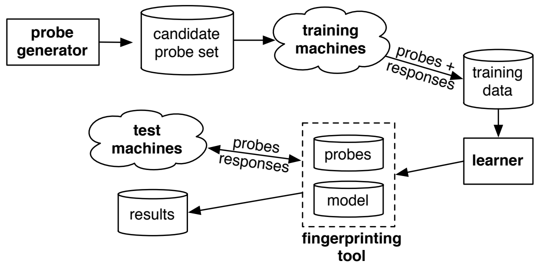


**Figure 1: Experimental data flow. This figure shows how data flows in our platform during training and testing.**

of data through our experimental system. We built our system in Python, but with some use of C to speed up critical components.

## 3.1 Experimental machines

We collected 329 unique virtual machine instances from a variety of sources, including Amazon EC2 instances, VMware appliances, and distributions from Linux websites. Though we emphasized Linux variants with differing kernel versions, we also gathered some Windows, BSD, and Solaris machine instances. For each instance, we manually produced a ground-truth label and examined the machine to determine a suitable open TCP port to probe. For most non-Windows machines, we used port 22 (ssh), and for Windows, we primarily used port 3389 (RDP).

We expect that a fingerprinting tool will be able to easily distinguish between major OS categories, such as Windows and Linux, but it will have a more difficult time classifying minor differences between OS instances, such as Linux kernel versions 2.2.10 and 2.2.20 or the Ubuntu and Redhat distributions. To let us experiment with this tradeoff between accuracy and fidelity, each of our ground-truth labels actually reflects two different hierarchies of classification. The first hierarchy contains three different levels of OS kernel granularity, and the second hierarchy contains three different levels of OS distribution granularity. Our goal is not for these 329 VM instances to approximate the distribution of OS variants found on the Internet. Rather, by having both hierarchies, we can test whether a fingerprinting tool is more likely to find behavioral differences attributable to kernel differences or to OS configuration and packaging differences.

Table 1 shows the labels we assigned to our machine instances in each of the two classification hierarchies, and the number of machine instances to which we assigned each label. Note that our machine instances are skewed towards Linux. This ensures that we have enough variety of Linux machines to compare the accuracy of fingerprinting at fine-granularity and coarse-granularity classification levels. However, as we will see later, the population bias that it introduces has negative effects on our fingerprinting tools.

## 3.2 Probe generator and data collector

The probe generator produces a large set of non-fragmented network packets by assigning randomly generated values to various IP and TCP fields, subject to the constraints that packets must be well-constructed and routable to a target machine. To avoid overwhelming the system with undeliverable packets, we did not experiment with adding IP options, though we did include randomly generated TCP options. To help detect non-deterministic values in responses, the data collector transmits each probe multiple times. In Section 4.1, we describe our experiments that explore how many times each probe should be sent. For a detailed explanation of the IP and TCP fields we used, as well as an example of a probe packet

| level | label *(# machines with label)* |
|---|---|
| 0 | Linux *(314)* Windows *(12)* BSD *(2)* Solaris *(1)* |
| 1 | 2.6 *(312)* XP *(6)* winserver *(6)* others *(6)* |
| 2 | 2.6.15 *(9)* 2.6.16 *(74)* 2.6.18 *(17)* 2.6.21 *(189)* 2.6.22 *(5)* 2.6.24 *(7)* XP *(6)* winserver *(6)* others *(16)* |

**(a) OS kernel classification hierarchy**

| level | label *(# machines with label)* |
|---|---|
| 0 | Linux *(314)* Windows *(12)* BSD *(2)* Solaris *(1)* |
| 1 | ubuntu *(185)* fedora *(53)* debian *(47)* gentoo *(16)* centos *(12)* redhat *(4)* XP *(6)* winserver *(6)* freebsd *(2)* opensolaris *(1)* |
| 2 | ubuntu6.06 *(24)* ubuntu6.10 *(4)* ubuntu7.04 *(13)* ubuntu7.10 *(54)* ubuntu8.04 *(62)* ubuntu8.10 *(28)* fedora4 *(9)* fedora5 *(8)* fedora6 *(8)* fedora8 *(18)* fedora9 *(4)* debian4.0 *(26)* debian5.0 *(18)* gentoo *(16)* centos5.0 *(5)* centos5.2 *(4)* XP *(6)* winserver2003 *(6)* others *(16)* |

**(b) OS distribution classification hierarchy**

**Table 1: Machine instance classes. This table shows the labels we assign to machine instances, broken down into two different classification hierarchies. To save space, some rare labels are collapsed into an "others" field in the table, but are broken out in our actual experiments.**

and the resulting response packets from one of our training machines, we refer the reader to Appendix A.

## 3.3 Learners

Learners use training data gathered by the data collector to generate a fingerprinting tool, including both its classification model and probes. The key to understanding how a learner module works is to understand fingerprints, how they are used, and how they are implemented. We discuss each of these points below.

### *3.3.1 Fingerprints*

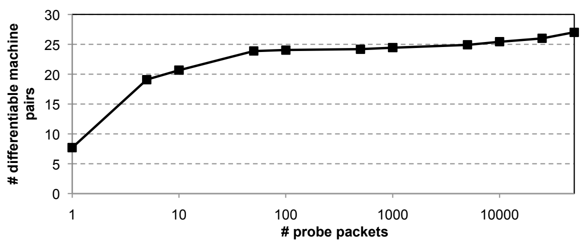
Consider a set $P$ of TCP/IP packets and a set of machines $M$, where each machine $m \in M$ has a known, labeled OS $OS(m)$. Having sent each packet $p \in P$ to every machine $m \in M$, the data collector records a response packet $R_p$ for each $p$, or null if no response is received. This yields a set of examples $E$ for the learner, where $E = (< p, R_p >, m)$ for $p \in P$ and $m \in M$. A learner $L$ takes as input the set of examples $E$ and produces a fingerprinting tool $f$. The tool $f$ takes as input an example $e$ and returns the best OS label for the example's machine $e(m)$. A tool is a function $f$ such that $f(e) = OS(e(m))$ for all $e \in E$.

To solve the OS classification problem, this tool $f$ should not only correctly return the OS of all examples in $E$, but it should also correctly return the OS of previously unencountered examples *not* in $E$. We experiment with classification accuracy at varying levels of OS granularity by directing our learners to use any of the different OS label levels from Table 1; in one experiment, we use course-grained labels (e.g., Windows vs. Linux), and in others, we can learn using finer-grained labels (e.g., Ubuntu vs. Debian).

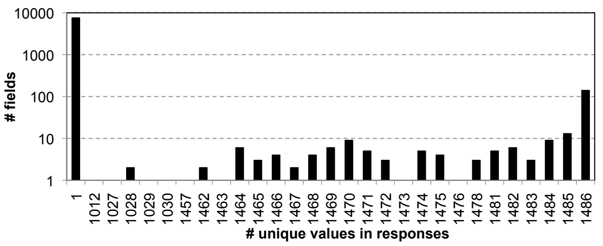
### *3.3.2 Model construction algorithms*

Many machine learning algorithms could be used to implement a learner $L$ and fingerprinting tool $f$. We used five algorithms implemented in the popular data mining toolkit Weka [11]:

**J48.** This algorithm is a clone of the C4.5 decision tree implementation by Ross Quinlan [17]. Decision trees are constructed by



(a) Effect of # probes on differentiability.



(b) Identifying determinism.

**Figure 2: Differentiability and determinism. Graph (a) shows the effect of increasing the number of probe packets on our ability to distinguish between 200 pairs of randomly chosen machines using only TCP or IP fields known to elicit deterministic, OS-specific differences. Histogram (b) shows the count of response fields that had a given number of unique values. Most response fields have a unique value, suggesting they are deterministic. Note that the x-axis is discontinuous.**

repeatedly partitioning the examples using the best attributes in the training data.

**JRip.** This algorithm implements a propositional rule learner based on repeated incremental pruning to produce error reduction (RIPPER). The generated rules are expressed as propositional logic equations of the attributes in the training data.

**RandomForest.** This algorithm constructs many decision trees and uses a form of consensus over the trees in order to label an example with a class value.

**SMO.** This algorithm is an implementation of *support vector machines*. SVMs attempt to represent the examples as points in space such that similar examples are grouped into the same region. New examples are classified by being mapping into the best fitting region of space.

**IBk.** This is a standard clustering algorithm using K-nearest neighbors. To classify a sample, the model finds the K-closest matching examples and returns the label held by the majority of the matches. In our analysis, we fix K to 3, resulting in 3-nearest neighbors for all of our experiments.

For comparison, we also use Nmap to classify machine instances. To make the comparison fair, we replaced Nmap's extensive classification database with one constructed using records gathered from our training sets. As we will explain in Section 4.3.1, we modified Nmap to emit a best, single guess when it would otherwise emit a list of labels between which it cannot differentiate.

### 3.4 Evaluation method

Given a fingerprinting tool produced by a learner, we run experiments to evaluate how effective that tool is at solving the OS classification problem. To do this, we use the standard method of splitting our machine set into two disjoint sets of training machines and test machines. Data gathered from the training machines is used by the learner to produce a tool $f$, and the tool $f$ is then used to classify machines in the test set. In all of our experiments, the probing and probed machine run on the same subnet.

## 4. RESULTS

### 4.1 Overcoming data collection challenges

We now evaluate the accuracy of automatically generated fingerprints, focusing on three questions. First, how difficult is it to overcome the practical challenges of clean data collection? Second, in our classification models, how often do we encounter overfitting to non-OS source code behavioral differences? Third, how accurate are automatically generated fingerprints at different classification granularities, and how does this compare to Nmap?

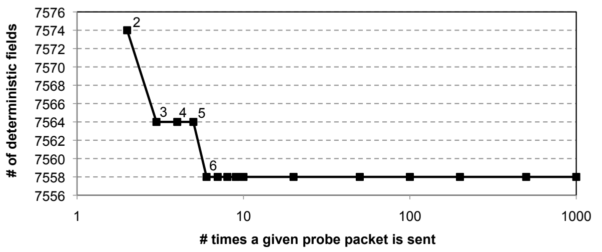
To be able to generate accurate models and tools, the data we collect should satisfy two criterion: the data must be *useful*, meaning that it contains enough differences for a model to distinguish between OSs, and it must be *consistent*, meaning that the data is repeatable across separate experimental runs. Our experiments show that there are three practical challenges any automatic fingerprinting tool must overcome to obtain useful and consistent data: (1) choosing the number of training packets to send, (2) eliminating non-determinism in the collected data, and (3) coping with packet loss when conducting experiments.

**Choosing the number of packets.** During training, collecting *useful* data requires sending an adequate number of probe packets. The more probe packets that are sent, the more likely the system is to obtain discriminative response data. However, sending too many packets is inefficient and risks introducing noise into the data. To show the relationship between the number of training packets sent and the amount of useful data gathered, we constructed an experiment using 200 randomly selected pairs of machine instances and 50,000 candidate probe packets. We count the number of machine pairs we could distinguish from each other, using only TCP or IP fields known to elicit deterministic, OS-specific differences, as a function of the number of probe packets sent. We then averaged this count over many randomly generated sets of probe packets of each size. Figure 2(a) shows the result of this analysis.

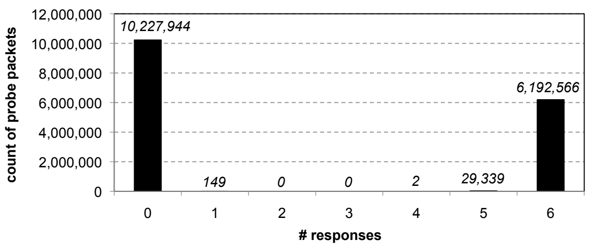
Not surprisingly, we found that, on average, our ability to distinguish between two randomly selected machines from our sample pool increased as we sent more probe packets: more data gives the classification models more input to work with. However, we also found that the rate of increased accuracy slows dramatically beyond roughly 100 packets, suggesting that very large packet numbers will only modestly increase accuracy at the risk of introducing noisy data into the models. We conservatively chose to use a set of 50,000 packets during the rest of our experiments.

**Eliminating non-determinism.** An obstacle to obtaining *consistent* data is non-determinism in response packet fields. To identify and eliminate non-deterministic fields during training, we used the simple approach of looking for changes in a field's value across successive probes. Instead of sending a probe to a target machine once, we send each probe multiple times, record the set of responses, and check that each field's value is consistent across all collected responses. A field whose value changes is excluded from consideration, and a field whose value remains stable is kept as a candidate for the learner.

To find a practical and efficient number of copies the data collector should send each probe to eliminate non-determinism, we selected at random 20 generated probe packets and 10 machines. Next, we sent each probe 1500 times and recorded the responses.



(a) Number of times a probe should be sent.



(b) Packet loss.

**Figure 3: Sending packets multiple times and packet loss. Graph (a) shows the effect of sending a probe packet multiple times on the number of non-deterministic fields discovered. Note that the graph's y-axis does not begin at zero. Histogram (b) counts the number of probe packets that received between 0 and 6 responses. Probes typically receive no responses, or all six, suggesting that packet loss is rare.**

Each response contains up to 39 separate fields, so we collected a total of 39 x 20 x 10 = 7800 fields, each with 1500 values. Not all machines returned responses for all 1500 probes; the smallest response set contained 1486 responses. This data lets us examine how hard it is to spot non-deterministic behavior by identifying fields that exhibit more than one value across the responses.

Figure 2(b) shows a histogram of the count of response fields that had a given number of unique values. The histogram is bimodal: most response fields have a single unique value, indicating that they are likely deterministic, while some fields have many unique values, suggesting that they are non-deterministic and that this non-determinism is easy to "provoke." Thus, sending each probe a relatively small number of times during training is sufficient in practice for uncovering non-determinism.

In Figure 3(a), we vary the number of times each probe is sent and show the number of fields that we declare to be deterministic. The results show that sending a probe six times is sufficient to uncover all non-determinism that we could identify in this small set of 20 probe packets and 10 machines. For the remainder of this section, we send each of our $50,000$ probe packets a total of 6 times per machine instance during training. However as we describe in Section 4.2.2 below, large-scale experiments still can uncover several rare, unlucky circumstances in which six probes are not sufficient to uncover non-deterministic behavior. At increasing scale, it is common to encounter such rarities!

**Coping with packet loss.** Our ability to find non-determinism using six transmissions of each probe packet is predicated on the assumption that probe and response packets are seldom dropped during transit. However, packet losses will occasionally occur. As well, since our probe packets contain randomly generated field values, many of our probes will be dropped by the remote host rather than eliciting a response. This lack of response is as valuable to our classification models as a valid response. However, it is impossible to distinguish between network-induced packet loss and packets dropped by the remote host. If we collect data on a lossy network, we run the risk of confusing dropped packets with deliberate lack of responses.

To understand the potential impact of packet loss on evaluation, we collected responses by sending all $50,000$ of our probe packets to all of our evaluation machines a total of six times each. For each set of six probes sent to a machine, we counted the number of response packets observed. Figure 3(b) shows a histogram of the number of packet responses received.

Figure 3(b) shows that in the vast majority cases, a machine either returns a response for all six probes, or it returns no responses at all. Occasionally, we observed single packet losses, and even more rarely, we observed a handful of multiple packet losses. These results are promising, since it means that a simple way to deal with potential packet loss is for our data collector to only consider probes that return all responses or none; other cases can be disregarded. In practice applying this technique in our carefully controlled networking environment resulted in us discarding just 121 out of 50,000 probe packets (0.24%).

## 4.2 Overfitting in the models

After having collected useful and consistent training data, the next challenge faced by an automatic fingerprinting tool is avoiding overfitting in the classification models. Overfitting tends to be introduced by fields whose behavior depends on something other than the source code of the OS itself.

### *4.2.1 What are the potential sources of overfitting?*

To better understand the kinds of overfitting that might occur in our models, for each response packet field in our probe data, we manually examined Linux and BSD source code to understand what factors could impact that field's value and to explore whether there were system call APIs, OS configuration files, and other sources of behavioral differences. Though it is difficult to be exhaustive, this characterization effort gave us a good sense of which fields, if used in our models, are likely to lead to overfitting.

Some fields were fairly straightforward to analyze and had unsurprising results. For example, the IP version field is primarily influenced by whether an OS is using IPv4 or IPv6; though it can also be influenced by OS configuration, by applications that select specific network interfaces, or by network middleboxes, in practice our tool restricts itself to sending IPv4 packets and thus we are very unlikely to encounter anything other than an IPv4 version field in a response.

Some fields were easy to analyze but had surprising results. For example, if a response packet contains the WScale TCP option, then the amount of memory on the remote host can influence the response packet's WScale value. This can be seen in the following snippet of source code from the Linux 2.6.19 kernel:

```
(*rcv_wscale) = 0;
if (wscale_ok){
   space = max_t(u32, sysctl_tcp_rmem[2],
                 sysctl_rmem_max);
   space = min_t(u32, space, *window_clamp);
   while (space>65535 && (*rcv_wscale)<14){
      space >>= 1;
      (*rcv_wscale)++;
   }
}
```

The value of the third element in the sysctl_tcp_rmem array is influenced by the amount of physical memory installed on a host. Hence, memory capacity, and thus hardware configuration, can influence the value of the WScale TCP option field.

We also ran into many ambiguous cases when deciding whether a source of influence ought to count towards a particular field's

|  | non-determinism | hidden state | network | hardware | applications | system configuration | OS source code |
|---|---|---|---|---|---|---|---|
| **IP field** |  |  |  |  |  |  |  |
| version |  |  | ● |  | ● | ● | x |
| hdrlen |  |  | ● |  | ● | ● | x |
| tos |  |  | ● |  | ● | ● | x |
| len |  |  | ● |  | ● | ● | x |
| id | x | x | ● |  |  |  | x |
| flags |  |  | ● |  |  |  | x |
| frag |  |  | ● |  |  |  | x |
| ttl | x |  | x |  | x | x | x |
| proto |  |  | ● |  |  |  | x |
| chksum | x | x | x | x | x | x | x |
| **TCP field** |  |  |  |  |  |  |  |
| seq | x | x | ● |  |  |  | x |
| ack |  |  | ● |  |  |  | x |
| dataofs |  |  | ● |  | ● | ● | x |
| reserved |  |  | ● |  |  |  | x |
| flags |  |  | ● |  | ● | ● | x |
| win size |  |  | ● | ● | ● | ● | x |
| chksum | x | x | x | x | x | x | x |
| urgptr |  |  | ● |  | ● | ● | x |
| op order |  |  | ● |  | ● | ● | x |
| op MSS |  |  | ● |  | x | x | x |
| opt wscale |  |  | ● | x | x | x | x |
| opt tsval | x | x | ● |  | ● | ● | x |

**Table 2:** **Sources of observable behavioral differences. This table classifies the sources of observable behavioral differences associated with TCP and IP fields.**

value. For example, even though an application can in principle affect the IP header length field by causing IP options to be added to a packet, in practice, we never encountered this. Thus, even though in principle the IP header length field can be influenced by an application or OS configuration, in practice it is usually safe for an OS classification tool to use this field.

Table 2 presents a summary of our detailed analysis. An "x" in a cell signifies that the indicated field's value is likely to be influenced by the specific category, while a dot in a cell signifies that it is hypothetically possible for the value to be influenced by the category, but it is very unlikely that this would be encountered in practice.

### *4.2.2 How much overfitting occurs in our models?*

Our next step is to evaluate the likelihood that overfitted fields will be used by classifiers in our model. To do this, we first quantify the value of each field in our probe data to the classifiers used by our fingerprinting tool. Because all classifiers share a common goal of identifying and using high-value fields to build their classification models, if we find that the majority of high-value fields in our probe data are prone to overfitting, then by extension we can conclude that any classifier we use will likely suffer from overfitting.

Intuitively, high-value fields discriminate between many different machines in the training data. For example, if we find that some field $F$ is always set to 0 for Windows machines and 1 for Linux machines, then $F$ has a high value in discriminating between our evaluation machines at the most general classification level. We employ a standard way of quantifying the "value" of a field by using *information gain*. The information gain of a field captures how useful knowing the field's value is when correctly classifying the

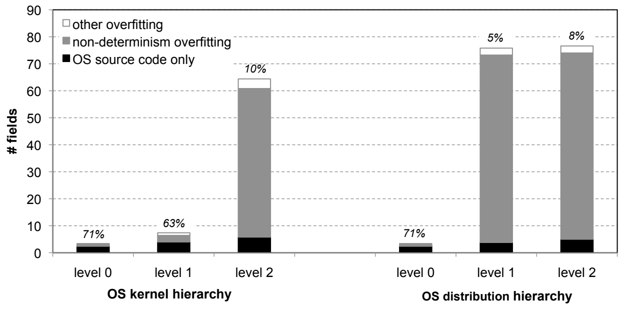


**Figure 4:** **Overfitting in OS classification. This figure shows the number of high-value fields that are prone to overfitting and therefore likely to be included in a tool's classification model, as a function of the label classification hierarchy and level used while training. We also show the number of non-overfitting-prone fields in the "OS source code" category; the italicized numbers show the percentage of fields used that fall in "OS source code."**

329 evaluation machines. The higher the information gain, the better the field can discriminate between machines.

Given our probe data, for each field we split the evaluation machines into subsets based on the value of that field. Information gain allows us to evaluate which field best splits the machines into subsets. The field with the highest information gain is selected and recorded, and we repeat this field selection and machine splitting process on its derived subsets. This process recurses until either all remaining subsets contain machines with the same classification, or no subset can be further subdivided. The resulting set of recorded fields constitutes the highest valued fields in our probe data.

Given a set of high-valued fields returned by our algorithm and the manually constructed set of overfitting sources (summarized in Table 2), we can calculate the distribution of high-value fields prone to overfitting versus those that are not. This lets us quantify how much overfitting we expect to occur in the models generated by our classifiers.

To perform this analysis, we randomly selected 263 of the 329 machine instances (80%) as a training set, and used our algorithm to construct the set of high-value fields in the training data gathered from all 50,000 probes sent to each instance six times. As well, so that we could explore the impact of classification granularity on overfitting, we repeated this process for each of the the label classification levels presented in Table 1.

For each high-value field produced by our algorithm, we refer to its sources of behavior differences from Table 2, and count a field as an instance of overfitting if any of its sources are other than "OS source code." Figure 4 shows our results. Every model falls prey to some overfitting, though the degree of overfitting is higher as the classification granularity becomes more fine-grained. Interestingly, there is less overfitting when using labels from the OS kernel hierarchy; an automatic tool is more able to find genuine differences between Linux kernel versions than between Linux distributions.

Despite our efforts to weed out non-deterministic fields by sending six repeats of a probe, many slipped through and polluted our models. At scale, we stumbled across unlucky coincidences that make non-deterministic fields seem deterministic. For example, one of our probe packets always elicits the value 0 in the IP ID response field from Windows XP machines, but non-deterministic values from BSD. When these responses happen to flow over the same number of network hops for each of the six probes, the IP TTL field also seems deterministic. In combination, the IP checksum field appears to have a deterministic value for Windows XP under this probe packet, despite the fact that IP checksums are actually non-deterministic. Thus, the checksum field appears to have high value, leading classifiers to mistakenly adopt a rule that classifies an instance as Windows XP if it observes responses with this "unlucky" checksum value.

Classifiers exacerbate the impact of misidentified non-deterministic fields. Because classification algorithms hone in on field value differences that appear to distinguish between OS versions, non-deterministic values tend to fool our learning process into mistakenly considering these fields as genuinely discriminative. The tools and models have no way to distinguish legitimate behavioral differences from rare, unlucky coincidences, and at scale and fine granularity, these rare coincidences seem to be more frequent than genuine, stable distinguishing criteria.

## 4.3 OS classification accuracy

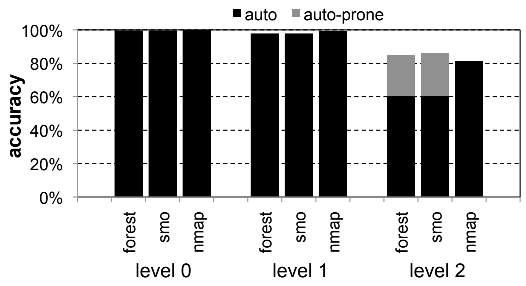
We now evaluate the accuracy of our automatically generated fingerprinting tools at solving the OS classification problem. In this subsection, we explore the limits of how well automatic tools can perform for OS classification if overfitting is eliminated as a confounding factor, and we compare that to tools that are allowed to fall prey to overfitting.

Given a test machine, a fingerprinting tool will emit a single label that it believes matches the test machine. We define a simple metric for evaluating the performance of a tool: the *accuracy* of a tool is the percentage of test machines for which a tool returned the correct OS label. We evaluate this metric for our fingerprinting tools by creating a learner using each of the 5 different classification model construction algorithms chosen from the Weka toolset. For each learner, we repeat all experiments three times: once when we allow our learner to use overfitting-prone fields, once when we constrain our learner to "good" fields that we manually vetted as not being prone to overfitting using Table 2, and once using Nmap.

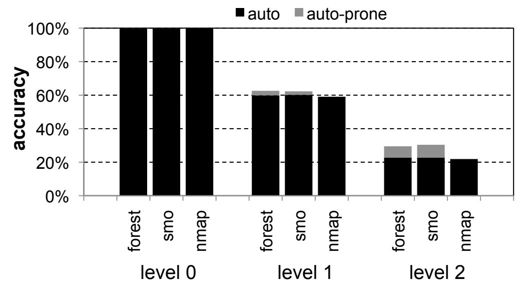
Each experiment averages our accuracy metric over a standard 10-fold cross validation of the evaluation machines. We randomly split the evaluation machines into 10 equally sized sets. For each of these 10 splits, we consider the machines in the first split to be test machines and the machines in the remaining 9 splits to be training machines. We then send all 6x50,000 probe packets to each of the training machines. The response packets are passed to the learner to construct the fingerprinting tool, and we record the percentage of correctly labeled test machines classified by the tool. The final accuracy metric for the tool is the result of averaging the individual accuracy percentages returned by each of the 10 folds.

We also repeated this learning and 10-fold cross-validation using each of the label classification levels from Table 1. Finally, we compared the accuracy of our automatically generated tools to that of Nmap version 4.6.9. As previously mentioned, to make the comparison fair, we replaced the extensive classification record database that ships with Nmap with one constructed using Nmap records gathered from the training machine sets.

Figure 5(a) shows our results for the kernel classification hierarchy and Figure 5(b) shows our results for the distribution classification hierarchy. For brevity, we only show results for the SMO and RandomForest learners and for Nmap. The remaining 3 learners performed comparatively. At coarse-grained levels of both classification hierarchies, all tools are quite accurate. At level 0, i.e., when classifying machines as Windows, Linux, BSD, or Solaris, our fully automatic tools are more than 98% correct, as are the same tools with overfitting manually removed. Even at level 1 in the kernel hierarchy, our tools are correct more than 97% of the time. This confirms the results in Caballero et al.'s prior work [6].



(a) Kernel classification hierarchy.



(b) Distribution classification hierarchy.

**Figure 5: OS classification accuracy. This chart shows the accuracy of our OS classification tools for 2 Weka learning algorithms: RandomForest and SMO. Nmap accuracy is also shown. Graph (a) shows accuracies for the kernel classification hierarchy, while graph (b) shows the accuracies for the distribution classification hierarchy. Two versions of our tools are considered: one where overfitting-prone fields are allowed (*auto-prone*), and one where overfitting-prone fields removed (*auto*).**

At finer granularity levels in both hierarchies, all tools begin to run into trouble. Overfitting causes our fully automatic tools to lose their ability to distinguish between some sets of machines. As a result, at level 1 of the distribution classification hierarchy our tools emit the correct label only 62% of the time. At level 2, accuracy drops to less than 30%.

At the kernel classification hierarchy, our fully automatic tools are correct 85% of the time at level 2. However, the equivalent tools with manually eliminated overfitting perform worse, with 60% accuracy. Similarly inflated accuracies exist for both classification hierarchies. These accuracy inflations show that fully automatic tools are using overfitted fields at finer granularities, resulting in overly optimistic accuracy. Nmap remains competitive with all of our tools, and it particularly shines at level 2 of the kernel classification hierarchy, a level of granularity for which Nmap is known to be well-suited. However, even Nmap cannot finding enough distinguishing features at the lowest classification levels.

### 4.3.1 *Effects of training bias*

Our previous analysis suggests that automatic fingerprinting tools perform as well as manually crafted tools like Nmap. However, this analysis did not consider the effects of training bias. Training bias can occur when a tool weights its classifications using the probability distribution of class labels that it encountered during training. In this subsection, we show that training bias in our data substantially affects the accuracy of our automatically generated tools.

To quantify the effect of training bias, we performed two experiments. In the first experiment, we randomly selected 50 Fedora machines and 50 Ubuntu machines from our evaluation set. We ran a 10-fold cross validation on these machines to calculate the accu-

| | | precision | recall | accuracy | confidence |
|---|---|---|---|---|---|
| **evenly biased** | ubuntu | 52.4% | 100% | 54% | 55.3% |
| | fedora | 100% | 8% | | |
| **Ubuntu biased** | ubuntu | 90% | 100% | 90% | 91.4% |
| | fedora | 100% | 0% | | |

**Figure 6: Effects of training bias. The precision, recall, accuracy, and confidence values for an OS classification tool using RandomForest with an even biased training population (50% Ubuntu and 50% Fedora) or with a heavy bias (90% Ubuntu and 10% Fedora).**

racy of a RandomForest learner-based fingerprinting tool. In our second experiment, we heavily biased the population with Ubuntu machines so that we had 90 Ubuntu machines and 10 Fedora machines. Any substantial difference in accuracy between the two experiments will illustrate how training bias affects our results.

In addition to accuracy, we calculated three metrics:

- *precision*: the percentage of test instances correctly classified with label $C$ out of all test instances labeled as $C$ by our tool.
- *recall*: the percentage of test instances correctly classified with label $C$ out of all test instances that are labeled with class $C$ in the ground truth.
- *confidence*: for correctly classified test instances, the average confidence returned by a learner's model that the label it emitted was correct. 100% confidence indicates that the learner was absolutely certain about the correctness of the labels it returned.

Figure 6 confirms that our automatically generated tools are affected by training bias at lower levels of the classification hierarchy. Accuracy in the even biased training data is barely higher than random, and confidence is low, indicating a failure of our tool to find enough distinguishing features. Fedora's low recall of 8% shows that few of the 50 Fedora instances are ever correctly labeled.

There is a large increase in both accuracy and confidence in the Ubuntu-biased training data. The learner's model is latching onto the bias to guess "Ubuntu" when unsure, and because the training data is so skewed to Ubuntu, these guesses are often correct. No Fedora instances are ever correctly classified (0% recall).

Unfortunately, because our full set of 329 machines is skewed towards Ubuntu Linux, the training bias we see in these small scale experiments also affect our results in Section 4.3. To observe this, we recorded the confidence values for all of our fingerprinting tools at each classification level using 10-fold cross validation on all 329 machines. We find that except for course-grained levels of classification, all of our tools again have low confidence value (from 7% to 33% at level 2 of the distribution hierarchy). As in our previous experiments, the automatically generated tools lack confidence when they emit labels, and are relying on biased best guesses to classify test machines, artificially inflating the tools' accuracy results.

This bias is precisely why we needed to modify Nmap to make a training population-driven "best guess" in the case that it could not distinguish between some classes. Without this modification, Nmap's accuracy fell from 21% to 7.5% at level 2 of the OS distribution classification hierarchy.

## 4.4 Summary

Our evaluation suggests that automatic fingerprinting tools for OS classification suffer from several limitations when fingerprinting at scale or with fine-grained classification labels. A fingerprinting tool must overcome mechanical issues while collecting data, such as coping with packet loss and identifying and eliminating non-deterministic fields. As well, automatic fingerprinting tools are prone to overfitting their models to packet response features that are influenced by factors other than true operating system differences. As the scale of the problem increases and granularity becomes more fine-grained, it becomes harder to find genuine and stable discriminative features. As a result, at scale, an automatic tool is more likely to stumble across rare, unlucky coincidences that pollute its model, or to fall prey to training bias in the absence of discriminative features.

## 5. RELATED WORK

Our paper builds on the prior work of Cabellero et al. [6], who first presented the notion of automatically generated fingerprinting tools. We explored the practical challenges and fundamental limitations that arise at scale to confound the accuracy of these tools.

Comer and Lin probed TCP stacks to find flaws, protocol violations, and vendor-specific design decisions [7]. Later work explored passively identifying TCP implementations by looking at packet traces [15], and identified some of the challenges in reliably analyzing TCP traffic. Pahdye et al. used active TCP fingerprinting to study thousands of web servers, characterizing the differences to learn about congestion control mechanisms used in the Internet and to identify compliant TCP implementations [14].

The most prominent TCP fingerprinting tool used for remote OS classification is Nmap [21]. Nmap sends 16 carefully crafted probe packets, calculates a fingerprint based on probe responses, and matches the fingerprint against a database of known OS's fingerprints. The closest match is returned. Other tools such as snifp [3] and synscan [20] perform fingerprinting by sending a small number of well-formed packets to a single TCP port.

Using ICMP for remote OS fingerprinting has also been suggested [1]. Xprobe [2] uses ICMP to avoid detection by IDSs, and also addresses problems with noise from firewalls, packet shaping devices, and user settings such as sysctls. They use a "fuzzy" matching algorithm as an alternative to Nmap that is more robust to small, noise-induced fingerprint variations.

Greenwald et al. investigated optimizing Nmap's probes to avoid IDSs [10]. They analyze Nmap's probes and responses to find the features with the most information gain. They showed that a competitive version of Nmap can be constructed using only 2 to 3 valid TCP SYN probe packets. Prior work has explored improving Nmap's classification algorithm accuracy using neural networks [5]. Other work used logic to specify the OS fingerprinting problem and automated reasoning to arrive at the answer [9].

Passive fingerprinting using traffic traces has been explored by siphon [18] and p0f. Motivated in part by p0f, Beverly et al. propose a passive fingerprinting technique using Web logs [4]. Their classification database is dynamically constructed from web logs, and a naïve Bayesian classifier is used to classify the OS. Lippmann et al. provide a general overview and comparison of the various tools and techniques for passive OS fingerprinting [13]. In addition, they develop their own technique that returns approximate matches from an OS database using k-nearest neighbor when no exact match is available. An overview of passive tools and techniques is provided by Spangler et al. [19].

Techniques have been developed to either mask or spoof the identify of a machine. Smart et al. developed a tool for sanitizing the responses from a network stack to mask the machine from OS fingerprinting tools. Newer Linux-specific tools such as ippersonality and morph modify the network behavior of the machine to match user-specified input behavior. Virtual honeypots spoof the behavior of a network stack to create the illusion of many different machines and attract network intrusion attempts [16].

Fingerprinting has been extended beyond TCP stacks to identify physical devices. Franklin et al. fingerprint drivers on wireless devices by measuring signals generated when scanning for access points [8]. A technique for identifying remote machines using small deviations in clock skew was explored and evaluated by Kohno et al. [12].

## 6. CONCLUSIONS

In this paper, we evaluated the accuracy of automatic OS fingerprinting tools. Extending the earlier work of Cabellero et al. [6], we examined the limitations and challenges of automatically generated tools in more realistic and challenging scenarios, such as when using fine-grained classification labels or when applying the tool in large-scale scenarios. Our tools employed a range of state-of-the-art machine learning algorithms in to provide a realistic assessment of these tools in practice.

We found that automatic OS fingerprint generation faces four major challenges. First, at scale and fine granularity, it is difficult for any tool (manual or otherwise) to find generalizable and sufficiently discriminative probe packets and classification rules; OS variants are simply difficult to distinguish from each other. Second, automatic tools are prone to non-representative bias in the training data. If encountered, the accuracy of the tool during testing might be substantially different than its accuracy when used in the wild. Third, automatic tools are prone to overfitting, in which they mistake an unlucky coincidence for a useful probe or rule. At scale, these unlucky coincidences are more frequently encountered than useful rules, and they tend to pollute the tools' classification models with flawed rules that are unstable or do not generalize. Fourth, fully automatic tools cannot easily exploit semantic knowledge of protocols or generate multi-packet probes and attributes, and as such they are at a disadvantage with respect to manual, expertly generated tools. Overall, these challenges can confound the ability of automatically generated OS fingerprinting tools to work in realistic scenarios.

## Acknowledgments

We thank Roxana Geambasu and our anonymous reviewers for their helpful feedback and guidance. This work was supported in part by the National Science Foundation under grants CNS-0627367, CNS-0430477, and CNS-0722000, a Torode Family Endowed Career Development Professorship, and gifts from Nortel Networks and Intel Corporation.

## 7. REFERENCES

[1] O. Arkin. ICMP usage in scanning: The complete know-how. Technical report, The Sys-Security Group, June 2001.

[2] O. Arkin, F. Yarochkin, and M. Kydyraliev. The present and future of Xprobe2: The next generation of active operating system fingerprinting. Technical report, The Sys-Security Group, July 2003.

[3] P. Auffret. Sinfp, unification de la prise d'empreinte active et passive des systèmes d'exploitation. In *Proceedings of the Symposium sur la Sècuritè des Technologies de l'Information et des Communications*, Rennes, France, June 2008.

[4] R. Beverly. A Robust Classifier for Passive TCP/IP Fingerprinting. In *Proceedings of the 5th Passive and Active Measurement Workshop (PAM 2004)*, Antibes Juan-les-Pins, France, 2004.

[5] J. Burroni and C. Sarraute. Using neural networks for remote OS identification. In *Proceedings of the Pacific Security Conference (PacSec '05)*, Tokyo, Japan, November 2005.

[6] J. Caballero, S. Venkataraman, P. Poosankam, M. G. Kang, D. Song, and A. Blum. FiG: Automatic fingerprint generation. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS '07)*, San Diego, CA, February 2007.

[7] D. E. Comer and J. C. Lin. Probing TCP implementations. In *Proceedings of the USENIX Summer 1994 Technical Conference*, Boston, MA, June 1994.

[8] J. Franklin, D. McCoy, P. Tabriz, V. Neagoe, J. V. Randwyk, and D. Sicker. Passive data link layer 802.11 wireless device driver fingerprinting. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, B.C., Canada, July 2006.

[9] F. Gagnon, B. Esfandiari, and L. E. Bertossi. A hybrid approach to operating system discovery using answer set programming. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management*, Munich, Germany, May 2007.

[10] L. G. Greenwald and T. J. Thomas. Toward undetected operating system fingerprinting. In *Proceedings of the First USENIX Workshop on Offensive Technologies (WOOT '07)*, Boston, MA, August 2007.

[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11, 2009.

[12] T. Kohno, A. Broido, and K. Claffy. Remote physical device fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[13] R. Lippmann, D. Fried, K. Piwowarski, and W. Streilein. Passive operating system identification from TCP/IP packet headers. In *Proceedings of the ICDM Workshop on Data Mining for Computer Security (DMSEC)*, Melbourne, FL, November 2003.

[14] J. Pahdye and S. Floyd. On inferring TCP behavior. In *Proceedings of the 2001 ACM SIGCOMM Conference*, San Diego, CA, August 2001.

[15] V. Paxson. Automated packet trace analysis of TCP implementations. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Cannes, France, September 1997.

[16] N. Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, Washington, DC, August 2003.

[17] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[18] C. Smith and P. Grundl. Know your enemy: Passive fingerprinting. Identifying remote hosts without them knowing. Technical report, Honeynet Project, March 2002.

[19] R. Spangler. Analysis of remote active operating system fingerprinting tools. Technical report, June 2003.

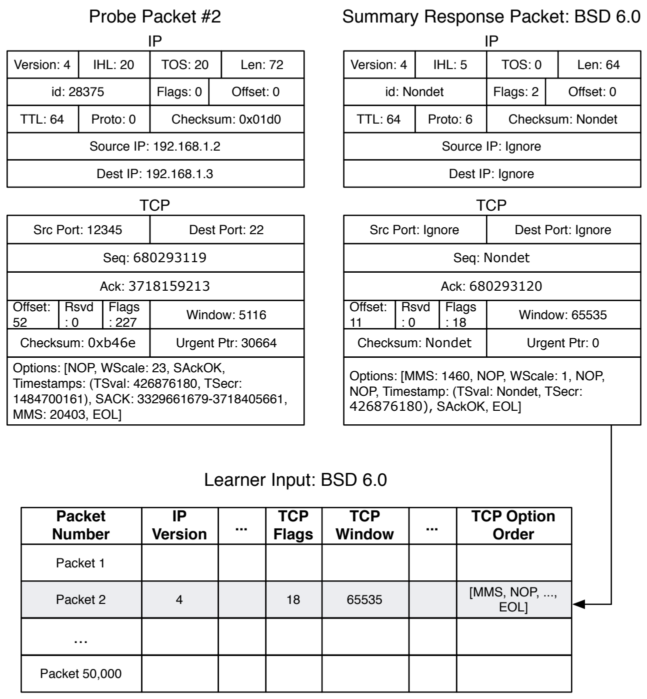
[20] G. Taleck. SYNSCAN: Towards complete TCP/IP fingerprinting. In *Proceedings of the Canada Security West Conference (CanSecWest '04)*, Vancouver B.C., Canada, April 2004.

[21] F. Yarochkin. Remote OS detection via TCP/IP fingerprinting (2nd generation). `http://nmap.org/book/osdetect.html`, January 2007.

| IP Fields |
|---|
| Version, IHL, TOS, Len, Id, Flags, Offset, TTL, Proto, Chksm, Src IP, Dest IP |

| TCP Fields |
|---|
| Src Port, Dest Port, Seq, Ack, Offset, Reserved, Flags, Window, Chksm, Urgent Ptr, Payload?, Option Order, MMS, WScale, Timestamp.TSval, Timestamp.TSecr, SAckOK, SAck.left, SAck.right, SAck.more, Other options |

**Figure 8: List of Response Fields. This figure shows a list of the IP and TCP fields whose values were used by our learner in generating fingerprinting tools.**



| Packet Number | IP Version | ... | TCP Flags | TCP Window | ... | TCP Option Order |
|---|---|---|---|---|---|---|
| Packet 1 | | | | | | |
| Packet 2 | 4 | | 18 | 65535 | | [MMS, NOP, ..., EOL] |
| … | | | | | | |
| Packet 50,000 | | | | | | |

**Figure 7: Experimental Data Example. This figure shows the IP and TCP fields and values of an actual probe packet and the summary response packet returned by a FreeBSD 6.0 training machine in our experiments. The feature vector for this machine containing the response data from all 50,000 probe packets is also shown.**

## APPENDIX

## A. EXPERIMENTAL DATA EXAMPLE

In Section 3, we described the design and implementation of our automatic fingerprint generation tool, as well as the data sets we used for evaluation. We discussed how probe packets were sent to a set of training machines, and how the responses were then sent to a learner responsible for generating the fingerprinting tool. Figure 8 shows a list of all the IP and TCP fields whose values we included in the data sent to our learners. Figure 7 provides a detailed example of the kind of probe packets, response packets, and learner input that we used in our experiments.

Figure 7 shows the IP and TCP fields and values of an actual probe packet that we sent to our training machines. We also show a summary response packet for this probe from a FreeBSD 6.0 training machine. Recall that we send each probe packet to each training machine a total of 6 times, collapsing the responses into a single "summary response packet" to weed out non deterministic fields. This resulting summary response packet is what we record for a probe's response. Note that some fields in the summary response packet in Figure 7 such as the IP Checksum were deemed non-deterministic by our implementation during the response packets collapsing procedure.

Each of our 50,000 probe packets generated a similar summary response packet for this machine. As we show in Figure 7, the input passed to our learner for this training machine consisted of a feature vector of the IP and TCP field values from all 50,000 such summary response packets. Each training machine used by the learner during an experiment generated the same feature vector, but presumably with different values.