Advanced Passive Operating System Fingerprinting Using Machine Learning and Deep Learning

Desta Haileselassie Hagos*, Martin Løland†, Anis Yazidi‡, Øivind Kure §, Paal E. Engelstad ¶
*§¶University of Oslo, Department of Technology Systems, Kjeller, Norway
*†‡¶Oslo Metropolitan University, Department of Computer Science, Oslo, Norway
Email: *destahh@ifi.uio.no, †martin.loeland@gmail.com, {‡anis.yazidi, ¶paal.engelstad}@oslomet.no, §oivind.kure@its.uio.no

***Abstract*—Securing and managing large, complex enterprise network infrastructure requires capturing and analyzing network traffic traces in real-time. An accurate passive Operating System (OS) fingerprinting plays a critical role in effective network management and cybersecurity protection. Passive fingerprinting doesn't send probes that introduce extra load to the network and hence it has a clear advantage over active fingerprinting since it also reduces the risk of triggering false alarms. This paper proposes and evaluates an advanced classification approach to passive OS fingerprinting by leveraging *state-of-the-art* classical machine learning and deep learning techniques. Our controlled experiments on benchmark data, emulated and realistic traffic is performed using two approaches. Through an Oracle-based machine learning approach, we found that the underlying TCP variant is an important feature for predicting the remote OS. Based on this observation, we develop a sophisticated tool for OS fingerprinting that first predicts the TCP flavor using passive traffic traces and then uses this prediction as an input feature for another machine learning algorithm for predicting the remote OS from passive measurements. This paper takes the passive fingerprinting problem one step further by introducing the underlying predicted TCP variant as a distinguishing feature. In terms of accuracy, we empirically demonstrate that accurately predicting the TCP variant has the potential to boost the evaluation performance from 84% to 94% on average across all our validation scenarios and across different types of traffic sources. We also demonstrate a practical example of this potential, by increasing the performance to 91.3% on average using a tool for TCP variant prediction in an emulated setting. To the best of our knowledge, this is the first study that explores the potential for using the knowledge of the TCP variant to significantly boost the accuracy of passive OS fingerprinting.**

***Keywords*—Operating System, Fingerprinting, Machine Learning, Deep Learning, Passive Measurements**

I. Introduction and Motivation

As modern network infrastructures grow in size, collecting detailed relevant knowledge about the dynamic characteristics and complexity of large heterogeneous networks is crucial for many purposes e.g., network vulnerability assessment and monitoring, spam detection, etc. Developing advanced network security and monitoring techniques are important for both the research and security practitioners. There has been a significant research work in the context of network management and cybersecurity on developing network security tools to fingerprint remote Operating Systems (OSes) [26, 27, 28, 41, 42]. OS fingerprinting is the process of inferring the OS of a machine operating with TC/IP by a remote device connected on the Internet without having physical access to the device [20]. There are many different custom tools for fingerprinting of the most commonly used OSes based on the characteristics of its underlying TCP/IP network stack [20] and this, to a large extent, is due to variability in how the TCP/IP stack is traditionally implemented across different OSes [25]. One common approach, for example, is by collecting the TCP/IP stack basic parameters [23], e.g., IP initial Time To Live (TTL) default values [5], HTTP packets using the User-agent field [22], Internet Control Message Protocol (ICMP) requests [29], known open port patterns, TCP window size [18], TCP Maximum Segment Size (MSS) [31], IP Don't Fragment (DF) flag [30], a set of other specific TCP options to mention a few. However, in our work, we want to take this one step further by combining these basic features and other settings with the underlying TCP variant as a feature in our model due to the fact that different OSes are doing slightly different implementations of TCP. Some implementations of common TCP variants quickly overshoot the size of the Congestion Window (cwnd) because of differences in the variant implementations. Hence, we believe that knowing the implementation of the underlying OS may help us understand better their exact behavior. It can also help us explore how to classify an OS when different OSes are implementing the same TCP variant.

Fingerprinting Techniques: We can determine what OS a remote computer on the Internet is running by either passively listening to traffic captured from a network or by actively sending it packets. The most widely used complementary remote OS fingerprinting proven approaches that employ a variety of TCP/IP stack scanning are broadly categorized into classes of *active* and *passive* methods.

- ***Active Fingerprinting***: This technique is based on actively transmitting one or more specially crafted network packets with different packet settings or flags to a remote network device in order to analyze the corresponding potentially identifying replies [26, 41]. This method determines knowledge of the underlying OS according to the received responses from the target device by examining the network behavior of known TCP/IP stack [35]. However, since this approach injects additional traffic to the network by generating active probes, it may itself trigger alarms and get blocked by firewall rules and Network address translators (NATs) [8].
- ***Passive Fingerprinting***: This approach, on the other hand, inspects and analyzes packets traveling between end hosts without injecting any traffic into the network [27, 28, 42]. This technique with little resource simply analyzes a pattern of the OS-specific information that has already been sent in the network traffic and compares for a match with a predefined database that contains a list of known signatures of different OSes. Passive fingerprinting doesn't send probes and hence it has a clear advantage over active fingerprinting since it reduces the risk of triggering alarms [8].

OS fingerprinting can also be performed using classical techniques known as "*banner grabbing*". It is an approach used to gain detailed information about a remote computer system on a network and the associated services running on its opened ports [33]. Using techniques like this, some remote computers announce their underlying OS freely and running application services with their versions in use to anyone connecting to them as part of welcome banners or header information. Some of the widely used services that serve *banner grabbing* are: *Telnet*, *FTP*, *NetCat*, *SMTP*, etc. However, it is useful to remember that some of these basic services are effective against less secure networks.

Potential benefits and applications: Network scanning and accurate remote OS fingerprinting are the crucial steps for penetration testing in terms of security and privacy protection. Note that attackers can also embrace passive fingerprinting techniques to search for potential victims in a network. For example, by identifying the OS running on a remote computer and the list of services it runs, an attacker can target the device to eavesdrop on the communication between the endpoints without having physical access to the device. However, we argue that our work presented here is motivated by a number of practical applications that can be positively used by network and system administrators. Passively fingerprinting an OS by analyzing the packets it generates and transmits over a network is extremely important in the areas of network management and computer security for several reasons. For example, it is useful to explore a network for potential exploitations of security vulnerabilities which can be exploited by attackers, auditing, identify critical attacks, reveal new information about a network user etc. Network administrators can, therefore, use this OS related information to maintain the security policy and reliability of their network by configuring a network-based Intrusion Detection Systems (IDS) [24]. Vulnerabilities and security threats in a network may result from rogue or unauthorized devices [38], unsecured internal nodes within the network, and from external nodes [4]. Hence, passively fingerprinting an OS has a potential benefit in addressing these critical problems. This, from an academic point of view, is interesting and something that needs to be addressed from a network security research point of view.

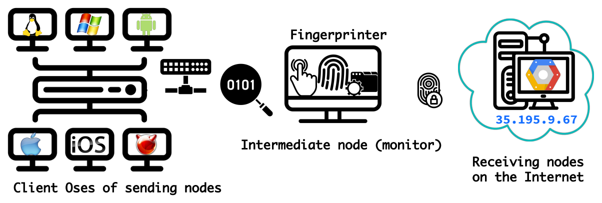

Fingerprinter
0101
Intermediate node (monitor)
35.195.9.67
Receiving nodes on the Internet
iOS
Client Oses of sending nodes

Fig. 1: Network architecture for passive OS fingerprinting by an intermediate node.

Limitations of previous works: Traditionally, most of the existing general OS fingerprinting techniques resort to manually generated signature matching from a database of heuristics which contains features of widely used OSes. This means, after comparing the generated signatures, the first set of responses match with the highest confidence against a database of fingerprints would be used to select the specific probable OS. However, manually updating a large number of signature and managing databases of new OSes adds a considerable amount of time and hence we may suffer from the consequences of the lack of recent signature updates of the known OSes. For example as reported in [22], the last updates of the fingerprint databases of *Ettercap* [28] and *p0f* [42] date to 2011 and 2014 respectively. Consequently, new OSes families like Android 4.4 and higher versions of Android, Windows 10 distributions, etc. will not be recognized by these tools since they are not included in their fingerprint databases. Hence, we argue that it is important to consider making use of a fingerprint database that contains variations of most currently used OSes and automating these tasks by employing learning algorithms capable of extracting all possible OS-specific features for discovering the underlying OSes. To explore this idea of applying learning algorithms, we present a unified and robust classification approach to an advanced passive OS fingerprinting that leverages both machine learning and deep learning methods. Our fingerprinting technique is completely passive meaning that we only need to be able to observe network traffic from a target machine at any observation point on the network without injecting any traffic into the network. Note that the TCP/IP header fields would not be impacted by SSL/TLS encryption of the TCP payload. Hence, since we utilize features that are readable even with encryption, our approach is independent of whether the flow is encrypted or not. Figure 1 shows the architecture for implementing our fingerprinting methodology.

Why machine learning approaches to OS fingerprinting? There are several limitations imposed by classical fingerprinting techniques. Passive OS fingerprinting generally relies on recognizing the default values for various TCP/IP stack parameters. If a user changes these parameters, the task of OS fingerprinting becomes much more challenging. Most of the existing works on fingerprinting provide a little capability to address this challenge. Motivated by this problem, we proposed a novel approach by leveraging both machine learning and deep learning-based techniques that consider the set of parameters as a whole, rather than individually so that our model caters for variations in TCP parameters. If a user changes the initial receive window size, for instance, we may still be able to recognize the OS from other parameters that have not been changed (TCP congestion control algorithm, initial cwnd size, etc.). Note that this depends entirely on the changes made by the user to the default TCP or OS stack parameters that are commonly used for signature-based fingerprinting. The other reason why we create a model by employing learning techniques is to understand the complex patterns of the varying values in the TCP header and extract useful input features. Because machine learning offers new possibilities as it can extract patterns and general rules for classification. Machine learning can also be more robust to small variations in the input parameters. In addition to this, with the use of learning techniques, we argue that avoiding using manually updated static signature databases has two potential benefits. Firstly there is no tedious task of creating these unique fingerprints, all you need is a set of values or features. The second benefit comes from a known flaw in many of the existing fingerprinting tools, where a "first-match" policy is applied, meaning that if two fingerprints are equal the tool would always predict the first OS with that exact fingerprint. However, learning techniques, on the other hand, make calculated guesses of which of the classes with the same fingerprint that will be predicted.

Contributions: We summarize our main contributions below.

- We propose and evaluate a robust approach to OS fingerprinting from passive measurements by leveraging machine learning and deep learning techniques.
- We investigate the use of TCP congestion control variant as a distinguishing feature in passive OS fingerprinting.
- We explore variability in implementations of TCP variant by different OSes and its effect on classifying remote OS.
- We study the applicability of Recurrent Neural Networks (RNN)-based models for robust and advanced passive OS fingerprinting by combining the basic TCP/IP features and the predicted TCP variant as input vectors.
- We show that the TCP flavor has a great potential for boosting passive OS fingerprinting.
- We build a universal tool for passive monitoring that can be applied to first estimate the TCP cwnd, second predict the TCP flavor, and finally uses the TCP variant as an input feature to detect the remote computer's OS.

Roadmap: The rest of the paper is organized as follows. Section II discusses related work, and Section III presents the experimental datasets. Section IV presents the machine learning of the OS fingerprinter. The machine learning of the TCP variant prediction tool is presented in detail in Section V. Section VI presents the experimental results without a known TCP variant which will play the role of baseline. In order to assess the importance of knowing the TCP variant, experimental results of all the use cases with an Oracle-given TCP variant are presented in Section VII. Section VIII presents the experimental results with the predicted TCP variant. Section IX presents the transfer learning results. Finally, Section X concludes our paper and suggests directions for future research work.

II. Related Work

Remote OSes fingerprinting has a long history in the computer security community [2, 22, 23, 26]. TCP/IP header fingerprinting and any information related to application protocols are used to identify the underlying OS running on a remote host either actively or passively [25]. As we explained in Section I, there are multiple existing tools for both the predominant active and passive OS fingerprinting approaches, where *Nmap* [26] is one of the most prominent open-source active fingerprinting tools. The work presented in [36], *SYNSCAN*, works in a similar fashion to *Nmap*, but it performs the fingerprinting task by actively sending a small number of crafted network packets to a single TCP port. *Xprobe2* [41] is another popular fingerprinting tool, that relies primarily on ICMP packets, and it depends on how many changes we make to the default TCP/IP stack parameters. Since *Xprobe2* does fuzzy fingerprinting with a signature matching algorithm as an alternative to *Nmap*, it means that if we make a lot of changes to the default TCP/IP stack parameters, the underlying OS will not be detected. However, *Xprobe2* is more robust to small fingerprint variations as compared to *Nmap*. As explained above the other fingerprinting tools, *Ettercap* [28] and *p0f* [42], have not been updated since 2011 and 2014 respectively to include variations of most widely used modern OSes. For passive OS fingerprinting to be effective, we believe that the limitations of these fingerprinting tools need to be addressed. The work in [23] also demonstrates that the OS fingerprinting accuracy of the *Ettercap* and *p0f* signature databases is low and techniques to improve performance was proposed. Hence, the paper presents rule-based machine learning classifiers capable of identifying 75 classes of OSes from TCP/IP packet headers found in the *Ettercap* database. They proposed a classifier technique using k-nearest neighbors (KNN) that returns an approximate first match for an OS from a fingerprint database. This counters the problem of classifying hosts as unknown if no exact match is found in the database [23]. However, their evaluation yielded poor experimental results, rejecting as much as 84% of the test packets, while 44% of the accepted patterns were wrongly classified [23]. The problems contributing to poor performance was believed to be caused by two main issues. The first reason is substitution errors due to multiple OSes with exactly the same fingerprint feature values. The second reason for this poor performance is the high rejection rate caused by numerous unique feature values derived from the same OS. After combining the OS classes most often confused with each other, eliminating all the classes where the error could not be reduced by combining classes, the error percentage was reduced to 9.8% with no rejected packets.

A recent study that is most closely related to our work, and which has also given a comprehensive survey on passive fingerprinting methods, can be found in [22]. The authors have employed OS fingerprinting methods in the environment of wireless networks. Besides using the three basic TCP/IP stacks (i.e., TTL, window size, and initial SYN packet size), the authors suggested also using the user-agent information in HTTP request headers and communication with OS-specific domains can be usable in large dynamic networks [22]. The average accuracy of OS classification using the TCP/IP parameters reported in [22] is 80.88%. Zhang et al.'s paper on OS detection [43] utilizes only one machine learning technique namely Support Vector Machine (SVM). However, the testing error rate of identifying some of the OSes e.g., Mac, Cisco, FreeBSD, and OpenBSD is 25.80%, 24.22%, 17.71%, and 15.85% respectively [43]. Aksoy et al. [2] have employed genetic algorithms for identifying packet features suitable for OS classification based on the analysis of the network TCP/IP packets using machine learning algorithms. However, most of these previous works use the basic actual TCP/IP features for evaluating passive OS fingerprinting. Besides, we believe that these tools have the inability to extract all possible OS-specific features for passively fingerprinting the underlying OSes. In contrast, what separates our contribution in this paper from the other previous related works is that our model supports a wider range of TCP/IP network stack features. As shown in Figure 2, the main goal of our work presented here is to combine these basic TCP/IP features that are the basis of OS fingerprinting with the underlying TCP variant by leveraging both machine learning and deep learning techniques. This contribution remains largely unexplored and is not used by existing fingerprinting techniques. Detecting the implementation of a TCP variant passively is a challenging task and this, we believe, is the reason why no previous works use it to passively fingerprint remote OSes. However, in our case, we already have a general solution for this difficulty presented in our previous works [11, 12, 13]. The reason why we focus on the implementations of the underlying TCP variant as a feature in our OS classifier model is due to the fact that different OSes are doing slightly different implementations of TCP. Hence, we believe that passively observing the network-level characteristics found in TCP packets can give us more information about the remote computer's underlying OS. We further believe that this will also help us to explore in detail the long-term characteristics of TCP traffic. To the best of our knowledge, this is the first study of passive fingerprinting OSes by applying RNN methods combining the basic TCP/IP features and the underlying TCP variant as input vectors.

III. Experimental Datasets

Our machine learning models for OS classification is developed and tested on three datasets, presented below.

A. Benchmark Data

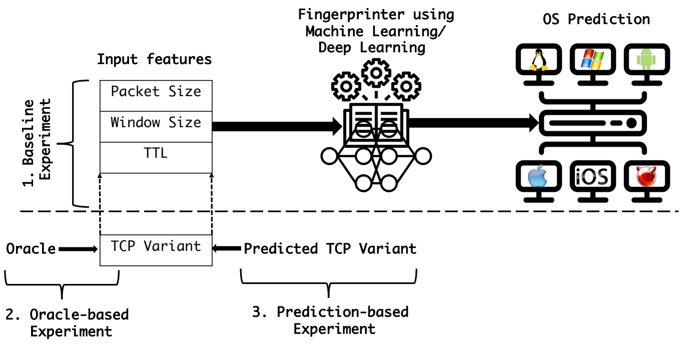
First, we utilize a large benchmark dataset that has been used for OS fingerprinting in a previous related work [22]. This dataset is closely aligned with our task. The benchmark dataset was used in the previous work for OS fingerprinting based on the HTTP header, while the ambition of our paper is to do generic fingerprinting based only on the TCP packet fields. Since we aim at fingerprinting that is not application-specific, the TCP information in the dataset is useful for our purpose, while the HTTP User-agent information in our experiments is used only to establish ground truth about the OS that was used. The benchmark dataset contains 79087345 flows, activity of 21746 unique users, 253374 WiFi sessions, 25642 unique MAC addresses, and 6104 unique IP addresses, a fingerprint database of 2078 standard TCP/IP signatures of 51 known unique OSes with a total of 529 variations when considering major and minor versions [22]. It consists of three basic TCP/IP network stack features, i.e., initial SYN packet size, TTL, and TCP window size [22]. After our first set of testing, we realized that the data was severely skewed and that only a few of the classes contained almost all of the entries, giving us artificially good classification results. We then removed most of the very seldom occurring classes and ended up with 33 reduced classes. We also removed all traffic that did not contain HTTP User-agent information, since we could not establish ground truth for this traffic. In addition, we created a new dataset where all the classes were bucketed into seven groups, consisting of the six most widely used major OS families: Android, Linux, Mac OS, Unix, Windows, iOS, and a seventh class called "Other" for OSes not suited for any of the other groups. Finally, we ended up distributing all of the labels equally so that each OS class had the same number of occurrences. This helps us improve the generalizability of our model with a unified approach that encompasses all variations of the most widely used OSes.

B. Realistic Traffic

While benchmark traffic is useful to link our experiments to previous related work, we also wanted additional realistic traffic for which we have more control, and that allows us to make our own assurances of the quality of the data. Thus, we passively collected our realistic dataset from TCP traffic originated from the internal network of the Oslo Metropolitan University and destined to various hosts on the Internet. First, we collected data for fixed (non-mobile) desktop computers (typically using OSes like Windows, Linux, Unix, Mac OSx, etc.) by using an intermediate node as shown in the network setup in Figure 1. Then, we passively collected the data that covered mobile devices, like *android* and *iOS*. The latter was collected from the 5G 4IoT research lab [1, 34] of the Oslo Metropolitan University.

We spent a significant amount of effort in establishing ground truth, i.e., determining the actual OS that has been used for each traffic flow. To establish ground truth in the realistic dataset, we follow two approaches. The first approach was only applicable to the non-mobile desktops, while the second method was used for both mobile and non-mobile devices. With the first method, we leveraged the DHCP log messages associated with the non-mobile desktops to derive the ground truth from the DHCP server of the Oslo Metropolitan University network that logs the sessions by the MAC address and name of the device. Since we collect the real data from the internal network of our university, extracting the DHCP log messages can give us detailed information about the OSes. We could, for example, see information about the *vendor-specific prefixes* since most of the OS variants are identified based on their vendors. The list of device vendor prefixes is useful in revealing the specific implementation of an OS because most of the modern OSes from the same device vendor usually share the same OS kernel and similar network behaviors. For example, we found out that Apple products often share the same TCP/IP parameters. The second approach we used to identify the OS is getting the predefined browser strings that loosely tell the name of the underlying OS assigned by the vendor from Webserver.

We believe changing the default device names by all users is not that common and sometimes discouraged by the vendors, e.g., Google and Apple OSes. However, the device name of Linux and Windows OSes could be changed easily by experienced users which would make passively identifying these devices hard. Since a number of computer vendors offer devices with a pre-installed OS and default device name and MAC address, we can use this information to derive the ground truth for OS fingerprinting. For example, Apple devices use a default string name of "<user>-iPhone", "<user>-iPad", Microsoft uses "Windows-Phone" for its mobile devices, and Android uses "android-<android_id>", etc. Our real traffic covers the communication to and from our university and hence all traffic whose source and destination IP addresses are within the subnets of our internal network. Hence the network administrator of our university has full control over the internal machines with real IP addresses that are not going to a NAT gateway, and therefore it is fairly possible to tell whether it is a laptop or a desktop PC by looking it up in the internal database owned by the university. However, since it is a dynamic network we do not have full control over external machines, because they can be anything behind an IP address that changes dynamically. This is because there is an endless number of machines spoofing scanning the network and they can appear as Linux-powered OSes but they could be Windows and vice versa and this happens because the user may have strongly tuned the TCP stack to look like something else. It is pretty hard to certainly say anything about the external computers because the communication can go through a NAT gateway possessing another OS type. For example, if a user is connected to a student wireless network, there is a chance that it may go to a Linux NAT gateway, and hence from outside the user is seen as Linux NAT which makes it hard to predict whether the underlying OS is Linux, Mac or Windows. Therefore, fingerprinting devices behind NAT technology on a distributed network where a number of devices can hide behind a NAT is another critical challenge. It is, therefore, worth noting that establishing ground truth in dynamic networks at a larger scale remains a challenging problem. Further investigation to explore these difficulties will be done in our future works. Finally, due to the privacy protection of possibly sensitive data, the payload of all the network packets collected was removed and anonymized with a prefix-preserving algorithm [7, 39]. Furthermore, we were only allowed to collect TCP headers of the traffic flows, while we could not collect complete traffic captures, due to privacy protection and legal reasons.



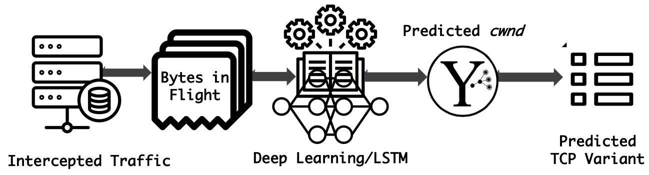


Fig. 3: The process implemented on the monitor for prediction of the TCP variant of the passively intercepted TCP traffic flow. An LSTM-based machine learning module predicts the cwnd from the outstanding bytes-in-flight. In the next step, the cwnd behavior is used to predict the TCP variant as explained in further detail in our previous works [11, 12, 13]. The predicted TCP variant is finally used as an input feature to the OS fingerprinting process (see bottom right part of Figure 2).

C. Emulated Traffic

In a real scenario where the OS fingerprinting is going on continuously in an intermediate node of an enterprise or production network, the intermediate node will have more information available than only the TCP header, such as the traffic profile or the knowledge of congestion or the outstanding bytes-in-flight of a flow. In our experiments below, we show how this information can be very useful for OS fingerprinting. Since we do not have full traffic packet captures in our benchmark dataset or in our realistic dataset, we needed an additional dataset that we collected from an emulated network, where there would be no privacy protection or legal issues related to our dataset. The architecture of our emulated network is similar to the network setup shown in Figure 1, except that all the nodes (the sender, the intermediate node, and the receiver) are implemented in virtual machines. All background traffic of the OSes for our emulated scenario is generated using the *iperf* [6]. Establishing ground truth is straightforward, as we have full control of the OSes used when generating the traffic. In addition to establishing the ground truth, we also wanted to allow the intermediate node to establish a prediction of the TCP variant by monitoring the on-going traffic profile of the TCP flow between the sender and the receiver. As shown later in the paper, using definitive or predicted knowledge of the TCP variant as an additional input feature to the OS fingerprinting, might boost the fingerprinting accuracy significantly. How the machine learning model for prediction of the TCP variant in the emulated scenario is trained and how the TCP variant is subsequently predicted are presented in the following.

IV. Machine Learning of the OS Fingerprinter

A. Classical Machine Learning Approaches

The OS fingerprinter takes various features as input parameters, and use machine learning to predict the OS as shown in Figure 2. Many machine learning techniques could be used to implement a model for passive OS fingerprinting. In this paper, we have employed the following most commonly used classical machine learning methods suitable for our task. In order to train and test our classification models, we employed every experiment with a ratio of 60% training, 40% testing split, and 5-fold cross-validation setting on all variations of the features into one learning model.

SVM: In order to perform an efficient multi-class SVM classification through cross-validation, we tuned the SVM hyperparameters using a *GridSearchCV* that allows specifying only the ranges of values for optimal parameters by parallelization construction of the model fitting. Finally, in our evaluation, we found out that *SVM* with a Radial Basis Function (RBF) kernel for classification model yields a substantially better result.

Random Forest (RF): We tuned the meta-estimator by varying the number of decision trees between 1 and 1000. We found out that increasing the number of trees more than 10 doesn't give much improvement in the classification accuracy.

KNN: We applied KNN by testing different values of K ranging from 5 to 100 followed by a weight function for a total of 20 observations. The observations have been conducted in two ways. In the first experiment, we set the weight to *uniform*. In the second experiment, the points are weighted by the inverse of their distance, causing closer neighbors to have greater influence. Finally, we choose the model that has the highest accuracy for a given unseen instance.

B. Deep Learning Approaches

To find the deeper characteristics of TCP variants implemented by respective OSes and exploit the extra OS-specific information, we apply the following two neural network architectures.

Multilayer Perceptron (MLP): In our evaluation, MLP model with a single-layer feedforward neural network [16, 32] has been used to classify the different classes of OSes. After the hyperparameter tuning, we tested our MLP model with a different number of batch sizes, hidden layers, and nodes (e.g., 0, 1, 2, 32, 64, 128) in each layer. Combining all of these, a total of 324 models were trained with and without the default TCP variant. We found out that the results for both with and without a known TCP variant were almost the same with an insignificant drop in the accuracy irrespective of which hyperparameters performed the best. Finally, 128 nodes of the network per dataset are trained for 150 epochs with a batch size of 500 by SGD with momentum of 0.9 and a constant learning rate of 0.01. However, we learned that SGD is sensitive in regards to the selection of the learning rate since it doesn't automatize the values and we also found that it suffers from premature convergence and is outperformed by *Adam*-based optimization methods. Hence, both *Adam* and *Nadam* gradient-based optimization algorithms fit for our purpose and that is because we wanted to use an optimization algorithm that adapts its learning rate dynamically in a way that doesn't affect the objective function and learning process of the model. Our experimental results show that the hyperparameter tuning baseline experiments by applying *tanh* as activation function and *Adam* optimization algorithm and training the model for 150 epochs, provides a substantial improvement in accuracy as compared to the other parameters.

Long Short-Term Memory (LSTM) models: We have explored an approach to classify the underlying OS from passive measurements using LSTM-based RNN architecture by combining the basic TCP/IP features and the underlying TCP variant shown in Table 2 as input vectors. For more details about LSTM applied in the context of computer networks, we refer the reader to our previous paper [12]. We trained our LSTM model over 150 epochs of the training samples with a batch size of 32 as values in time-series. We propagate the input feature vector (x) to the model through a multilayer LSTM cell followed by a fully connected dense layer of 150 hidden nodes with Rectified Linear Unit (ReLU) activation function using the *hard_sigmoid* as recurrent activation for the different layers that generates an output of a sequence dimensional vector of predicted OSes (y_t). We trained our LSTM-based learning algorithm without the knowledge of the input features from the true signatures of the OSes during the learning phase. We learn the model from the training data and then finally predict the test labels from the testing instances on all variations of the OS-specific parameters. In order to train our prediction model more quickly, and get a more stable and robust to changes OS classification model, we have applied one of the most effective optimization algorithms in the deep learning community, the *Adam* stochastic algorithm [19] with an initial *learning rate* of *0.001* and *exponential decay rates* of the first (β_1) and second (β_2) moments set to 0.9 and 0.999 respectively. We further optimize a wide range of important hyperparameters related to the neural network topology to improve the performance of our OS classification model.

C. Experimental Hardware Setup

All our machine learning experiments are carried out using a cluster of HPC machines based upon the GNU/Linux operating system running a modified version of the *4.15.0-39-generic* kernel release. The prediction model is performed on an NVIDIA Tesla K80 GPU accelerator computing with the following characteristics: Intel(R) Xeon(R) CPU E5-2670 v3 @2.30GHz, 64 CPU processors, 128 GB RAM, 12 CPU cores running under Linux 64-bit. All nodes in the cluster are connected to a low latency 56 Gbit/s Infiniband, gigabit Ethernet, and have access to 600 TiB of BeeGFS parallel file system storage.

D. Objectives of our Experiments

The aim of our experiments is to explore the effect of the TCP variant as an input feature when passively detecting the underlying OS. To investigate this, we divide our analysis into three different experiments. First, in the baseline experiment (Section VI) we carry out the OS fingerprinting without using a known TCP variant as an input feature. This corresponds to the simplest state-of-the-art transport layer method, which is illustrated in the upper part of Figure 2. Since there is a close connection between existing popular OSes and the TCP variants they use, our hypothesis was that the potential for improvement by using the TCP variant as an input feature would be significant. For example, CUBIC [9] is the default congestion control algorithm as part of the Linux kernel distribution configurations from version 2.6.19 onwards. Since Android devices are also Linux-powered, CUBIC remains to be the default TCP congestion control algorithm. Many Windows 7 distributions have been shipped with the default New Reno [15] and whereas Windows 8 families with CTCP [37]. Therefore, in the next Oracle-based experiment (Section VII), we investigate the potential of knowing the TCP variant, and how much this knowledge might boost the fingerprinting accuracy. Here we assume that there is an Oracle that can identify and give the TCP variant used in the TCP flow that is fingerprinted. This is illustrated in the bottom left part of Figure 2. However, in a real scenario, the intermediate node would not have access to definite knowledge of the TCP variant (e.g., given by an Oracle). Instead, the intermediate node might at best try to infer it from the monitored traffic. Thus, in the third prediction-based experiment (Section VIII), we first allow the intermediate node to predict the TCP variant passively. This is illustrated in the bottom right part of Figure 2. The OS fingerprinter then uses that TCP variant prediction as an input feature to make the OS prediction illustrated in the upper part of Figure 2. The TCP variant is predicted by analyzing the famous sawtooth pattern behavior of estimated cwnd of TCP, which is computed based on the outstanding bytes-in-flight [12, 13]. This is presented in more detail in the next section. Since the latter experiment requires TCP traffic details of outstanding bytes-in-flight, which is not available in our benchmark and realistic datasets, this experiment is only possible with our emulated dataset.

V. Machine Learning of the TCP Variant Prediction Tool

The main goal of the experiments in the emulated network is to use the predicted TCP variant as an additional input feature to the OS fingerprinting. The TCP variant is predicted by the process illustrated in Figure 3. As described in sufficient detail in our previous works [11, 12, 13], we used a database to match and join the intercepted TCP traffic on both the intermediate node and the sending node. The outstanding bytes-in-flight of the traffic (i.e., the number of bytes that have been sent but not yet acknowledged) is used as input to our machine learning model to predict the cwnd behaviour of the traffic. We use LSTM for the machine learning. We trained and verified the machine learning model by matching the predicted TCP states with the actual TCP kernel states directly logged from the Linux kernel. Since we have full control of the sending nodes, we can track the system-wide TCP state of every packet that is sent and received from the kernel to verify our model's prediction accuracy against the actual TCP variant by matching with the actual sending TCP states using the techniques presented in our previous works [11, 12, 13]. After the verification, we can run our learning model and get the cwnd predictions of the TCP stack in use.

Once we can estimate the cwnd of the sender, we can also infer the multiplicative back-off factor (β) which is an important feature for uniquely identifying the TCP variants. Finally, we combine the predicted TCP variant as the basis of OS fingerprinting with the basic TCP/IP features as shown in Figure 2. Here, we consider only loss-based TCP congestion control algorithms, e.g., BIC [40], CUBIC [9], CTCP [37], Reno [17], and New Reno [15]. Delay-based TCP variants are investigated in a follow-on paper [14]. Our approach could also be useful to other TCP variants like Google's QUIC [21]. QUIC uses packet loss as an indicator of congestion and supports a number of different congestion control algorithms, including CUBIC [9] and BBR [3].

VI. Baseline Experiment: Results without Knowing the TCP variant

Here we present the results of the machine learning and deep learning techniques under all the validation scenarios presented above without a known underlying TCP variant which will play the role of baseline for the other evaluations.

A. Based on Benchmark Data from Previous Related Work

Looking at Tables I and II, both machine learning and deep learning classification techniques have consistently achieved good levels of precision and recall for all general classes of OSes except iOS. Quantitatively, iOS, and Mac OS devices were underrepresented in the benchmark data from previous related work. Besides, as it is shown in Figures 4, there is a slightly higher misclassification of iOS as unknown and this is why the precision and recall of iOS are comparably lower than the rest of OSes. We also believe that the limited TCP/IP stack basic features could contribute to the indistinguishability and misclassification of OS classes with the same kernel implementation. The false positives are easier to notice in the corresponding confusion matrices.

TABLE I: Benchmark data [22] experimental results without a known TCP variant using SVM, RF, and KNN.

	SVM		RF		KNN	
OS	Precission	Recall	Precision	Recall	Precision	Recall
Android	0.74	0.88	0.87	0.91	0.87	0.91
Linux	0.85	0.85	0.91	0.90	0.91	0.90
Mac OS	0.65	0.77	0.61	0.83	0.58	0.88
Other	0.91	0.81	0.92	0.81	0.92	0.81
Unix	0.91	0.99	0.94	0.99	0.94	0.99
Windows	0.97	0.88	0.98	0.91	0.98	0.91
iOS	0.73	0.55	0.72	0.53	0.79	0.47
Average	0.83	0.82	0.85	0.84	0.86	0.84
Accuracy	**81.96%**		**84.07%**		**83.95%**	

B. Based on Realistic Traffic

Our performance results of the realistic traffic without a known TCP variant using the machine learning and deep techniques are presented in Tables III and IV respectively. The respective confusion matrices are presented in Figures 5.

TABLE II: Benchmark data [22] experimental results without a known TCP variant using MLP and LSTM.

	MLP		LSTM	
OS	Precision	Recall	Precision	Recall
Android	0.75	0.92	0.77	0.85
Linux	0.90	0.82	0.83	0.85
Mac OS	0.62	0.81	0.58	0.83
Other	1.00	0.74	0.91	0.81
Unix	0.94	0.99	0.94	0.99
Windows	0.97	0.91	0.97	0.86
iOS	0.67	0.57	0.79	0.48
Average	0.84	0.82	0.83	0.81
Accuracy	**82.16%**		**81.04%**	

TABLE III: Realistic traffic experimental results without a known TCP variant using SVM, RF, and KNN.

	SVM		RF		KNN	
OS	Precision	Recall	Precision	Recall	Precision	Recall
Android	0.75	0.89	0.86	0.90	0.84	0.93
Linux	0.89	0.82	0.94	0.89	0.93	0.88
Mac OS	0.63	0.81	0.61	0.82	0.61	0.82
Unix	0.94	0.99	0.94	0.99	0.94	0.99
Windows	0.97	0.89	0.98	0.89	0.98	0.89
iOS	0.88	0.72	0.86	0.73	0.88	0.72
Average	0.85	0.83	0.86	0.85	0.87	0.85
Accuracy	**83.43%**		**85%**		**85.10%**	

TABLE IV: Realistic traffic experimental results without a known TCP variant using MLP and LSTM.

	MLP		LSTM	
OS	Precision	Recall	Precision	Recall
Android	0.81	0.83	0.76	0.86
Linux	0.89	0.79	0.90	0.81
Mac OS	0.61	0.82	0.82	0.79
Unix	0.92	0.99	0.94	0.99
Windows	0.98	0.89	0.97	0.89
iOS	0.84	0.73	0.70	0.92
Average	0.84	0.83	0.83	0.84
Accuracy	**83.91%**		**83.27%**	

C. Based on Emulated Traffic

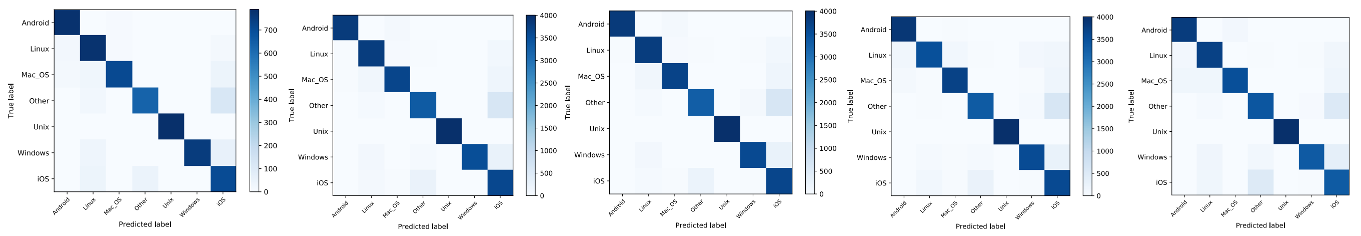
Our performance results of the emulated traffic without a known TCP variant as an input feature using both machine learning and deep learning techniques are presented in Tables V and VI respectively. As we can see in the corresponding confusion matrices presented in Figures 6, there is a slightly inaccurate classification of the Mac OS due to its underrepresentation. The precision and recall for the rest of the OSes using machine learning and deep learning techniques are reasonably good.

TABLE V: Emulated traffic experimental results without a known TCP variant using SVM, RF and KNN.

	SVM		RF		KNN	
OS	Precision	Recall	Precision	Recall	Precision	Recall
Android	0.74	0.90	0.86	0.90	0.85	0.91
Linux	0.92	0.82	0.94	0.89	0.92	0.90
Mac OS	0.63	0.81	0.61	0.82	0.61	0.82
Unix	0.94	0.99	0.94	0.99	0.94	0.99
Windows	0.97	0.89	0.98	0.89	0.98	0.89
iOS	0.88	0.73	0.86	0.73	0.88	0.73
Average	0.85	0.84	0.86	0.85	0.87	0.85
Accuracy	**84.67%**		**85.73%**		**85.27%**	

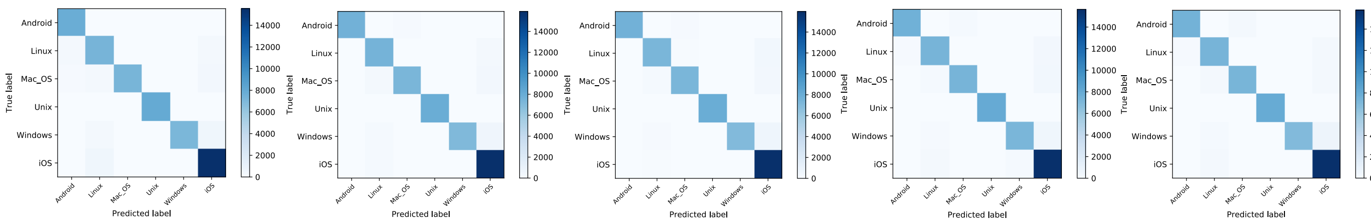
D. Comparison of Results Without Known TCP Variant

As shown in Tables I, II, III, IV, V, and VI, our experimental results are pretty consistent. Firstly, we can see that there is not much difference in performance across different machine learning and deep learning techniques. But more importantly, there are not many differences in performance between results from using different types of



(a) SVM (b) KNN (c) RF (d) MLP (e) LSTM

Fig. 4: Confusion matrix comparison of the machine learning and deep learning techniques using the benchmark data [22].



(a) SVM (b) KNN (c) RF (d) MLP (e) LSTM

Fig. 5: Confusion matrix comparison of the machine learning and deep learning techniques using a realistic traffic.

TABLE VI: Emulated traffic experimental results without a known TCP variant using MLP and LSTM.

	MLP		LSTM	
OS	Precision	Recall	Precision	Recall
Android	0.75	0.88	0.91	0.85
Linux	0.93	0.78	0.92	0.74
Mac OS	0.62	0.81	0.86	0.88
Unix	0.92	0.99	0.94	1.00
Windows	0.93	0.91	0.98	0.73
iOS	0.88	0.73	0.82	1.00
Average	0.85	0.83	0.89	0.88
Accuracy	**84.05%**		**88.44%**	

experimental data. This is intuitively correct, since the OS fingerprinting is based on the basic TCP/IP packet fields, and should not differ much between various types of data, whether we do evaluation using the benchmark data, real data or emulated data. Secondly, we believe accuracy in the range of 82-88% (average value) is perhaps not sufficient for a product in a real deployment. Our hypothesis is that this accuracy could be boosted considerably had we only known the implementation of the underlying TCP variant. We will explore this hypothesis in the next section.

VII. Oracle-based Experiment: Results using Oracle-given TCP variant

Here we assume that we know exactly the underlying TCP variant, i.e., we assume it is given by an Oracle. We show that knowledge of the TCP variant has a great potential for boosting passive fingerprinting of OSes, and in this section, we will try to quantify this potential. In the next section, we will show that much of this potential can be harvested by using a tool that predicts the TCP variant.

A. Based on Benchmark Data from Previous related Work

Tables VII and VIII show a significant performance gain across all classes of OSes when we assume prior knowledge of the underlying TCP variant, as compared to the results when the TCP variant is unknown presented in Tables I and II.

TABLE VII: Benchmark data [22] experimental results with Oracle-given TCP variant using SVM, RF and KNN.

	SVM		RF		KNN	
OS	Precision	Recall	Precision	Recall	Precision	Recall
Android	0.96	0.99	0.99	0.98	0.99	0.98
Linux	0.86	0.95	0.92	0.95	0.93	0.94
Mac OS	0.98	0.89	0.97	0.92	0.97	0.92
Other	0.93	0.81	0.93	0.81	0.90	0.83
Unix	1.00	1.00	1.00	1.00	1.00	1.00
Windows	0.99	0.89	0.97	0.92	0.99	0.91
iOS	0.75	0.89	0.75	0.91	0.76	0.91
Average	0.92	0.92	0.93	0.93	0.93	0.93
Accuracy	**91.71%**		**92.73%**		**92.69%**	

TABLE VIII: Benchmark data [22] experimental results with Oracle-given TCP variant using MLP and LSTM.

	MLP		LSTM	
OS	Precision	Recall	Precision	Recall
Android	0.96	0.97	0.94	0.97
Linux	0.89	0.92	0.88	0.93
Mac OS	0.96	0.92	0.97	0.88
Other	0.93	0.81	0.84	0.84
Unix	1.00	1.00	1.00	1.00
Windows	0.96	0.92	0.98	0.84
iOS	0.76	0.89	0.73	0.83
Average	0.92	0.92	0.91	0.90
Accuracy	**91.91%**		**90.03%**	

B. Based on Realistic Traffic

The performance results of the realistic traffic with the Oracle-given TCP variant presented in Tables IX and X show the potential of knowing TCP variant given by an Oracle for passive OS fingerprinting in a realistic scenario.

TABLE IX: Realistic traffic experimental results with Oracle-given TCP variant using SVM, RF and KNN.

	SVM		RF		KNN	
OS	Precision	Recall	Precision	Recall	Precision	Recall
Android	0.95	1.00	0.99	0.98	0.99	0.98
Linux	0.86	0.91	0.94	0.93	0.92	0.94
Mac OS	0.99	0.90	0.96	0.92	0.97	0.92
Unix	1.00	1.00	1.00	1.00	1.00	1.00
Windows	0.99	0.89	0.99	0.89	0.99	0.89
iOS	0.93	0.96	0.91	0.99	0.92	0.98
Average	0.95	0.95	0.96	0.96	0.96	0.96
Accuracy	**94.81%**		**95.65%**		**95.69%**	

TABLE X: Realistic traffic experimental results with Oracle-given TCP variant using MLP and LSTM.

	MLP		LSTM	
OS	Precision	Recall	Precision	Recall
Android	0.98	0.97	0.98	0.97
Linux	0.92	0.92	0.90	0.93
Mac OS	0.96	0.92	0.96	0.92
Unix	1.00	1.00	1.00	1.00
Windows	0.97	0.91	0.99	0.88
iOS	0.92	0.97	0.91	0.98
Average	0.95	0.95	0.95	0.95
Accuracy	**94.98%**		**94.89%**	

C. Based on Emulated Traffic

Our performance results of the emulated traffic with the Oracle-given TCP variant using both classical machine learning and deep learning techniques are presented in Tables XI and XII. We can see that this shows a significant improvement in performance over the results without a known TCP variant presented in Tables V and VI. Both machine learning and deep learning techniques have comparable and consistent results in terms of accuracy.

TABLE XI: Emulated traffic experimental results with the Oracle-given TCP variant using SVM, RF, and KNN.

	SVM		RF		KNN	
OS	Precision	Recall	Precision	Recall	Precision	Recall
Android	0.97	0.98	0.99	0.98	0.99	0.98
Linux	0.90	0.91	0.95	0.93	0.92	0.95
Mac OS	0.99	0.90	0.97	0.92	0.97	0.92
Unix	1.00	1.00	1.00	1.00	1.00	1.00
Windows	0.99	0.89	0.97	0.91	0.97	0.91
iOS	0.91	0.98	0.92	0.98	0.93	0.97
Average	0.95	0.95	0.96	0.96	0.96	0.96
Accuracy	**95.10%**		**96.02%**		**95.83%**	

TABLE XII: Emulated traffic experimental results with the Oracle-given TCP variant using MLP and LSTM.

	MLP		LSTM	
OS	Precision	Recall	Precision	Recall
Android	0.98	0.97	0.96	0.98
Linux	0.97	0.89	0.93	0.91
Mac OS	0.93	0.94	0.94	0.92
Unix	1.00	1.00	1.00	1.00
Windows	0.99	0.89	0.98	0.88
iOS	0.91	0.99	0.91	0.97
Average	0.95	0.95	0.95	0.95
Accuracy	**95.24%**		**95.08%**	

D. Comparison of Results with Oracle-given TCP Variant

Our accuracy results presented in Tables VII, VIII, IX, X, XI, and XII, demonstrate that by knowing the TCP variant we obtain a considerable performance boost in all our experimental results, compared to our previous results obtained without knowledge of the TCP flavor. With an Oracle-given TCP variant, we obtain a prediction accuracy of 94-96%, with an average value of 94.1% over all traffic classes and of 95.4% over only emulated traffic. The accuracy results are pretty consistent across all scenarios. Comparing these results with our previous results that do not use the Oracle (84.1% on average for all traffic types and 85.6% only for emulated traffic), we observe a solid increase in the OS fingerprinting performance. This improvement would significantly boost the usefulness of a product to be implemented in a real enterprise network infrastructure. As in the previous section, here again, we observe highly consistent performance results across different machine learning and deep learning techniques and also between the use of different types of experimental data. The latter is useful knowledge for the next section since it means that performance increases obtained over one traffic type is shown to be amenable to other traffic types as well. In the next section, we will have to base our evaluation on emulated data, since we do not have the TCP traffic patterns of the realistic data or benchmark data at hand. These traffic patterns are required to be able to passively infer the TCP variant in the experiments presented in the next section. In this section, the idealistic Oracle was used only to demonstrate the potential of knowing the TCP variant, but this is not a realistic assumption. Thus, in the next section, we will instead base our evaluation on a TCP variant that is passively predicted by a deep learning-based tool that we developed and presented in our previous work [11, 12, 13]. Using this tool, we explore how close our performance will get to the ideal solution of having an Oracle.

VIII. Prediction-based Experiment: Results using TCP variant prediction

In Section VII, we showed that Oracle-given knowledge of the TCP variant has a great potential for improving the passive OS fingerprinting. In reality, however, we don't have an Oracle-given TCP variant. Since passively detecting the TCP variant is a challenging task, this is where our tool from previous works on predicting the underlying TCP variant from passive measurements [11, 12, 13] comes into play. In this Section we use the TCP variant passively *predicted* by this tool as an input feature for the passive OS fingerprinting. The TCP variant is inferred from the famous Additive Increase and Multiplicative Decrease (AIMD) sawtooth pattern of TCP's estimated cwnd computed based on the outstanding bytes-in-flight. Since we don't have access to the actual cwnd of the senders in the benchmark data and realistic traffic, here we consider only the emulated traffic.

A. Based on Emulated Traffic

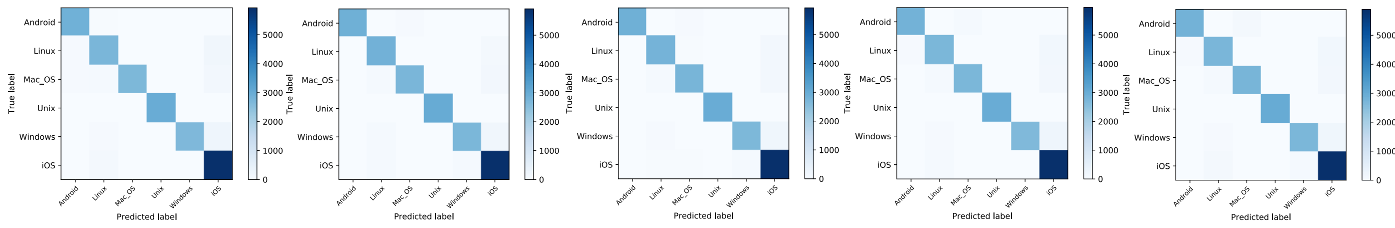
In this section, we use a tool to predict the TCP variant from passive measurements of TCP traffic patterns, and this prediction is used as input to the passive OS fingerprinting method presented above. The experimental results of both techniques are presented in Tables XIII and XIV.

TABLE XIII: Emulated traffic experimental results with predicted TCP variant using SVM, RF, and KNN.

	SVM		RF		KNN	
OS	Precision	Recall	Precision	Recall	Precision	Recall
Android	0.92	0.96	0.92	0.97	1.00	0.97
Linux	0.79	0.85	0.94	0.82	0.92	0.94
Mac OS	0.96	0.88	0.97	0.87	0.85	0.94
Unix	1.00	1.00	1.00	1.00	1.00	1.00
Windows	0.92	0.78	0.85	0.80	0.88	0.91
iOS	0.85	0.94	0.86	0.96	0.93	0.87
Average	0.90	0.90	0.91	0.91	0.93	0.93
Accuracy	**90.01%**		**91.09%**		**92.15%**	

TABLE XIV: Emulated traffic experimental results with predicted TCP variant using MLP and LSTM.

	MLP		LSTM	
OS	Precision	Recall	Precision	Recall
Android	0.95	0.97	0.92	0.96
Linux	0.98	0.79	0.86	0.90
Mac OS	0.95	0.90	0.95	0.88
Unix	1.00	1.00	1.00	1.00
Windows	0.94	0.77	0.97	0.77
iOS	0.82	0.99	0.88	0.96
Average	0.92	0.91	0.92	0.92
Accuracy	**91.45%**		**91.93%**	



(a) SVM (b) KNN (c) RF (d) MLP (e) LSTM

Fig. 6: Confusion matrix comparison of the machine learning and deep learning techniques using an emulated traffic.

B. *Comparison of results with a predicted TCP variant*

Results with emulated data and a passive prediction of the TCP variant (Tables XIII and XIV) gives an accuracy of 91.3% on average, which comes pretty close to the accuracy of 95.4% obtained on emulated traffic with the TCP-variant given by the Oracle. Intuitively, when we do learning based on the TCP variant prediction, the accuracy must be lower than the Oracle-given TCP variant, but the question is how close we can get to the idealistic scenario of having an Oracle. Our results show that using our tool for TCP variant prediction gives reasonably good OS fingerprinting accuracies that come close to the results obtained by using Oracle-given TCP variant. Even though the performance results with the TCP variant passively predicted by our deep learning-based tool are slightly lower as compared to the TCP variant given by an idealistic Oracle, our performance results of using our tool are reasonably competitive.

IX. Transfer Learning Results

Transfer learning is the ability to take a model trained in one scenario and apply it for classification in a different scenario. For example, in our case, that means we are able to train our model on a dataset created in an emulated network with an Oracle-given TCP variant and apply it for classification of our dataset from the realistic traffic. Results shown in Tables XV and XVI shows that the learning of the OS fingerprinter transfers well into other scenarios. Good transfer learning results indicate that our passive OS fingerprinting model is able to discern the results of unforeseen scenarios and still perform reasonably well. In previous works, we have also demonstrated that the TCP variant predictor performs well in terms of transfer learning [11, 12, 13].

TABLE XV: Transfer learning experimental results using SVM, RF, and KNN.

	SVM		RF		KNN	
OS	Precision	Recall	Precision	Recall	Precision	Recall
Android	0.95	1.00	0.98	0.98	0.99	0.98
Linux	0.86	0.91	0.90	0.95	0.92	0.95
Mac OS	0.99	0.90	0.98	0.92	0.97	0.92
Unix	1.00	1.00	1.00	1.00	1.00	1.00
Windows	0.99	0.89	0.98	0.90	0.97	0.91
iOS	0.93	0.96	0.93	0.97	0.93	0.97
Average	0.95	0.95	0.95	0.95	0.96	0.96
Accuracy	**94.79%**		**95.35%**		**95.76%**	

X. Conclusion and Future Work

In this paper, we proposed and evaluated a novel approach that attempts to passively fingerprint the underlying remote OS by leveraging *state-of-the-art* machine learning and deep learning techniques under multiple controlled scenarios. We show that knowing the Oracle-given TCP variant has a great potential for boosting the classification performance of passive OS fingerprinting. In our setting, we demonstrate that using the idealistic Oracle has the potential to boost the prediction accuracy from 84.1% to 94.1% on average across all traffic types tested, and from 85.6% to 95.4% in an emulated setting. However, in reality, we don't have the Oracle-given TCP variant and hence we don't know exactly the underlying TCP flavor. To address this, we demonstrated a method for passive OS fingerprinting where the cwnd is first computed based on the outstanding bytes-in-flight, then the underlying TCP flavor is predicted from the estimated cwnd, and finally, the predicted TCP variant is used as an input feature to detect the remote computer's OS. This is an additional feature that is added to the basic TCP/IP features that are the basis of OS fingerprinting in previous works. We demonstrate that our method performs significantly better than not using the predicted TCP variant as an input feature, increasing the accuracy in our experiment from 85.6% to 91.3%. The results of this method come close to the accuracy of 95.4% obtained by using the idealistic Oracle. To the best of our knowledge, this is the first study that reports the potential of the underlying TCP feature in boosting significantly the accuracy of passive OS fingerprinting. We further validate and demonstrate the transferability approach of our OSes classification models by conducting a series of controlled experiments against other scenarios. Through comparing the experimental results between the benchmark dataset, realistic, and emulated traffic in terms of accuracy and confusion matrix, it is clear that our passive OSes classification models are able to discern the results to unforeseen scenarios. Therefore, we are able to show that the learned passive OS fingerprinting model by leveraging a pre-trained knowledge of classification techniques from the emulated network performs reasonably well as it is shown in the experimental results when it is applied and transferred to a realistic scenario. Lastly, in all our experiments, we made sure that both the training and validation accuracies are closer which gives an idea about the ability of the OSes classification models to generalize on unforeseen scenarios.

TABLE XVI: Transfer learning experimental results using MLP and LSTM.

	MLP		LSTM	
OS	Precision	Recall	Precision	Recall
Android	0.97	0.98	0.97	0.96
Linux	0.95	0.85	0.91	0.91
Mac OS	0.94	0.94	0.96	0.90
Unix	1.00	1.00	1.00	1.00
Windows	0.99	0.89	0.98	0.87
iOS	0.90	0.98	0.90	0.98
Average	0.95	0.95	0.94	0.94
Accuracy	**94.72%**		**94.28%**	

The method presented in this paper, where the cwnd is first computed based on the outstanding bytes-in-flight, then the underlying TCP flavor is predicted from the estimated cwnd, is particularly efficient for loss-based TCP variants. In previous works, we have also developed a tool for the prediction of delay-based TCP flavors [10]. We plan to extend the method presented in this paper to also cover delay-based TCP variants and present it in a follow-on paper [14]. Note that passively detecting the TCP variant is a challenging task, which led to a two-step approach, where the TCP variant prediction of a deep learning-based tool is used as input to another machine learning method in the next step. However, by integrating the two machine learning approaches better, there should be potential for increasing the performance even further and get even closer to the idealistic results of using an Oracle. Exploring such optimizations is also left for future work. It is known that TCP clock drift improves OS fingerprinting and hence measuring differences in the timing of how the IP stack works may allow us to predict the underlying OS with greater assurance in terms of accuracy. We, therefore, argue for using other TCP options like timestamps and queueing delay characteristics as an input feature vector for passive OSes fingerprinting model as another interesting direction. Finally, in addition to the difficulties of establishing ground truth (e.g., the TCP variant) at a larger scale on a dynamic network addressed in Section III, there is a lot of other work to be done as an extension of our work presented here. For example, addressing answers to valid questions like: *What happens if an end-user (client) changes default parameters that are the basis of OS fingerprinting?* is one possibility for our future work. We expect that end-users don't change parameters often, while servers may do so if it helps improve performance. We believe this would make OS fingerprinting potentially hard.

References

[1] 5G4IoT. 5G4IoT, 2019.

[2] A. Aksoy, S. Louis, and M. H. Gunes. Operating system fingerprinting via automated network traffic analysis. In *IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2017.

[3] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-based congestion control. 2016.

[4] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet security: repelling the wily hacker*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[5] N. Davids. Initial TTL values, 2011.

[6] ESnet. iperf3, 2017.

[7] J. Fan, J. Xu, M. H. Ammar, and S. B. Moon. Prefix-preserving IP address anonymization: measurement-based security evaluation and a new cryptography-based scheme. 2004.

[8] L. G. Greenwald and T. J. Thomas. Toward Undetected Operating System Fingerprinting. *WOOT*, 7:1–10, 2007.

[9] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS*, 2008.

[10] D. H. Hagos, P. E. Engelstad, and A. Yazidi. Classification of Delay-based TCP Algorithms From Passive Traffic Measurements. In *18th NCA*. IEEE, 2019.

[11] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. A machine learning approach to TCP state monitoring from passive measurements. pages 164–171. IEEE, 2018.

[12] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. Recurrent Neural Network-based Prediction of TCP Transmission States from Passive Measurements. In *NCA*, pages 1–10. IEEE, 2018.

[13] D. H. Hagos, P. E. Engelstad, A. Yazidi, and Ø. Kure. Towards a Robust and Scalable TCP Flavors Prediction Model from Passive Traffic. In *2018 27th ICCCN*, pages 1–11. IEEE, 2018.

[14] D. H. Hagos, A. Yazidi, P. E. Engelstad, and Ø. Kure. A Deep Learning-based Universal Tool for Operating Systems Fingerprinting from Passive Measurements. 2020. Submitted for publication.

[15] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno modification to TCP's fast recovery algorithm. RFC 6582, 2012.

[16] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 1989.

[17] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*. ACM, 1988.

[18] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. RFC 1323, 1992.

[19] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[20] T. Kohno, A. Broido, and K. C. Claffy. Remote physical device fingerprinting. *IEEE*, 2005.

[21] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. pages 183–196. ACM, 2017.

[22] M. Lastovicka, T. Jirsik, P. Celeda, S. Spacek, and D. Filakovsky. Passive os fingerprinting methods in the jungle of wireless networks. In *NOMS*. IEEE, 2018.

[23] R. Lippmann, D. Fried, K. Piwowarski, and W. Streilein. Passive operating system identification from TCP/IP packet headers. In *Data Mining for Computer Security*. Citeseer, 2003.

[24] R. Lippmann, S. Webster, and D. Stetson. The effect of identifying vulnerabilities and patching software on the utility of network intrusion detection. Springer, 2002.

[25] G. F. Lyon. Remote OS detection via TCP/IP stack fingerprinting. *Phrack Magazine*, 8(54), 1998.

[26] G. F. Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. 2009.

[27] Netresec. NetworkMiner, 2007.

[28] A. Ornaghi and M. Valleri. Ettercap, 2015.

[29] J. Postel. Internet control message protocol. RFC 792, 1981.

[30] J. Postel. Internet protocol. RFC 791, 1981.

[31] J. Postel. Transmission control protocol. RFC 793, 1981.

[32] F. Rosenbaltt. The perceptron–a perciving and recognizing automation. *Cornell Aeronautical Laboratory*, 1957.

[33] J. Scambray, S. McClure, and G. Kurtz. *Hacking exposed*. McGraw-Hill Professional, 2000.

[34] SCOTT. European Leadership Joint Undertaking, 2019.

[35] R. Spangler. Analysis of remote active operating system fingerprinting tools. *University of Wisconsin*, 2003.

[36] G. Taleck. Synscan: Towards complete tcp/ip fingerprinting. *CanSecWest, Vancouver BC, Canada*, pages 1–12, 2004.

[37] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound TCP approach for high-speed and long distance networks. In *Proceedings IEEE INFOCOM*, 2006.

[38] W. Wei, K. Suh, B. Wang, Y. Gu, J. Kurose, and D. Towsley. Passive online rogue access point detection using sequential hypothesis testing with TCP ACK-pairs. ACM, 2007.

[39] J. Xu, J. Fan, M. Ammar, and S. B. Moon. On the design and performance of prefix-preserving IP traffic trace anonymization. In *ACM SIGCOMM*, pages 263–266. ACM, 2001.

[40] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. IEEE, 2004.

[41] F. Yarochkin and O. Arkin. Xprobe2- A'Fuzzy'Approach to Remote Active Operating System Fingerprinting, 2002.

[42] M. Zalewski. p0f: Passive OS fingerprinting tool. *Online at http://lcamtuf.coredump.cx/p0f3*, 2017.

[43] B. Zhang, T. Zou, Y. Wang, and B. Zhang. Remote operation system detection base on machine learning. IEEE, 2009.