nPrint: A Standard Data Representation for Network Traffic Analysis

Jordan Holland[1], Paul Schmitt[1], Nick Feamster[2], Prateek Mittal[1]
[1] Princeton University
{jordanah, pschmitt, pmittal}@princeton.edu
[2] University of Chicago
feamster@uchicago.edu
https://nprint.github.io/nprint

ABSTRACT

Conventional detection and classification ("fingerprinting") problems involving network traffic commonly rely on either rule-based expert systems or machine learning models that are trained with manually engineered features derived from network traffic. Automated approaches in this area are typically tailored for specific problems. This paper presents nPrint, a *standard*, packet-based representation of network traffic that can be used as an input to train a variety of machine learning models without extensive feature engineering.

We demonstrate that nPrint offers a suitable traffic representation for machine learning algorithms across three common network traffic classification problems: device fingerprinting, operating system fingerprinting, and application identification. We show that models trained with nPrint are at least as accurate as widely used tools, but in contrast do not rely on brittle, manually updated rules and features. Finally, we release nPrint as a publicly available software tool to encourage further use, testing, and extensions to the existing network traffic representation.

1 INTRODUCTION

Identifying or "fingerprinting" network devices, operating systems, or applications based on network traffic is a common problem area in networking. From the network operations point of view, operators often need to identify applications and devices for a variety of reasons, including identifying rogue devices or determining vulnerable devices on the network [11]. In the realm of security and privacy, fingerprinting may constitute a threat to privacy, and thus understanding the capabilities and limitations of current fingerprinting techniques can help shed more light on privacy risks [9, 13, 25].

Fingerprinting typically relies on a combination of active or passive network measurement, collection of corresponding traffic, and commonly involves application of a set of rules to the resulting traffic capture. For example, one common tool in device and operating system fingerprinting is Nmap, which sends packets to devices to elicit responses that are characteristic of a particular device and applies a fixed set of rules to the corresponding responses; p0f , another popular fingerprinting tool, fingerprints operating systems by examining passively captured network traffic and matching aspects of that traffic (specifically, operating system-specific defaults such as TCP options) to a set of rules that identify the operating system [19, 23]. More broadly, the field of traffic analysis aims to identify application traffic, devices, web browsing behavior, and even human behavior from network traffic patterns.

Yet, while the area of traffic fingerprinting is broad, many fingerprinting tasks have generally involved devising a specific set of fingerprints (i.e., features and rules) for the corresponding task [6, 9, 13, 17, 25, 33, 38]. Crafting these rules is typically painstaking and manual; worse, the rules themselves are *brittle*: as new devices, applications, operating systems, software updates, and behaviors emerge, the rules need to be updated.

The emergence of a variety of supervised and unsupervised machine learning algorithms creates new opportunities for learning features and rules that can perform fingerprinting. The success of any machine learning approach ultimately depends on presenting the models with appropriate *representations*. Even in cases where traffic classification relies on machine learning (e.g., denial of service attack detection, botnet detection), traffic classification typically involves significant manual feature engineering to derive features that are characteristic of the classification task, ultimately training a model on those specific features. In many cases, feature engineering requires a significant amount of effort. Even with expert domain knowledge, the exploration and engineering process is manual, and subject to a focus on unimportant features or omission of features that either were not immediately apparent or involve complex relationships (e.g., non-linear relationships between features). Furthermore, assumptions and uses can change over time, rendering models and hand-crafted features obsolete.

This paper explores takes a different approach, exploring whether a single, standard network traffic representation can apply to a broad array of traffic classification tasks. A primary contribution of this paper is an in-depth evaluation of whether a standard encoding of network traffic data can yield as accurate performance for a variety of fingerprinting tasks as bespoke representations and models that are tailored to each task. We design a general packet representation, *nPrint* that encodes data in a *normalized, bit-aligned, space-efficient* format that facilitates representation learning which automatically discovers the semantically important parts of network packets. We apply nPrint to three network traffic fingerprinting problems—devices, operating systems, and applications—using three different machine learning models: a random forest model, a multi-layer perceptron, and a convolutional neural network. We find that models trained with nPrint representations for these three tasks work as well (or better than) existing approaches, for a variety of models, suggesting the promise for a universal, standard representation of traffic that can apply to a variety of fingerprinting tasks.

We have implemented nPrint in a publicly available open-source tool for others to use and extend. We envision extensions to this work in a number of areas: First, although we have demonstrated nPrint on three broad classes of fingerprinting problems, we encourage others to test the representation on a broader set of traffic analysis problems. Second, the representation can be extended to capture other aspects of network traffic that could be useful for fingerprinting or classification: for example, nPrint supports representations on collections of packets (e.g., the first N packets in a sequence) as well as temporal representations (e.g., packet inter-arrival times) through relative timestamps. Future work involves exploring how these aspects of the nPrint representation can potentially improve the accuracy of the fingerprinting methods we present in this paper.

The rest of the paper proceeds as follows. Section 2 describes background on machine learning models and related work applying these models to device and OS identification and fingerprinting problems. Section 3 presents nPrint in detail. Section 4 describes how nPrint fits into a standard machine learning pipeline, as well as evaluating the performance of nPrint in terms of packet transformation time, lines of code necessary to customize nPrint for a given problem, and the amount of time to train models used in this work. Sections 5, 6, and 7 evaluate both the various representations and their applications to three contexts: active device fingerprinting , passive OS detection, and application identification through DTLS handshakes. Section 8 concludes with a discussion of open questions and future directions.

2 BACKGROUND AND RELATED WORK

In this section, we provide background on machine learning models and their applications and survey related work in fingerprinting.

2.1 Machine Learning Models

Deep learning techniques have been extensively used in other fields, including computer vision and image recognition, to accurately classify images. *Deep neural network (DNN)* architectures, such as convolutional neural networks and mutli-layer perceptrons (MLP) are two of the most extensively used deep learning architectures for supervised learning. *Multi-layer perceptrons (MLP)* are simple feed-forward neural network architectures that consist of an input layer with a number of neurons equal to the number of features in the data, any number of hidden layers that are fully connected to the next, and an output layer with a number of neurons equal to the number of classes in the supervised learning problem [12].

Convolutional neural networks (CNNs) are another extensively used DNN architecture. CNNs consist of a *convolutional* layer, which transforms input features using *kernels*, $M \times N$ matrix that effectively applies a spatial filter (e.g., a smoothing operation) to the resulting input values [12]. Subsequently, *pooling* operation is applied to reduce the dimensionality of the features by replacing each output with a summary statistic of a group of outputs. After one or more convolutional layers, the network consists of one or more fully connected layers and an output layer much like an MLP. DNN architectures have been used extensively in cases where it is possible to use raw data (e.g., images, audio) as input, without engineering features *a priori*; as a result, DNNs have been popular in image recognition and computer vision, where it is straightforward to provide raw pixels as input to a model.

2.2 Fingerprinting

Past research has applied both manual and automated approaches to fingerprint networked devices and applications.

TCP-based host fingerprinting. Idiosyncrasies between TCP/IP stack implementations have often been the basis of networked host fingerprinting techniques. Actively probing to differentiate between TCP implementations was introduced by Comer and Lin [7]. Padhye and Floyd identified differences between implementations [24]. Paxson passively identified TCP implementations passively using traffic traces [26].

Past work has also developed techniques to fingerprint host operating systems. There are multiple tools and methods for host OS fingerprinting, using both active and passive techniques. One common tool is Nmap [19], which sends

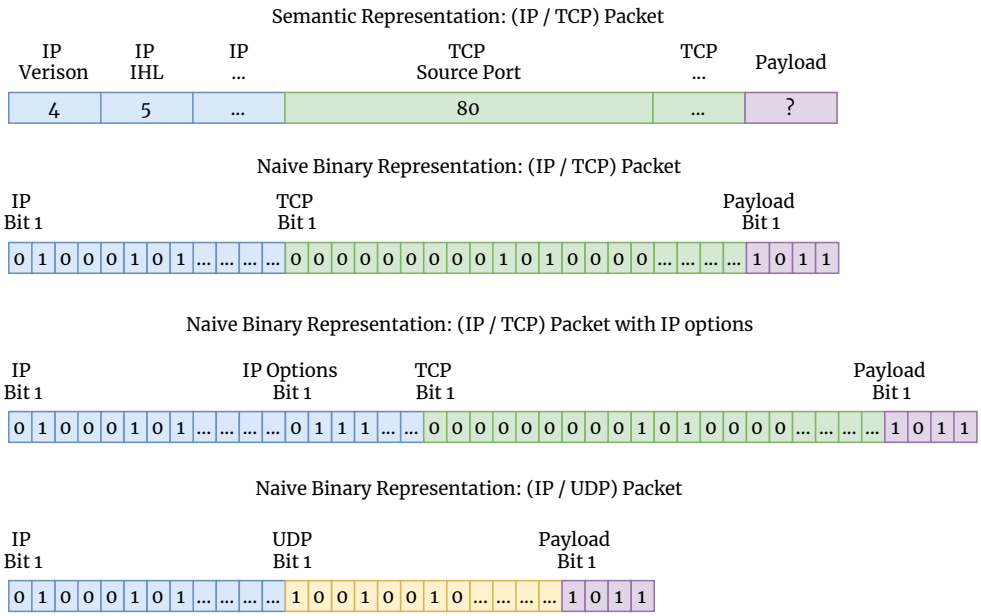


Figure 1: *Semantic and naive binary packet representations introduce various problems that make modeling difficult: Semantic representations may assign relationships to continuous values where distance is meaningless (e.g., port numbers). Naive binary representations lack alignment, which can confuse model training.*

probes and examines the responses sent by a target host, focusing on TCP/IP settings and Internet Control Messaging Protocol (ICMP) implementation differences between different operating systems and devices. Nmap is widely considered the "gold standard" of active probing tools.

Passive OS identification aims to identify operating systems from passively captured network traffic [5, 18]. P0f passively observes traffic and determines the operating system largely based on TCP behavior [23]. In contrast, the research does not focus on heuristics and *a priori* knowledge of implementation differences between host networking stacks. Instead, we allow the model to learn the differences during training.

Remote fingerprinting can be used to characterize aspects of the remote system other than its operating system or networking stack. Clock skew information determined from the TCP timestamp option was leveraged to identify individual physical devices by Kohno *et al.* [17]. Formby *et al.* passively fingerprint industrial control system devices [10].

Machine learning-based fingerprinting. Machine learning techniques have been long used for network traffic classification and fingerprinting [2, 3, 35, 42]. Wang *et al.* developed an ML-based classification model to detect obfuscated traffic [36]. Sommer and Paxson demonstrated that using machine learning to detect anomalies can have significant drawbacks, as network anomalies can exhibit different behavior than other problems solved by ML [32]. Many recent works have used machine learning to identify websites visited through the Tor network [14, 25, 37, 39].

Machine learning techniques have recently garnered attention as they have proven to be applicable to the task for inferring information from encrypted network traffic. Various work has used machine learning models to fingerprint websites visited through the Tor network [22, 28, 31]. These works differ from this work, due to their focus on the Tor setting. In Tor, all packets are the same size, meaning network traffic in Tor can be represented by a series of -1s and 1s that represent the direction of the traffic. This work instead considers traffic over any network that can vary in size and protocol. Trimananda *et al.* used DBSCAN to identify smart home device actions in network traffic [34].

Deep learning techniques have become popular for network traffic classification problems [1, 16, 41, 43]. Yu *et al.* used convolutional autoencoders for network intrusion detection [43]. Wang *et al.* argued one-dimensional convolution neural networks are appropriate for network traffic rather than two dimensional CNNs [41]. In contrast, nPrint aligns packets, building two-dimensional representations over time as packets are collected. Wang *et al.* applied an off-the-shelf deep learning techniques from image recognition and text analysis to intrusion detection; in contrast, we focus specifically on creating a general representation for network traffic that can be used in a variety of different models, across a broad class of problems[40]. Our results also suggest that

Wang *et al.*'s model may be more complex than necessary, and that better input representations such as nPrint could result in simpler models.

3 THE NPRINT REPRESENTATION

In this section, we enumerate the design requirements for a standard data representation, explore various strawman representations and explain why they are not suitable, and describe the nPrint representation.

3.1 Design Requirements

The essence of nPrint involves transforming packets into a representation that can be learned from and the ability to transform all parts of each packet into the resulting representation. As detailed in Section 2, deep learning techniques have proven to work well with problems related to image classification and computer vision, where pictures have a standard representation of values that can easily be represented as an NxM image of 0-1 normalized pixel values. Unfortunately, network traffic does not easily lend itself to this format. First, packet lengths vary considerably. Second, and equally problematic, different packet types contain entirely different information (*e.g.*, a UDP packet cannot be represented as a TCP packet). To create a general representation of network traffic that is usable across both different classification techniques and different problems, we define a list of requirements for a packet representation:

- **Complete:** each feature is represented for every packet: every bit of data in each packet must be able to be included in the representation.
- **Aligned:** every location in the representation has the same meaning across all packets, which allows models to perform representation learning according to fixed locations in the packet bitfield.
- **Constant size:** the size of the representation is the same for each packet.
- **Normalized:** all features lie between 0 and 1.
- **Efficient size:** the size of the representation for each packet can be feasibly used to train deep learning models.

3.2 Strawman Representations

To put nPrint in context, we discuss possible representations that can be used for network traffic in learning pipelines.

Semantic network representation. A semantic view of network traffic involves packets being broken up into headers, with each header broken into header fields, such as TCP source port or IP total length fields. One straightforward representation of a packet that can be used for training models involves directly encoding each header value as a feature. This representation is aligned, and can capture many of the header fields. Figure 1 shows an example of the semantic representation.

This conventional, semantic view of network traffic has several drawbacks. First, it is not complete. Although it captures named header fields that are easily represented as integers, parts of the packets with less structure are much harder to represent to a model. For example, the payload of a packet has no consistent semantic mapping, making it incredibly difficult to capture in a named feature.

Another subtle issue with encoding each specific field with its corresponding value is the loss of ordering. For example, IP and TCP headers have options section where a host can encode options specific to the packet being sent. The order of these options has been used as a predictive feature for some classification problems, which is lost in a semantic representation that parses each option [19].

Finally, semantic representations may encounter difficulties when mapping values from their semantic meaning to a continuous value space for training models. Network ports exhibit a unique problem for semantic representations: they do not correctly map to a continuous valued feature. For example, TCP port 79 is the Finger protocol, while TCP port 80 is used for HTTP traffic. Although these ports are close numerically, they have no semantic correlation with each other.

Finally, the semantic representation is not normalized. Normalizing the values for each feature in each classification problem requires extra passes of the input data to scale each value, which must be carefully considered as the size of network traffic traces grows quickly.

One-hot encoded semantic representation. A typical method for avoiding issues with continuous value representation is to use a one-hot-encoding of specific features. A one-hot-encoding (OHE) of a feature transforms one feature with n different known values into n binary features with possible values of 0 or 1. However, one-hot-encoding features across different types of packet headers quickly results in a packet representation that violates the efficient size requirement. For example, one-hot encoding the TCP source port alone could add over 65,000 features to a representation for a single packet. Sequence numbers, acknowledgement numbers, and checksums further contribute to this explosion of features. One-hot-encoding also requires processing the entire dataset for each problem to determine the n values that exist for each feature.

Naive binary representation. Rather than relying on semantic information, we can instead employ a representation that simply models each packet as a raw bitmap. This choice leads to a consistent, normalized, efficient size representation that creates a 1xM "image" of each packet which can be directly fed into machine learning models. We see an example of this

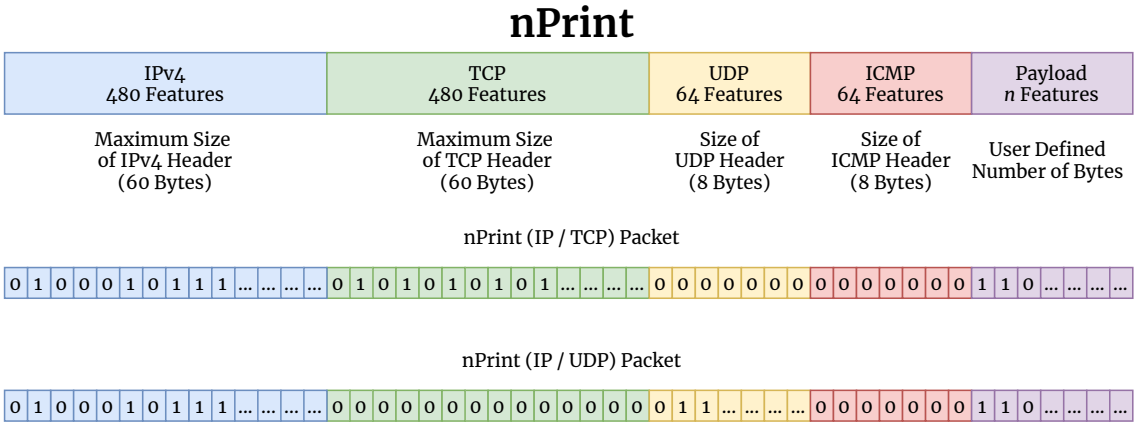


Figure 2: *nPrint, the complete, aligned packet representation. Headers that do not exist in the packet being transformed are zero filled, while headers that exist but are not of maximum size are zero padded for alignment across nPrints.*

representation in Figure 1. Transforming each packet into its bitmap representation ignores many of the intricate details that must be considered when modeling network traffic, namely varying size and different types of packets. These issues can cause two packets to have different meanings for bit i in the representation. For example, a TCP packet and a UDP packet would have completely different values represented at the same bit location. Figure 1 shows this problem in detail.

Worse, this problem can occur even within two packets of the same type. For example, an TCP/IP packet with IP options and an TCP/IP packet without IP options will cause the bits to be misaligned in the two representations. Misalignment manifests itself in two ways: 1) the resulting representation is not interpretable, as we cannot correctly map each bit in the representation back to a semantic meaning; and 2) it can decrease model performance as the misaligned bits introduce noise in the model where important features may exist.

3.3 nPrint

We build upon the naive binary representation of a packet to create a representation that meets all of the requirements. Figure 2 introduces nPrint, a single packet representation which is designed to be directly used with machine learning methods, allowing models to *learn* important characteristics of the traffic, rather than manually encoding it. nPrint meets all of our design requirements: it is complete, aligned, constant size, normalized, and of efficient size. nPrint is complete: any packet can be represented. It is aligned and constant size: using internal padding and including space for each header type regardless of whether that header is actually present in a given packet ensures that each packet is represented in the same number of features, and that each feature has the same meaning. Alignment gives nPrint a distinct advantage over many network representations in that it is interpretable at the bit level. This allows for researchers and practitioners to map nPrint back to the semantic realm to better understand the features that are driving the performance of a given model. For instance, for the model to learn that the TCP RST flag is an important feature for a given problem, it must be in the same bitmap location across all packets. Not all *models* are interpretable, but by having an interpretable *representation*, we can better understand models that are. nPrint is also normalized: by directly using the bits of the packets and filling missing values with 0, each feature lies between 0 and 1. Finally, nPrint is efficiently sized: the size of the representation in Figure 2 is 1,088 + n payload features for a packet, where n is defined by the user. We make the payload features optional in nPrint as many problems may deal with encrypted traffic.

A single nPrint represents one packet. Many classification problems require sets of packets. nPrint can be easily extended to multi-packet learning problems (in about 20 lines of Python code). If we consider each single nPrint as a 1xM image, where M is the number of features in the fingerprint, we can create a NxM nPrint for multi-packet learning problems. Each NxM packet image can then be used as a single sample for training different model architectures.

Finally, nPrint is extendable and modular: protocols, such as ICMP, can be easily added or removed from the representation without invalidating any of the representation requirements or deriving new features from the protocols themselves.

4 THE NPRINT PIPELINE

This section discusses in detail each step of the pipeline we use in this work. Figure 3 shows a conventional machine learning pipeline and where nPrint fits in the pipeline.

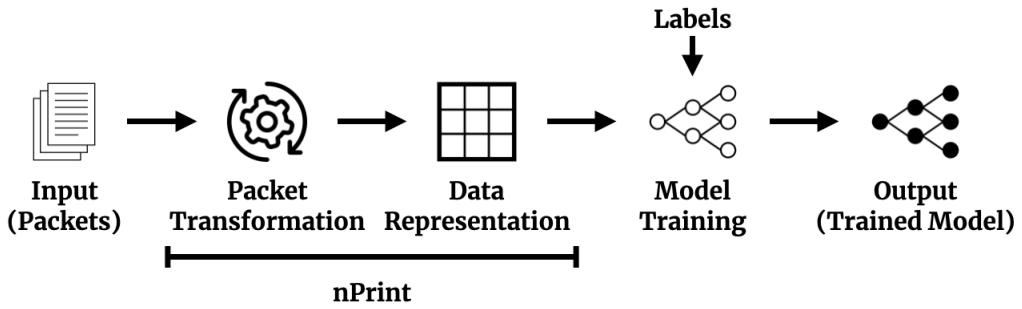


Figure 3: *nPrint results in a general network traffic representation that can be applied to a variety of machine learning methods and tasks.*

4.1 Input

nPrint takes two types of inputs. First, standard PCAPs where network traffic is most often recorded. Second, hex-encoded strings of packets in a CSV format which are compact and created as output of some network scanning tools such as Zmap [8].

4.2 Packet Transformation and Data Representation

nPrint transforms each packet into the standard nPrint representation described in Section 3. We implement nPrint in C++ to efficiently transform packets at about 1 million packets per minute on a single thread. We further explore the performance of nPrint in Section 4.4. Although we do not examine problems that concern time information such as website fingerprinting, we have implemented in nPrint the option to include a relative timestamp to capture time-series information. Finally, we do not consider IPv6 in this work, but we have implemented IPv6 fixed header parsing in nPrint, which will be included in the tool.

4.2.1 Compressing nPrint to Decrease Training Time. A nPrint is of efficient size, but there remain multiple avenues to compress the number of features that must be considered for a given classification problem. We aim to compress the representation to reduce disk use and decrease model training time. We notice that many parts of a packet header are either unused or static. For instance, the reserved bits in the TCP and IPv4 headers can likely be dropped from the packet representation, along with other parts of the header that are essentially constant values, such as the IPv4 version number bits.

We compress a nPrint by removing all features in the dataset that have no variance. Doing so eliminates features that are not useful for classification, reduces the size of the DNN models, and increases the speed of training and classification. This compression method also results in a representation that remains interpretable.

Other methods of compressing a nPrint are possible. For instance, tailoring which packet headers to include in the final representation is a quick method to reduce the representation size. For example, a nPrint can be reduced by 480 features if TCP traffic is not of interest for a given classification problem. In each case, the representation requirements mentioned in Section 3.2 still hold. This method is useful to reduce disk space while collecting traffic traces. We note here that while this compression method requires "expert intervention", the question of "*is TCP traffic important for the problem?*" is much easier to answer than "*how do we choose and represent each field in the packet?*"

Perhaps the most efficient way to compress a nPrint is by use of an Autoencoder. *Autoencoders* are DNN architectures that learn efficient data representation for sets of data by reducing the dimensionality of the data. An autoencoder learns representations in an unsupervised setting, wherein it compresses a sample of data and then attempts to rebuild the original sample from the compressed form. Autoencoders learn to create a compressed representation that contains information almost equivalent to the original representation. Although an autoencoder may create the most efficient representation of a nPrint, we choose not to use this method in this work as the new representation would not be interpretable. We highlight their use for compression here for cases where representation size is of higher importance than interpretability.

4.3 Model Training

nPrint is a single packet representation that is usable in multiple types of models, including DNN architectures and ensemble learning techniques. We choose to train and test nPrint on three types of models: multi-layer perceptrons, convolutional neural networks, and random forests. Details of MLPs and CNNs are explained in Section 2. Here we choose these models to test both deep learning and ensemble learning techniques using nPrint, but nPrint can be combined with any model, as well as unsupervised learning techniques.

Metrics. We define a *false positive* for class C as any sample that is not of class C, but misclassified as class C by the classifier. A *false negative* for class C is any sample of class C that is not classified as class C. We then evaluate each trained model using multiple metrics including accuracy, ROC AUC, and F1 scores. We use a *balanced accuracy score* to account for any class imbalance in the data. In the multi-class classification case we consider ROC scores in a "one vs rest" manner, where each class C is considered as a binary classification task between C and each other class. F1 scores represent a weighted average of precision and recall. In the multi-class classification task we report an F1 score that is calculated by counting the total number of true positives, false positives, and false negatives.

Multi-layer Perceptron (MLP). We use an 8 layer model for the MLP, starting with an input layer consisting of a number of neurons equal to the number of features in the given representation. The input layer is followed by 4 successive pairs of fully connected layers consisting of 5,000, 5,000, 5,000 and 2,500 neurons and dropout layers with a rate of 0.25, 0.25 0.25, and 0.50 respectively. The output layer consists of a number of neurons equal to the number of classes in the dataset. The activation function for the output layer is set to the sigmoid function for multiclass classification tasks and softmax for binary classification tasks. We use a SGD optimizer and set the loss function to categorical cross-entropy loss for multiclass classification and binary cross-entropy loss for binary classification tasks.

Convolutional Neural Network (CNN). We use an 11 layer VGG-based convolutional network for the CNN starting with a convolutional layer consisting of 32 convolutional filters each of size 3x3, an input shape equal to the size of the NxM representation being trained on, and a reLU activation function. This layer is followed by an identical convolutional 2D layer and then a max pooling layer of size 2x2. The max pooling layer is followed by a flatten layer and three pairs of fully connected layers consisting of 2,000, 2,000, and 1,000 neurons followed by dropout layers with a dropout rate of .5, .25, and .25 respectively. Finally, we have a fully connected layer of 500 neurons followed by an output layer fit with a sigmoid activation function for multiclass classification and softmax for binary classification. We use a SGD optimizer and set the loss function to categorical cross-entropy loss for multiclass classification tasks and binary cross-entropy loss for binary classification tasks.

Random Forest (RF). To directly interpret the feature importance of each bit in nPrint, we consider a third, non-DNN model using a random forest ensemble classifier. We specifically choose random forests as the binary features in nPrint lend themselves to a decision tree structure. Random forest models do not consider two-dimensional sets of features. Therefore, we must flatten any two-dimensional feature vector before using the classifier. We use SciKit Learn's random forest implementation for the classifications, with 1,000 estimators and a "balanced" class weight to account for any imbalances in dataset labels [27].

Problem	# Packets	nPrint Transformation (Seconds)	LOC (Python)
Active Fingerprinting	274,010	14	16
Passive OS Detection	1,343,920	54	27
Application Identification	46,816	12	33

Table 1: *nPrint currently transforms over 1 million packets per minute on a single thread.*

4.4 Performance Evaluation

Finally, we implement, evaluate, and publish (upon publication) nPrint, as well as the entire pipeline presented in Figure 3. We believe that performance must be accounted for when employing the method shown in this work. To this end, we have implemented nPrint in C++, which transforms each packet in an input file into a nPrint. We evaluate the performance of the pipeline on a system with two 8-core, 2.6 GHz CPUs (Intel Xeon E5 2640) and 128 GB of RAM. Deep learning models are trained on a machine with two NVIDIA TITAN RTX graphics cards, with a total of 64 GB of RAM.

Table 1 shows the performance of nPrint on each of the problems we have evaluated in this work. Ultimately we find that nPrint can currently transform packets at over 1 million packets per minute, on a single thread. If higher performance is required, one can run the packet transformer on multiple PCAPs at once. We also note that nPrint can be run on systems with incredibly limited RAM, as only a single packet is stored in memory at any given time.

nPrint can currently transform IPv4, fixed IPv6 headers, UDP, TCP, ICMP, and payloads. nPrint also has the capability to include a relative timestamp field for problems that may need timing information. We demonstrate the flexibility of nPrint in Section 7.3, where we used nPrint to rapidly test the performance of different constraints on the problem.

Each problem to be solved with nPrint does require some tuning to match nPrint with labels. Table 1 shows the minimal amount of effort required to test each additional type of problem, requiring as little as 16 lines of Python code to modify nPrint for the active fingerprinting problem. All of this work is associating each nPrint with a label for training, not additional feature engineering.

Finally, Table 2 shows the amount of training time for each model we train on each problem. All models are trained on each dataset in under 30 minutes. Each row represents a single fold of the data when performing cross validation.

Problem	Model Training Time (Minutes)		
	CNN (150 Epochs)	FNN (150 Epochs)	Random Forest
Active Fingeprinting	20	10	1
Passive OS Detection	25	10	6
Application Identification	5	3	0.5

Table 2: *Approximate model training time for each separate problem we examine. nPrint supplies a standard representation for models to efficiently train on.*

Vendor	Device Type	Probed Devices	Responsive Devices
Cisco	Network Device	1,500	1,451
Miktrotik	Network Device	1,500	1,358
Huawei	Network Device	1,500	1,409
H3C	Network Device	1,500	1,380
NEC	Network Device	1,500	1,450
Lancom	Network Device	1,500	1,426
Juniper	Network Device	1,500	1,445
Adtran	Network Device	1,500	1,449
ZTE	Network Device	1,500	1,425
Ubiquoss	Network Device	1,500	1,476
Dell	Network Device	1,500	1,449
Avtech	IoT Camera	2,672	2,152
Axis	IoT Camera	2,900	2,653
Chromecast	IoT Streaming	2,984	2,872
Roku	IoT Streaming	2,966	2,403

Table 3: *The active device fingerprinting dataset.*

Ultimately, we find that nPrint, combined with machine learning models, allows for rapid testing of classification problems, allowing researchers and practitioners to spend a larger amount of time understanding model performance rather than developing features for the models.

5 ACTIVE DEVICE FINGERPRINTING

We first examine the utility of nPrint to extract features from packets in active device fingerprinting. Specifically, we compare the performance of models trained with nPrint to Nmap, one of the most popular device fingerprinting tools, which has been developed for over 20 years.

Nmap's approach to fingerprinting remote devices is to send specifically crafted probes which have been fine-tuned over the course of the tool's lifetime. Nmap's detection system consists of 16 probes: 13 TCP, 2 ICMP, and 1 UDP. These probes are designed to elicit responses that give insight into quirks of different device types, such as TCP sequence generation, TCP options, and ICMP response behavior. Nmap transforms the responses to the probes into a fingerprint of the device using a collection of over 25 tests. Table 4 shows a summary of the tests Nmap performs on the responses to the probes it sends. Many of Nmap's most complex tests are performed by examining data from both the sent probes and the responses to those probes, comparing sent and received values.

5.1 Input (Packets)

We run Nmap on a collection of labeled devices to compare nPrint's performance to Nmap's hand engineered features. Holland *et al.* previously examined using a *subset* of Nmap's probes to fingerprint network device vendors at Internet scale [15]. They curate a labeled dataset of network devices through an iterative clustering technique on SSH, Telnet, and SNMP banners, which provides a list of labeled network devices to Nmap.

Although this previous work was concerned with fingerprinting devices at scale, we are only concerned with nPrint's performance against Nmap's *full* suite of features. As such, we downsample the labeled network device dataset to create a set of devices to Nmap. This dataset is shown in Table 3.

We further expand the types of devices we are testing to test the adaptability of nPrint across a larger range of device types. To this end, we add a new device category to the dataset: Internet of Things (IoT) devices. We gather labels for four types of IoT devices, 2 IoT cameras and 2 IoT TV streaming devices through Shodan [30]. Table 3 shows the distribution of labels from Shodan.

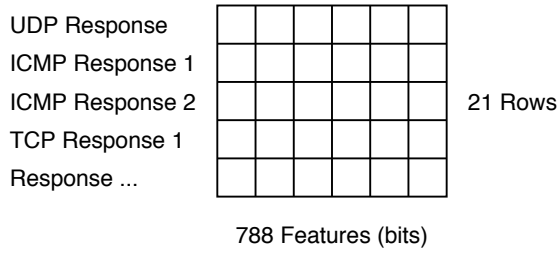


Figure 4: *Visualizing a 2D nPrint Nmap fingerprint. Rows and columns are consistent across each fingerprint.*

We modify Nmap to output the raw packet responses to each probe and Nmap each device in the dataset. Table 3 shows the distribution of devices that responded to Nmap's probes, which constitutes the final active fingerprinting dataset.

5.2 Packet Transformation and Data Representation

Nmap. Nmap transforms the responses to each probe into a fingerprint using a series of tests. We convert the fingerprints that Nmap generates into a feature vector for each device by considering each feature as a categorical feature and one-hot-encoding the values in the fingerprints. Holland *et al.* show this technique to be effective using Nmap's closed

Test Name	Summary	Nmap Weight	nPrint Importance
Explicit Congestion Notification	TCP Explicit Congestion control flag.	100	12
ICMP Response Code	ICMP Response Code.	100	10
Integrity of returned probe IP Checksum	Valid checksum in an ICMP port unreachable.	100	Inapplicable
Integrity of returned probe UDP Checksum	UDP header checksum received match.	100	Inapplicable
IP ID Sequence Generation Algorithm	Algorithm for IP ID.	100	Inapplicable
IP Total Length	Total length of packet.	100	6
Responsiveness	Target responded to a given probe.	100	Inapplicable
Returned probe IP ID value	IP ID value.	100	7
Returned Probe IP Total Length	IP Length of an ICMP port unreachable.	100	4
TCP Timestamp Option Algorithm	TCP timestamp option algorithm.	100	Inapplicable
Unused Port unreachable Field Nonzero	Last 4 bytes of ICMP port unreachable message not zero.	100	Inapplicable
Shared IP ID Sequence Boolean	Shared IP ID Sequence between TCP and ICMP.	80	Inapplicable
TCP ISN Greatest Common Divisor	Smallest TCP ISN increment.	75	Inapplicable
DonâĂŹt Fragment ICMP	IP DonâĂŹt Fragment bit for ICMP probes.	40	Inapplicable
TCP Flags	TCP flags.	30	9
TCP ISN Counter Rate	Average rate of increase for the TCP ISN.	25	Inapplicable
TCP ISN Sequence Predictability Index	Variability in the TCP ISN.	25	Inapplicable
IP DonâĂŹt Fragment Bit	IP DonâĂŹt Fragment bit.	20	8
TCP Acknowledgment Number	TCP acknowledgment number.	20	5
TCP Miscellaneous Quirks	TCP implementations, e.g, reserved field in TCP header.	20	13
TCP Options Test	TCP header options, preserving order.	20	1
TCP Reset Data Checksum	Checksum of data in TCP reset packet.	20	Inapplicable
TCP Sequence Number	TCP sequence number.	20	4
IP Initial Time-To-Live	IP initial time-to-live.	15	3
TCP Initial Window Size	TCP window size.	15	2

Table 4: *Nmap device detection rules. nPrint discovers many of these without manual feature engineering and assigns different weights to features than Nmap.*

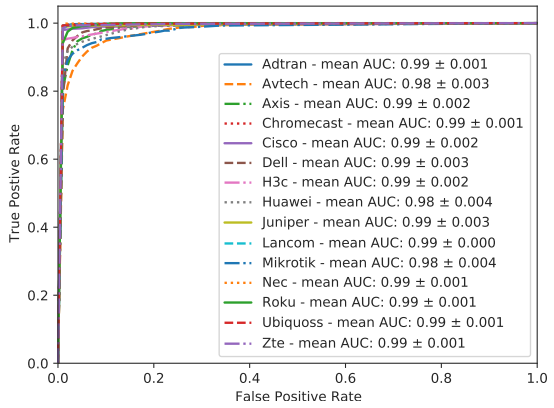
port probes, which comprise only 6 of the 16 probes Nmap sends to each device. We consider every probe Nmap sends when transforming each fingerprint into a feature vector. Nmap uses an internal heuristic, not machine learning, to determine the class of a device. We compare nPrint against Nmap's features in machine learning models to better understand the differences in performance due to *features*, instead of classification method. We make this decision as we find that some of the devices in our dataset, including 2 of the IoT devices, do not exist in Nmap's fingerprint database.

nPrint. We use the raw responses generated from the modified version of Nmap to build a nPrint for each device. Here we use the same response packets that Nmap uses to build a nPrint. Further, while Nmap computes many of its features across both sent and received packets, we only use the received packets when building a nPrint for each device.

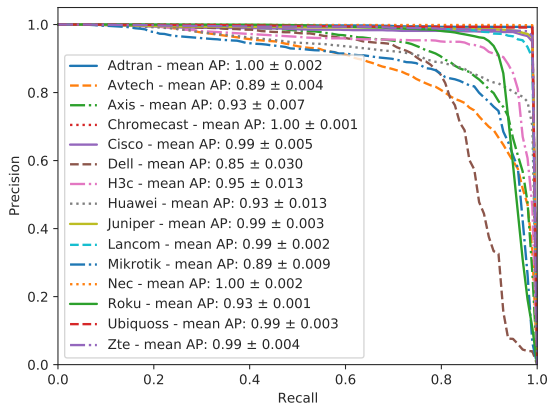
nPrint is a single-packet representation, while Nmap's classification represents a multi-packet learning problem. We must transform each Nmap response into a nPrint using the custom packet transformer and then build two-dimensional fingerprint for each device. An example of a 2D nPrint is shown in 4. We recognize that some work must be done to tailor nPrint to a specific classification problem, but point out that the amount of effort required is minimal compared to manually engineering features. For example, tailoring the single-packet nPrint packets into the 2D fingerprint shown in Figure 4 took only 16 lines of Python code.

We now examine in detail the fingerprint shown in Figure 4. First, we notice there are 21 rows in the fingerprint, while Nmap only sends 16 probes to the device. Nmap resends probes that do not garner a response up to three times. It uniquely re-names these probes. Rather than disambiguate the names, which is unreliable due to the unique naming scheme, we consider each uniquely named Nmap response as a row in the nPrint. This does not give us access to any information than Nmap does not use, and at worst duplicates data already in the nPrint. We fill any row (probe response) with 0's if the device did not respond to the probe. This 2D fingerprint can be flattened to 1 dimension, without losing representation interpretability, in 3 lines of python code for use with ensemble methods. Finally, for the active fingerprinting case it is important that ordering of the rows (responses) is consistent across each nPrint.

We compress the nPrint fingerprints by dropping each bit in the representation that has zero variance. Ultimately, we find that this compression method removes almost half of the bits (1,488 to 788) in the representation, decreasing model size and training time while maintaining the interpretability of the representation. As we remove only static bits, this compression has no effect on model performance, only training time.

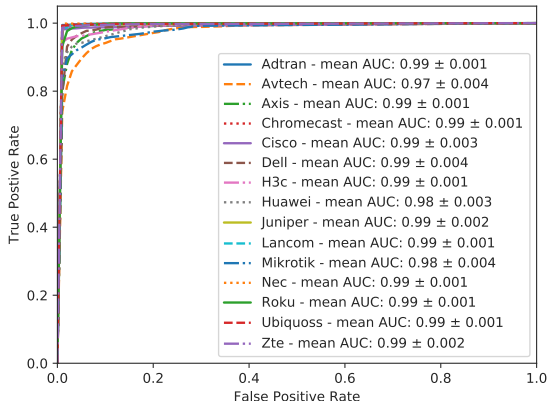


(a) *Nmap ROC*

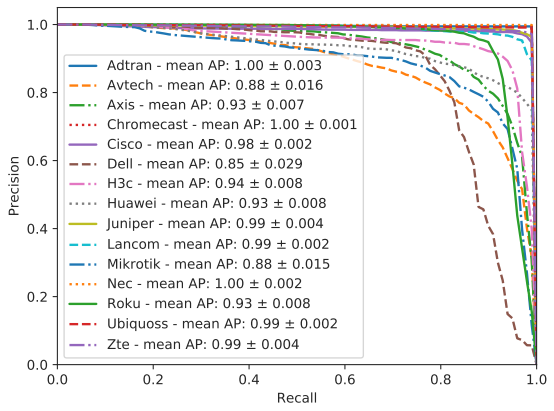


(b) *Nmap PR*

Figure 5: *Nmap's hand curated features perform well on the active fingerprinting task.*



(a) *nPrint ROC*



(b) *nPrint PR*

Figure 6: *nPrint provides equivalent performance to expert-derived features, even without access to Nmap's sent probes.*

5.3 Model Training

We train MLP, CNN, and random forest models on nPrint and MLP and random forest models on the one-hot-encoded fingerprints generated from Nmap's tests. Each metric reported represents the mean across 5-fold cross validation. Figures 5 and 6 show the ROC and PR curves of the random forest classifier for both Nmap and nPrint. Immediately, models can differentiate the devices without manual feature engineering.

Table 5 shows the performance of models trained on nPrint across different models compared to Nmap's hand engineered features. We see that, across the same models, nPrint is able to match Nmap's long-developed hand engineered features without access to the sent probes. For example, in the random forest, Nmap achieves a 92.2 F1 score and a 99.3 ROC AUC, while nPrint achieves a 92.1 F1 score and a 99.3 ROC AUC.

We have demonstrated that nPrint can be used to automatically extract information from packets. In Section 3.2, we presented multiple representation requirements and possible representations of network traffic. We now examine the pitfalls of the representations we did not choose by evaluating each representation using the active fingerprinting dataset.

Semantic representation. First, we train and evaluate a model with a variant of the semantic header representation. We transform each packet in the fingerprint into a collection of features that correspond to each header field. Each header field's value is considered as a continuous value. We fill any header field not present in a specific packet with a 0 for that feature. Ultimately this transformation led to each sample consisting of 21 rows (packets) and 38 features. Here creating a semantic representation of network traffic has problems that are eliminated with nPrint. For example, a semantic representation of this type can only consider named fields, and

Representation	Model	Accuracy	AUC ROC	F1
Nmap	RF	92.4	99.3	92.2
Nmap	MLP	90.9	98.6	90.6
Semantic	RF	91.3	99.3	91.7
Semantic	MLP	85.9	96.9	86.8
Semantic	CNN	89.0	96.6	88.8
Naive Binary	RF	90.1	99.1	90.7
Naive Binary	MLP	70.3	95.6	69.9
Naive Binary	CNN	90.9	96.2	91.2
nPrint	RF	91.7	99.3	92.1
nPrint	MLP	86.2	97.4	87.0
nPrint	CNN	90.3	98.3	90.9

Table 5: *Semantic encoding achieves high performance but requires protocol specific feature extraction. The naive binary representation does not require feature extraction but sacrifices performance. nPrint achieves high performance without the need to extract semantic features.*

is forced to ignore the payload of each packet as it cannot be reliably mapped to specifically named fields.

Table 5 shows that this semantic representation is successful for the active fingerprinting task. Although we see success in active fingerprinting, the representation still violates the representation requirements outline in Section 3. Furthermore, the semantic representation is still slightly outperformed by nPrint.

Naive binary representation. Finally, we examine the performance of the naive binary representation presented in Section 3.2. We transform each packet into its binary form and pad with 0's to the maximum packet size, resulting in a fingerprint for each device of 21 rows (packets) and 2,248 features. Table 5 shows that we achieve high performance just by naively considering each packet as a bitmap. More interestingly, this raw bitstring format generally results in a performance loss compared with nPrint. What we see is that the variable header length of network traffic mis-aligns the bits in each packet, introducing noise into the classifier. We stress that network traffic has many quirks that must be carefully considered when representing it to machine learning models. Further, blindly converting network traffic to its bitmap representation results in representations that cannot be translated back to their semantic meanings.

5.4 Feature Importance

We leverage the ability to map nPrint back to the semantic realm to examine the features that are driving the performance of the random forest model. Figure 7 shows a heatmap of the feature importances gathered from the random forest model trained on nPrint. This visualization illustrates the ability to interpret the bits that are driving the performance of the model can provide insight into what is differentiating the classes.

In this instance, we find that the TCP source port of the response probes are one of the more important features in classifying the device vendor. Upon further inspection, we find that Nmap does a port scan to find an open port to sent its open-port probes to. The IoT devices each have a specific port that is found to be open during the port scan that identify the class of devices from the routers. We also see that the TCP window size and many of the payload bits are important in classifying the device vendor. The payload bits are important in this case as Nmap sends a UDP probe to a closed port that elicits an ICMP error response back which contains the original packet that caused the error. Many devices copy differing amounts of the original packet in the ICMP error, indicating their underlying operating systems.

Finally, Table 4 directly compares the feature rankings learned by the model trained on nPrint to Nmap's internal heuristic weights. We see that nPrint places vastly different weights on many of the features that are directly comparable, with nPrint placing much higher importance on the TCP options and the TCP window size than Nmap's heuristic.

6 PASSIVE OS FINGERPRINTING

We now study the generalizability of nPrint by applying it to a different context: passive OS fingerprinting; determining the operating system of a device from network traffic that is collected passively, as opposed to sending specially crafted probes to the device. We compare the performance of a learning pipeline that utilizes nPrint to p0f, one of the most commonly used passive OS fingerprinting tools.

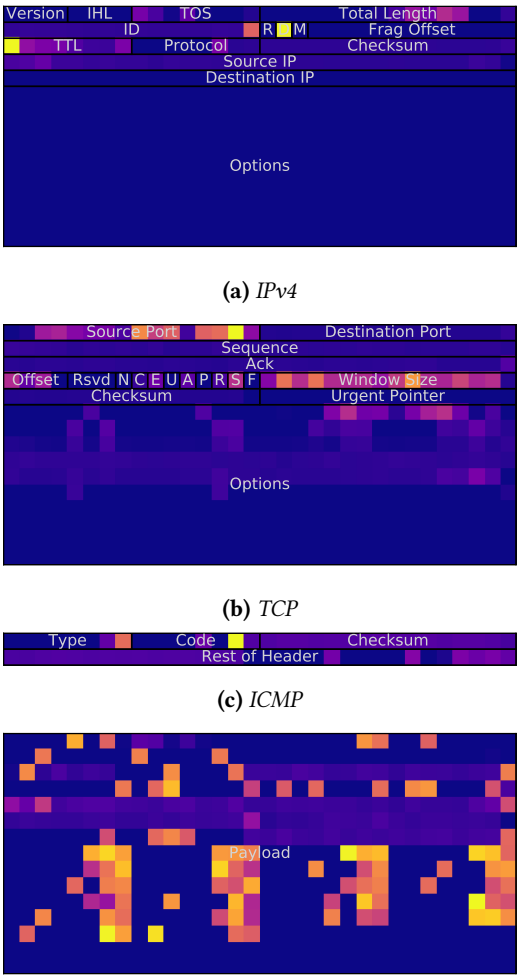
p0f utilizes an array of passive traffic fingerprinting mechanisms to identify the OS behind any TCP/IP communication. p0f relies on a user-curated database of signatures to determine the operating system of any given device. p0f generates small fingerprints of device traffic and looks for direct matches in its database for each OS.

6.1 Input (Packets)

We leverage the CICIDS2017 intrusion detection evaluation dataset, which contains PCAPs of over 50GB of network traffic over the course of 5 days [29]. The traffic contains labeled operating systems ranging from Ubuntu to Windows to MacOS. There are 17 hosts in the IDS dataset, but we find only 13 with usable traffic. The resulting devices are listed in the first column of Table 6.

6.2 Packet Transformation and Data Representation

p0f. p0f extracts a limited number of fields from each packet and looks directly in a fingerprint database to find a



(a) *IPv4*

(b) *TCP*

(c) *ICMP*

(d) *Payload*

Figure 7: *Per-bit feature importance for active fingerprinting (random forest). Brighter colors are more important. Given the nPrint representation, ML models can learn important features (e.g., IP TTL, window size), as opposed to relying on manual engineering.*

matching OS for the extracted fields. We run p0f, without modifications, to determine the operating system of the device that generated a given traffic sample. More specifically, we take the first 100,000 packets seen for each device and split them into 1,000, 100-packet traffic samples. We then run p0f on each separated 100-packet traffic sample.

nPrint. We split the traffic into 100 packet samples for each device, using the same traffic as the p0f runs. We transform each packet sample into a nPrint. Figure 8 visualizes a nPrint for passive OS detection. We drop the IP source address, the IP destination address, the TCP source and destination ports, and the TCP sequence and acknowledgement numbers from the representation to avoid direct identifiers of specific

Host	p0f			nPrint	
	Accepted Guess	Precision	Recall	Precision	Recall
Kali Linux	Kali Linux	-	-	100.0	100.0
Mac OS X	Mac OS X 10.x	100.0	0.88	100.0	0.99
Ubuntu 14.4 32B	Linux 3.11 and newer	100.0	0.69	100.0	0.99
Ubuntu 14.4 64B		100.0	0.65		
Ubuntu 16.4 32B		100.0	0.79		
Ubuntu 16.4 64B		100.0	0.68		
Ubuntu Server		100.0	0.14		
Web Server		100.0	0.14		
Windows 10	Windows 7 or 8	0.98	0.09	100.0	100.0
Windows 10 Pro		100.0	0.14		
Windows 7 Pro		100.0	0.71		
Windows 8.1		0.99	0.77		
Windows Vista		100.0	0.71		

Table 6: *Performance of nPrint vs. p0f for passive OS fingerprinting. nPrint achieves finer granularity for OS fingerprinting and near perfect precision and recall.*

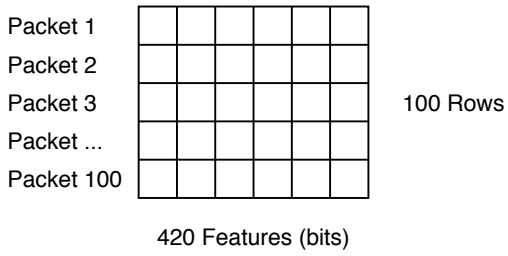
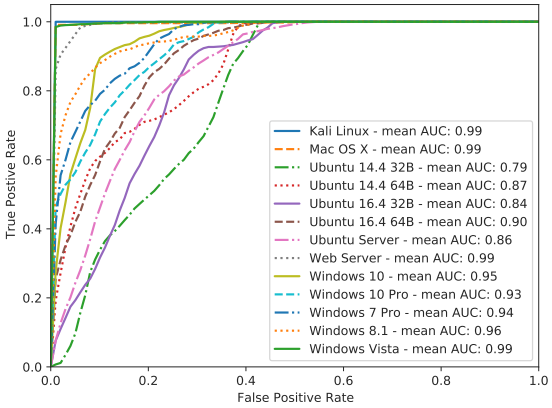


Figure 8: *Visualizing a 2D nPrint for passive OS detection. 100 packet sequences capture handshake behavior that p0f uses for fingerprinting operating systems.*

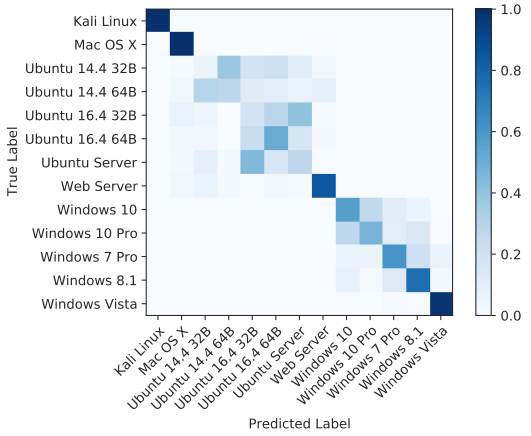
devices, rather than general operating system characteristics. Furthermore, we only use the IP and TCP headers to better compare nPrint to p0f. We compress each nPrint by dropping bits with zero variance across the entire dataset, reducing the amount of features in each nPrint from 800 to 420.

6.3 Model Training

nPrint. We train MLP, CNN, and random forest models on nPrint, finding minimal performance difference between the models. Figure 9a shows that nPrint's ROC AUC values are generally above .90 for most devices, with multiple devices being .99. We do see lower performance within the Ubuntu 14.4 and 16.4 devices, with ROC AUC values as low as .79. Figure 9b examines in more detail the performance of the model trained on nPrint. The classifier is learning to separate operating system characteristics at a high level, with the vast majority of the confusion being within the same operating system. Specifically, we see that the classifier is able to separate the devices into 6 classes: Kali Linux, Mac OS X, Ubuntu, the Web Server, Windows, and



(a) *nPrint ROC.*



(b) *nPrint Confusion Matrix.*

Figure 9: *Passive OS fingerprinting ROC and confusion matrix for nPrint. Models trained with nPrint learn to identify operating systems at a finer granularity (e.g., versions) than p0f.*

Windows Vista. Figure 9b also shows us that bit versions of the same operating systems are virtually indistinguishable, and that Windows 10 and Windows 10 Pro are difficult to distinguish. Both of these observations are not surprising, but we note there are clear limitations to operating system detection at this granularity.

p0f. To compare nPrint to p0f, we run p0f on the split files and gather all of the operating system estimates generated by the tool. p0f outputs an operating system guess only on packets that directly match a fingerprint in p0fs database, so the number of estimates varies between samples. We treat each estimate as a vote. For each sample, we tally the number of correct votes, incorrect votes, and cases where p0f offered no estimate. Using these values we calculate the precision and recall for each experiment. Table 6 shows the precision and recall for 100 packet samples.

p0f's precision is typically quite high, meaning that when it does guess the OS of a device it is correct. However, the recall values are often quite low as p0f regularly does not offer a vote for the sample, depending on host. P0f did not output a single vote for the Kali Linux machine. Interestingly, the granularity at which p0f offers operating system guesses is very low, classifying all Ubuntu devices, and the web server, as "Linux 3.11 and newer", and all of the Windows devices as "Windows 7 or 8". Interestingly, we found that nPrint was able to find distinct differences in Windows Vista from the other operating systems, but p0f makes no such distinction. To better compare nPrint directly to p0f, we use the "Accepted Guess" for p0f as a new label for each host and retrain a classifier using the 4, coarser-grain classes. Table 6 shows the results of this less granular classification. We see that nPrint has almost perfect precision and recall on the less granular problem. Here we stress the utility of nPrint because we can examine classification performance at different granularities, and even find new fingerprints for devices that may not currently be captured, such as the Windows Vista and Kali Linux differences that nPrint finds but p0f does not.

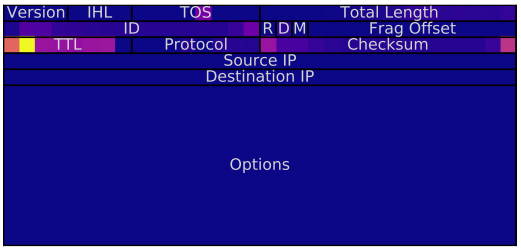
Finally, we seek to further verify that nPrint detects operating systems generally, rather than learning to identify specific devices. We set up an experiment where we compare sets of devices that share a common operating system. We take the five Ubuntu hosts and five Windows hosts in the dataset and set up a binary classification task in which we iteratively select pairs from the two lists to train a model, and test against the remaining hosts in the lists.

nPrint differentiates between Ubuntu and Windows machines with perfect balanced accuracy, ROC AUC scores, and F1 scores no matter which device pair was used for training. This is due to the different initial IP time-to-live that is set by the two operating systems which the model immediately learns. This experiment further illustrates that models can successfully identify operating systems generally from nPrint, as opposed memorizing individual device characteristics.

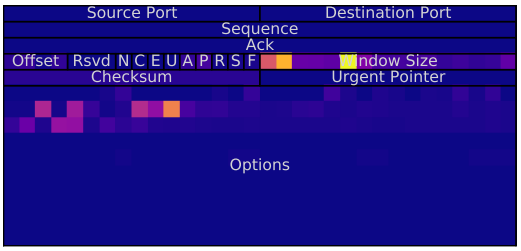
6.4 Feature Importance

Finally, we examine the feature importances for each bit in the IP and TCP headers for the passive operating system use case. Figure 10 shows the feature importance heatmaps. For the IPv4 header, the most important features are in the time-to-live (TTL) field and, to a lesser degree, the IPID field. These results confirm past observations that TTL IPID can be used for OS detection, because different operating systems use different default values for those fields [4, 21].

In the TCP header, the window size field is the most important feature. We also observe that certain bits in the TCP



(a) *IPv4*



(b) *TCP*

Figure 10: *Per-bit feature importance for passive OS detection (random forest). Brighter colors are more important. Given the nPrint representation, ML models can automatically discover important features (e.g. IP TTL, window size), as opposed to relying on manual engineering.*

options can help determine OS as some OS's include particular options by default such as maximum segment size, window scaling, or selective acknowledgement permitted. nPrint confirms much of the past literature on OS fingerprinting using certain header fields in network traffic. These results also illustrate that nPrint can achieve high accuracy without requiring hand-tailored feature engineering that previous tools are built upon.

7 APPLICATION IDENTIFICATION

Finally, we test the ability of nPrint to identify applications. More specifically, we aim to automatically identify the application and browser that generated a webRTC handshake with nPrint when provided with the handshake traffic. MacMillan *et al.* examined the feasibility of fingerprinting Snowflake, a pluggable transport for Tor that leverages WebRTC to establish browser-to-browser connections [20], which is built to be indistinguishable from other WebRTC services. They collect almost 7,000 dTLS handshakes from four different services: Facebook Messenger, Discord, Google Hangouts, and Snowflake, across two browsers: Firefox and Chrome. They then extract features from the handshakes to show that each service is uniquely identifiable. We are interested in using nPrint to automate this process entirely.

	Application Handshakes			
Browser	Snowflake	Facebook	Google	Discord
Firefox	991	796	1000	992
Chrome	0	784	995	997

Table 7: *The application identification dataset.*

Model	Accuracy	ROC AUC	F1
FFN	99.8	99.8	99.8
CNN	100.0	100.0	100.0
RF	99.4	100.0	99.4

Table 8: *Performance of models trained on nPrint. We see high performance in each model, with the CNN achieving perfect performance in all metrics considered.*

7.1 Input (Packets)

Table 7 shows the almost 7,000 dTLS handshakes collected by MacMillan *et al.*. MacMillan *et al.* examine the classification task solely at the application level. We further split the classification task into which specific browser, application pair created the handshake, increasing the number of classes in the task from four to seven.

7.2 Packet Transformation and Data Representation

We take each handshake, which was captured and filtered as a PCAP file, and transform it into a 2D nPrint, just as the previous two applications of nPrint. The number of packets in the handshakes vary from 4 to 13, we simply pad each fingerprint with rows of 0s to the maximum capture size, and allow the models trained on nPrint to identify the important features in the traffic. We do not compress this representation to test if the model is able to quickly filter out static bits in the traffic. Each nPrint consists of the IPv4, UDP, and first 10 bytes of the payload. We choose the first 10 bytes of the payload as the first few bytes of the TLS handshake messages identify which type of message is contained.

7.3 Model Training

We train FFN, CNN, and random forest models on the dataset in nPrint format. Table 8 shows the utility of nPrint on the application identification problem. The CNN is actually able to perform perfectly on the task, while the other two models perform just slightly below perfection.

We now examine the ability of nPrint to quickly ask and answer questions about constrained versions of the application identification problem. We can quickly experiment with the amount and type of traffic that must be saved to

Features	Accuracy	ROC AUC	F1
IPv4, UDP, 10 Payload Bytes	99.6	100.0	99.6
IPv4	96.6	99.9	96.9
UDP	99.5	99.9	99.6
10 Payload Bytes	77.4	95.0	78.8
25 Payload Bytes	99.7	100.0	99.7
100 Payload Bytes	99.7	100.0	99.7

Table 9: *Evaluating constrained versions of the applicaiton identifcation problem. nPrint allows researchers to better understand which headers and payload bytes are needed for specific traffic analysis problems.*

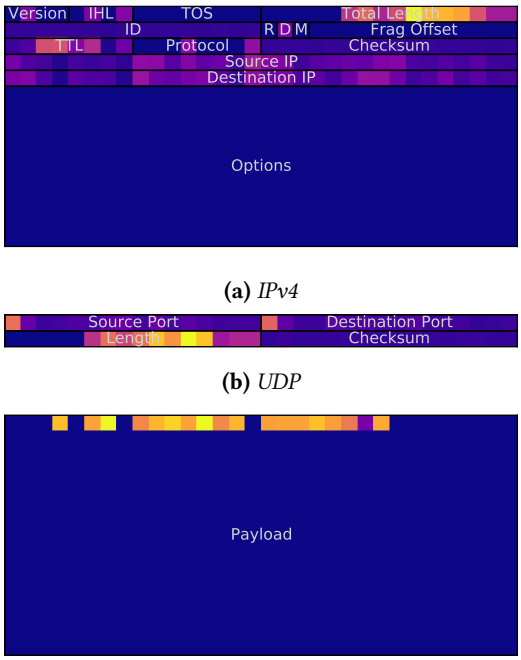
disk for a given classification problem. Table 9 shows the results of experimenting with different parts of the packet using nPrint. For this problem, the UDP header alone can be used, as the successive lengths of the packets define the handshake for each class. However, we find that the first 10 bytes of the payload alone perform poorly on the classification task, but with 25 payload bytes the classifier has much higher performance. We stress that nPrint can be used as a filtering mechanism for understanding differences in traffic. Rather than working from traffic and trying to manually extract differences to train a classifier with, we can instead train a classifier and let the model inform us of the parts of the packets important for classification. nPrint can be used in this manner to help optimize storage space in large-scale network data collection.

7.4 Feature Importance

We look to the feature importance of the random forest model to better understand the semantic features driving the performance in the model (Figure 11). We see that much of what separates the classes is in the successive lengths of the packets in the nPrint. Furthermore, the first ten bytes of the payload, combined with these lengths, can accurately predict the application and browser that generated the handshake.

8 CONCLUSION

This paper presented nPrint, a general, bitmap-based representation of network traffic that can be used for a wide variety of network traffic analysis problems. We developed design requirements for a general representation of network traffic, including the need for a complete, aligned, constant size representation that is nonetheless compact enough to be used as input for a variety of machine learning models, including deep neural networks. We evaluated nPrint on three supervised learning problems: device identification, as compared with the widely used Nmap approach; operating system identification, as compared with p0f; and application identification, based on the contents of TLS handshakes.



(a) *IPv4*

(b) *UDP*

(c) *Payload*

Figure 11: *Per-bit feature importance for (browser, application) identification (random forest). Brighter colors are more important. Given the nPrint representation, ML models can learn important features (e.g., length), as opposed to relying on manual engineering.*

We find that nPrint is at least as accurate as existing bespoke solutions to each problem across both deep learning and ensemble models. Furthermore, in contrast to existing approaches, which involve manual feature engineering specifically to each traffic analysis task, nPrint can be employed employed across a wide variety of tasks, and the general protocol-agnostic nature of the representation makes it amenable to widespread use. Finally, nPrint has the additional benefit that models can be retrained in the face of changing scenarios (e.g., software upgrades, adversarial evasion) without additional feature engineering.

We have released nPrint as an open-source software tool and highly encourage others to employ the approach on other traffic analysis problems—with a wider variety of datasets—and extend the representation as appropriate. Our evaluation shows promise for nPrint's generality, but ultimately we encourage the evaluation of nPrint on more problems. Along these lines, nPrint includes the ability to encode certain aspects of traffic such as sequencing and timing relationships (e.g., packet sequences, relative timestamps). Although we have not considered problems that leverage this information in this work, the existence of this information in nPrint make it a natural direction for future work.

9 ACKNOWLEDGEMENTS

This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-19-C-0079. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force and DARPA.

REFERENCES

[1] G. Aceto, D. Ciuonzo, A. Montieri, and A. PescapÃĺ. 2019. Mobile Encrypted Traffic Classification Using Deep Learning: Experimental Evaluation, Lessons Learned, and Challenges. *IEEE Transactions on Network and Service Management* 16, 2 (2019), 445–458.

[2] Mashael AlSabah, Kevin Bauer, and Ian Goldberg. 2012. Enhancing TorâĂŹs Performance Using Real-Time Traffic Classification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS âĂŹ12)*. Association for Computing Machinery, New York, NY, USA, 73âĂŞ84. https://doi.org/10.1145/2382196.2382208

[3] J. Barker, P. Hannay, and P. Szewczyk. 2011. Using Traffic Analysis to Identify the Second Generation Onion Router. In *2011 IFIP 9th International Conference on Embedded and Ubiquitous Computing*. 72–78. https://doi.org/10.1109/EUC.2011.76

[4] Steven M. Bellovin. 2002. A Technique for Counting Natted Hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment* (Marseille, France) *(IMW âĂŹ02)*. Association for Computing Machinery, New York, NY, USA, 267âĂŞ272. https://doi.org/10.1145/637201.637243

[5] Robert Beverly. 2004. A Robust Classifier for Passive TCP/IP Fingerprinting. In *Proceedings of the 5th Passive and Active Measurement (PAM) Workshop*.

[6] Vladimir Brik, Suman Banerjee, Marco Gruteser, and Sangho Oh. 2008. Wireless Device Identification with Radiometric Signatures. In *Proceedings of the 14th ACM international conference on Mobile computing and networking*. 116–127.

[7] Douglas E Comer and John C Lin. 1994. Probing TCP implementations. In *Usenix Summer*. 245–255.

[8] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. 2013. ZMap: Fast Internet-wide scanning and its security applications. In *Proceedings of the 22nd USENIX Security Symposium*. 605–620.

[9] Peter Eckersley. 2010. How Unique is Your Web Browser?. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 1–18.

[10] David Formby, Preethi Srinivasan, Andrew M. Leonard, Jonathan D. Rogers, and Raheem A. Beyah. 2016. Who's in Control of Your Control System? Device Fingerprinting for Cyber-Physical Systems. In *23rd Annual Network and Distributed System Security Symposium, NDSS, 2016, San Diego, California, USA, February 21-24, 2016*. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/who-control-your-control-system-device-fingerprinting-cyber-physical-systems.pdf

[11] Jérôme François, Humberto Abdelnur, Radu State, and Olivier Festor. 2010. Machine Learning Techniques for Passive Network Inventory. *IEEE Transactions on Network and Service Management* 7, 4 (2010), 244–257.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[13] Benjamin Greschbach, Tobias Pulls, Laura M. Roberts, Phillip Winter, , and Nick Feamster. 2017. The Effect of DNS on TorâĂŹs Anonymity. In *Network and Distributed System Security Symposium*.

[14] Jamie Hayes and George Danezis. 2016. k-fingerprinting: A Robust Scalable Website Fingerprinting Technique. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 1187–1203. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/hayes

[15] Jordan Holland, Ross Teixeria, Paul Schmitt, Kevin Borgolte, Jennifer Rexford, Nick Feamster, and Jonathan Mayer. 2020. Classifying Network Vendors at Internet scale. *arXiv preprint arXiv:2006.13086* (2020).

[16] Ren-Hung Hwang, Min-Chun Peng, Van-Linh Nguyen, and Yu-Lun Chang. 2019. An LSTM-Based Deep Learning Approach for Classifying Malicious Traffic at the Packet Level. *Applied Sciences* 9, 16 (2019), 3414.

[17] Tadayoshi Kohno, Andre Broido, and Kimberly C Claffy. 2005. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing* 2, 2 (2005), 93–108.

[18] Richard Lippmann, David Fried, Keith Piwowarski, and William Streilein. 2003. Passive operating system identification from TCP/IP packet headers. In *Workshop on Data Mining for Computer Security*, Vol. 40.

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

[20] Kyle MacMillan. [n.d.]. *Evaluating Snowflake as an Indistinguishable Censorship Circumvention Tool*. https://github.com/kyle-macmillan/snowflake_fingerprintability Dataset.

[21] Tony Miller. 2020. Passive OS Fingerprinting: Details and Techniques. http://www.ouah.org/incosfingerp.htm.

[22] Se Eun Oh, Saikrishna Sunkam, and Nicholas Hopper. 2019. p1-FP: Extraction, Classification, and Prediction of Website Fingerprints with Deep Learning. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (2019), 191–209.

[23] p0f 2016. p0f v3 (version 3.09b). http://lcamtuf.coredump.cx/p0f3.

[24] Jitendra Padhye and Sally Floyd. 2001. On Inferring TCP Behavior. *SIGCOMM Comput. Commun. Rev.* 31, 4 (Aug. 2001), 287âĂŞ298. https://doi.org/10.1145/964723.383083

[25] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. 2016. Website Fingerprinting at Internet Scale.. In *NDSS*.

[26] Vern Paxson. 1997. Automated Packet Trace Analysis of TCP Implementations. *SIGCOMM Comput. Commun. Rev.* 27, 4 (Oct. 1997), 167âĂŞ179. https://doi.org/10.1145/263109.263160

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[28] Vera Rimmer, Davy Preuveneers, Marc Juárez, Tom van Goethem, and Wouter Joosen. 2018. Automated Website Fingerprinting through Deep Learning. In *Network and Distributed System Security Symposium, NDSS*. San Diego, CA, USA.

[29] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. 2018. Toward generating a new intrusion detection dataset and intrusion traffic characterization.. In *ICISSP*.

[30] Shodan. 2020. Shodan. https://www.shodan.io/.

[31] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. 2018. Deep Fingerprinting: Undermining Website Fingerprinting Defenses with Deep Learning. *arXiv preprint arXiv:1801.02265* (2018). arXiv:1801.02265 [cs.CR]

[32] Robin Sommer and Vern Paxson. 2010. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE symposium on security and privacy*. IEEE, 305–316.

[33] Vijayanand Thangavelu, Dinil Mon Divakaran, Rishi Sairam, Suman Sankar Bhunia, and Mohan Gurusamy. 2018. Deft: A Distributed IoT Fingerprinting Technique. *IEEE Internet of Things Journal* 6, 1 (2018), 940–952.

[34] Rahmadi Trimananda, Janus Varmarken, Athina Markopoulou, and Brian Demsky. 2020. Packet-Level Signatures for Smart Home Devices. In *Network and Distributed System Security Symposium, NDSS*. San Diego, CA, USA.

[35] Shobha Venkataraman, Juan Caballero, Pongsin Poosankam, Min Kang, and Dawn Song. 2007. Fig: Automatic Fingerprint Generation.. In *Network and Distributed System Security Symposium, NDSS*.

[36] Liang Wang, Kevin P. Dyer, Aditya Akella, Thomas Ristenpart, and Thomas Shrimpton. 2015. Seeing through Network-Protocol Obfuscation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS âĂŹ15)*. Association for Computing Machinery, New York, NY, USA, 57âĂŞ69. https://doi.org/10.1145/2810103.2813715

[37] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. 2014. Effective Attacks and Provable Defenses for Website Fingerprinting. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 143–157. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/wang_tao

[38] Tao Wang and Ian Goldberg. 2013. Improved website fingerprinting on tor. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*. 201–212.

[39] Tao Wang and Ian Goldberg. 2017. Walkie-Talkie: An Efficient Defense against Passive Website Fingerprinting Attacks. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SECâĂŹ17)*. USENIX Association, USA, 1375âĂŞ1390.

[40] Wei Wang, Yiqiang Sheng, Jinlin Wang, Xuewen Zeng, Xiaozhou Ye, Yongzhong Huang, and Ming Zhu. 2017. HAST-IDS: Learning hierarchical spatial-temporal features using deep neural networks to improve intrusion detection. *IEEE Access* 6 (2017), 1792–1806.

[41] W. Wang, M. Zhu, J. Wang, X. Zeng, and Z. Yang. 2017. End-to-end encrypted traffic classification with one-dimensional convolution neural networks. In *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*. Beijing, China.

[42] Nigel Williams, Sebastian Zander, and Grenville Armitage. 2006. A Preliminary Performance Comparison of Five Machine Learning Algorithms for Practical IP Traffic Flow Classification. *SIGCOMM Comput. Commun. Rev.* 36, 5 (Oct. 2006), 5âĂŞ16. https://doi.org/10.1145/1163593.1163596

[43] Yang Yu, Jun Long, and Zhiping Cai. 2017. Network intrusion detection through stacking dilated convolutional autoencoders. *Security and Communication Networks* 2017 (2017).