nPrint: Standard Packet-level Network Traffic Analysis

Jordan Holland
Princeton University

Paul Schmitt
Princeton University

Nick Feamster
University of Chicago

Prateek Mittal
Princeton University

https://nprint.github.io/nprint

Abstract

This paper presents nPrint, a standard, packet-based representation of network traffic. nPrint enables machine learning on network traffic without manual feature engineering. We combine nPrint with automated machine learning (AutoML) to demonstrate that nPrint can generate a standard traffic representation across a variety of machine learning tasks and models. We present the design and implementation of nPrint, describe how we integrate it with AutoML, and apply the pipeline to three common network traffic classification problems: operating system detection, device fingerprinting, and application identification.

Our evaluation shows that models trained on nPrint achieve higher performance than the state-of-the-art tools for these tasks, without relying on manually engineered rules or features. nPrint's contribution is thus in its generality, as it lowers the barrier to applying machine learning techniques for a variety of network traffic analysis problems. We have implemented and released nPrint as open-source software. Our performance evaluation demonstrates that nPrint can be deployed many settings, from offline transformation of standard packet capture formats to online streaming deployments.

1 Introduction

Identifying or "fingerprinting" network devices, operating systems, and applications based on network traffic is a common task in communications networks. Network operators often need to identify applications and devices for many reasons, including identifying rogue devices, determining vulnerable devices on the network [6,15,43], and detecting anomalous or malicious application traffic. On the other hand, fingerprinting and application identification constitutes a privacy threat; thus, understanding the capabilities and limitations of current fingerprinting techniques can help shed more light on privacy risks [10,16,34].

Fingerprinting typically relies on a combination of active or passive collection of network traffic and usually involves application of fixed rules to the captured traffic. One common tool for fingerprinting operating systems, p0f, identifies operating systems by matching operating system-specific defaults such as TCP options to static rules that identify the operating system [32]. Nmap 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 [26]. More broadly, traffic analysis aims to identify application traffic, devices, web browsing behavior, and even human behavior from network traffic patterns.

Although traffic fingerprinting involves a broad set of problems, many fingerprinting tasks involve devising features and rules that are bespoke for that particular task [7,10,16,22,34,44,49]. Crafting these rules is often 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 many supervised and unsupervised machine learning algorithms creates new opportunities for automatically learning features that can perform fingerprinting. Yet, the success of any machine learning approach ultimately depends on presenting the models with appropriate *data 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 create a data representation that is appropriate for the classification task and ultimately training a model on that specific data representation. In many cases, feature engineering requires a significant amount of effort. Even with expert domain knowledge feature exploration and engineering is painstaking and manual. Such manual analysis may focus on unimportant features or omit 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 takes a different approach, eschewing manual feature engineering and selection entirely and instead exploring *whether (and how) a single, standard packet representation could apply to a broad array of traffic classification tasks*. We design a standard packet representation, *nPrint* that encodes each packet in an inherently normalized, binary representation. The design of this representation enables machine learning models to automatically discover important parts of packets for each distinct classification task.

A primary contribution of this paper is an in-depth evaluation of whether a standard encoding of network traffic can yield as accurate performance for a variety of fingerprinting tasks as bespoke representations and models that are tailored to each task. To this end, we apply nPrint to three network traffic fingerprinting problems—operating systems, devices, and applications. Rather than choosing a set of models to train and evaluate ourselves, we leverage new developments in automated machine learning (AutoML) to optimize feature selection, model selection, and hyperparameter optimization for each problem. Combining nPrint and public AutoML tools allows researchers and practitioners to use network traffic as input to a broad array of networking tasks, without the painstaking processes of manual feature engineering or model selection and tuning. Our evaluation of this process shows that models trained on nPrint can perform finger-grained OS detection than p0f, achieve higher accuracy than Nmap in device fingerprinting, and automatically identify applications in noisy traffic.

A significant aspect of this work has involved making the process of performing classification using nPrint as seamless as possible. We have released nPrint as open-source software, and the project is already garnering a community of users. nPrint is lightweight, requiring less than 1 MB of RAM, and it can capture traffic at rates of 10 Gbps, allowing it to operate in a variety of environments. We have tested the tool on offline packet traces, as well as in streaming, online settings. This work thus opens promising new directions for applications to an even broader set of traffic analysis and classification tasks.

2 Related Work

This section explores past work on manual and automated fingerprinting techniques and how they relate to nPrint.

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 [8]. Padhye and Floyd identified differences between TCP implementations to detect bugs in public web servers [33]. Paxson passively identified TCP implementations using traffic traces [35].

Past work has developed techniques to fingerprint host operating systems and devices. There are multiple tools and methods for host OS fingerprinting, using both active and passive techniques. Passive OS identification aims to identify operating systems from passively captured network traffic [6, 25]. P0f passively observes traffic and determines the operating system largely based on TCP behavior [32]. Another common tool is Nmap [26], which performs active fingerprinting. Nmap sends probes and examines the responses received from 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. In contrast, nPrint does not focus on heuristics and *a priori* knowledge of implementation differences between host networking stacks. Instead, nPrint relies on the model to learn these 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 used to identify individual physical devices by Kohno *et al.* [22]. Formby *et al.* passively fingerprint industrial control system devices [14].

Machine learning-based fingerprinting. Machine learning techniques have been applied to network traffic classification and fingerprinting [3, 4, 46, 53]. Wang *et al.* developed an ML-based classification model to detect obfuscated traffic [47]. 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 [42]. Trimananda *et al.* used DBSCAN to identify smart home device actions in network traffic [45].

Other work has used machine learning to identify websites visited through the Tor network [17, 34, 48, 50]. Deep 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 [31, 36, 40]. These works differ from this work, due to their focus on the Tor setting. In Tor, all packets are the same size and encrypted, 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.

Deep learning techniques have become popular for network traffic classification problems [1, 20, 52, 54]. Yu *et al.* used convolutional autoencoders for network intrusion detection [54]. Wang *et al.* applied 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 [51]. 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.

Automated machine learning. While the use of deep learning techniques can help automate feature engineering, a separate line of research has examined how to perform automated machine learning (AutoML). The work examines the use of optimization techniques to automate not only feature engineering and selection, but model archietecture and hyperparameter optimization as well [12, 13, 21, 23, 24]. These tools have recently been used for model compression, image classification, and even bank failure prediction [2, 18]. To our knowledge, we are the first to explore the combination of AutoML and network traffic classification.

We specifically use AutoGluon-Tabular, which performs feature selection, model selection, and hyperparameter optimization by searching through a set of base models [11].

These models include deep neural networks, tree based methods such as random forests, non-parametric methods such as K-nearest neighbors, and gradient boosted tree methods. Beyond searching the singular models, AutoGluon-Tabular creates weighted ensemble models out of the base models to achieve higher performance than other AutoML tools in less overall training time [11].

3 Data Representation

For many classification problems, the choice of how to represent the data is at least as important as the choice of model. Many machine learning models are well-tuned for standard benchmarks (e.g., images, video, audio), but unfortunately network traffic does not naturally lend itself to these types of representations. Nonetheless, the nature of the models dictate certain design requirements, which we outline in Section 3.1. Section 3.2.2 explores three possible standard representations of network traffic, including a semantic encoding, an unaligned binary representation, and a hybrid approach, a binary representation of each packet that also encodes the semantic structure of the bits.

3.1 Design Requirements

Complete. Rather than select for certain features (or representations), we aim to device a representation that includes every bit of a packet header. Doing so avoids the problems of relying on domain knowledge that one packet header (or combination thereof) is more important than others. Our intuition is that the models can determine which features are important for a given problem without human guidance given a complete, standard representation.

Constant size per problem. A majority of machine learning models assume that inputs are always the same size. For example, a trained neural network expects a certain number of inputs; other models such as random forests and even regression models expect that the input conforms to a particular size (i.e., number of elements). Thus, each representation must be constant size—even if the sizes of individual packets or packet headers vary. Common approaches used in image recognition, such as scaling, do not directly apply to network traffic. An ideal representation has a size per-problem that is determined *a priori*; this *a priori* knowledge is desirable to avoid multiple passes over a stored packet trace, and it is essential in data streaming contexts, where multiple passes are not possible.

Inherently normalized. Machine learning models typically perform better when features are normalized: even simple linear models (e.g., linear and logistic regression) operate on normalized features; in more complex models that use gradient-based optimization, normalization both decreases training time and increases model stability [29,39]. Of course, it is possible to normalize *any* feature set, but it is more convenient if the initial data representation is inherently normalized. Such a representation makes it possible to avoid making a full pass over the data to compute minimum and maximum values, particularly given that many network traffic datasets are large, and the models themselves may also be used in a streaming context.

Aligned. Every location in the representation should correspond to the same part of the packet header across all packets. Alignment allows for models to learn feature representations based on the fact that specific features (i.e., packet headers) are always located at the same offset in a packet. This requirement is needed when considering packets as bitmaps, as both protocols and packets differ in length.

3.2 Possible Standard Data Representations

Network traffic can be represented in multiple ways. Here we discuss three possible representation options: semantic, unaligned binary, and a hybrid representation we use in nPrint.

3.2.1 Semantic Representation

A classic view of network traffic examines packets as a collection of higher level headers, such as IP, TCP, and UDP. Each header is broken into respective semantic fields such as the IP TTL, the TCP port numbers, and the UDP length fields. A standard semantic representation of network traffic collects all of these semantic fields in a single representation. This semantic representation is complete and constant size but has drawbacks. Figure 1 shows an example of this semantic representation and some of its drawbacks as a standard representation. First, the representation does not preserve ordering of options fields, which have been long used to separate classes of devices in fingerprinting [26,41].

We examine two of the datasets presented in Sections 5.3.1 and 5.3.2 and find 10 and 59 unique TCP option orderings respectively. We further explore the effect option ordering can have on classification performance in Appendix A.1. Second, domain expertise is required to parse the semantic structure of each protocol, and even with this knowledge, engineering the representation of each feature is often a significant exercise. For example, domain knowledge might indicate that the TCP source port is an important field, but further (often manual) evaluation may be needed to determine whether it should be represented as a continuous value, or with a one-hot encoding. Semantic features also may need to be normalized depending on the model.

3.2.2 Unaligned Binary Representation

We can preserve ordering and mitigate reliance on manual feature engineering with a raw bitmap representation. This choice leads to a consistent, pre-normalized representation

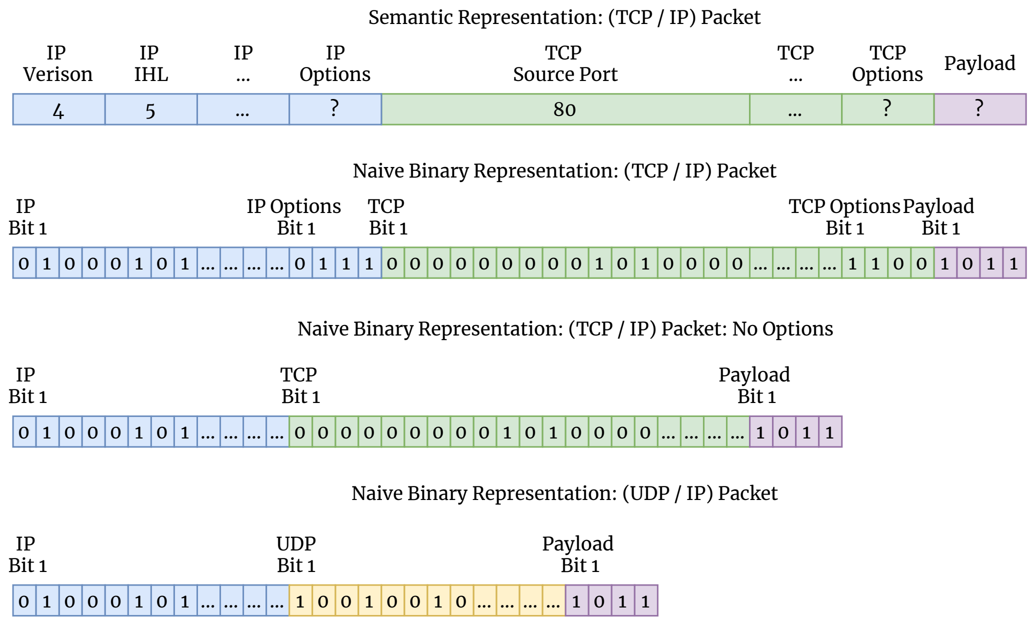


Figure 1: *Semantic and binary nPrint representations introduce various problems that make modeling difficult: Semantic loses ordering of specific fields such as the TCP options. Binary nPrints lack alignment, which can degrade performance by injecting noise at the misaligned features.*

that creates a 1xM "image" of each packet. We see an example of this representation (termed binary nPrint) in Figure 1. However, transforming each packet into its bitmap representation ignores many of the intricate details that must be considered when modeling network traffic: varying sizes and protocols. These issues can cause feature vectors for two packets to have entirely different meanings for the same feature. For example, a TCP packet and a UDP packet with the same IP header would have entirely different information represented as the same feature. Figure 1 illustrates this problem in detail.

Worse, this problem can occur within two packets of the same protocol. For example, a TCP/IP packet with IP options and a TCP/IP packet without IP options will cause the bits to be misaligned in the two representations. Misalignment can manifest itself in two ways: 1) decreasing model performance as the misaligned bits introduce noise in the model where important features may exist; and 2) the resulting representation is not interpretable, as we cannot correctly map each bit in the representation back to a semantic meaning. The ability to understand the features that are driving the performance of a given model is especially important in network traffic where we have an extensive understanding of the semantics of the underlying data, in contrast to image classification problems.

3.2.3 Hybrid: nPrint

Figure 2 exhibits nPrint, a single-packet representation that can be directly provided as input to machine learning models. This representation allows models to learn important characteristics of the traffic, rather than manually encoding those features directly. nPrint is a hybrid of semantic and binary packet representations, representing packets to models as raw binary data, but aligning the binary data in such a way that recognizes that the packets themselves have specific semantic structure. By using internal padding, nPrint mitigates misalignment that can occur with an unaligned binary representation while still preserving ordering of options. Further, nPrint encodes the semantic structure of protocols while only requiring knowledge of the maximum length of the protocol.

nPrint is complete, aligned, constant size per problem, and inherently normalized. nPrint is complete: any packet can be represented without information loss. It is aligned: 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. Not all models are interpretable, but by having an interpretable *representation*, we can better understand models that are. nPrint is also inherently 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 constant size per-problem: each packet is represented in the same number of features. We make the payload an optional number of bytes for a given problem: with the increasing majority of network traffic being

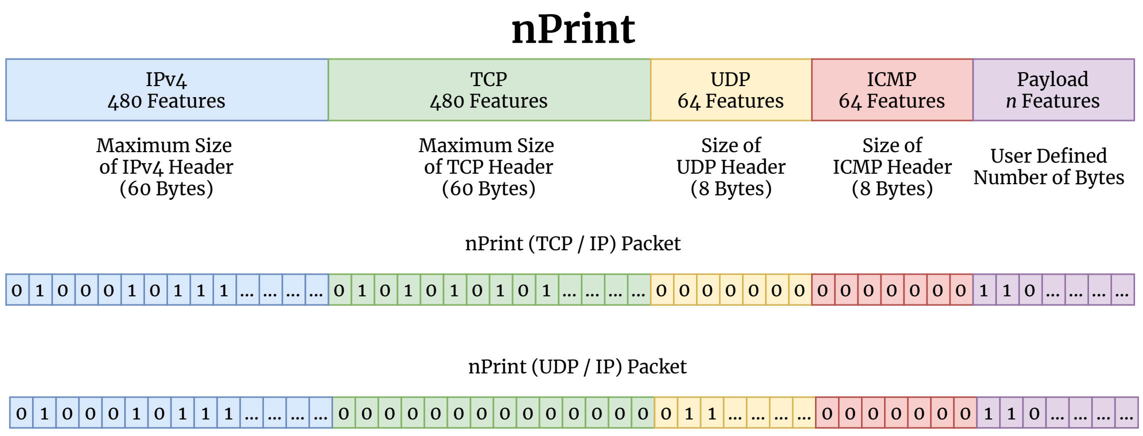

nPrint
IPv4
480 Features
TCP
480 Features
UDP
64 Features
ICMP
64 Features
Payload
n Features
Maximum Size of IPv4 Header (60 Bytes)
Maximum Size of TCP Header (60 Bytes)
Size of UDP Header (8 Bytes)
Size of ICMP Header (8 Bytes)
User Defined Number of Bytes
nPrint (TCP / IP) Packet
nPrint (UDP / IP) Packet

Figure 2: *nPrint, the complete, normalized, 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. Our approach removes reliance on expensive human-driven feature engineering while avoiding misaligned features.*

Problem	Unique LOC (Python)	Input (Packets)	nPrint (Seconds)
Passive OS Detection	27	1,343,920	49
Active Device Fingerprinting	16	274,010	13
Application Identification	33	48,816	12

Table 1: *nPrint requires minimal tailoring for different tasks and transforms packets quickly.*

encrypted, the payload is not usable for many traffic classification problems.

nPrint is modular and extensible. First, other protocols (e.g., ICMP) can be added to the representation. Second, many classification problems require sets of packets. nPrint can be used in such classification problems; we extended nPrint to do so in as little as 16 lines of Python code. If we consider each single nPrint as a 1xM matrix, where M is the number of features in the fingerprint, we can concatenate nPrints when classification problems depend on multi-packet input.

4 nPrint Implementation

We implemented and released nPrint in C++ and evaluated it in different contexts, including its memory footprint and at line-rate. nPrint currently supports Ethernet, IPv4, fixed IPv6 headers, UDP, TCP, ICMP, and any corresponding packet payloads. nPrint can either process offline packet packet capture formats such as PCAP and Zmap [9] output or capture packets directly from a live interface. nPrint can also reverse the encoding, creating a PCAP from the nPrint format.

We evaluate the performance of nPrint on a system with a 4-core, 2.7 GHz CPU (Intel Core i7-8559U) and 32GB of RAM. Table 1 shows that nPrint runs in under one minute for all of the datasets we consider in this paper.

Next, we examine the memory footprint of nPrint and the ability of nPrint to transform packets at line-rate on live traffic from a 10 GbE link.

Memory footprint. nPrint has a constant memory footprint that depends only on the output configuration. As an example, we profile memory usage with valgrind while configuring nPrint to include IPv4, UDP, ICMP, TCP, and 20 payload bytes. This configuration results in a constant memory footprint of about 310 KB. Running with a IPv4 and TCP output configuration (used for P0f evaluation) yields a constant memory footprint of about 295 KB.

Line-rate processing. nPrint processes each packet independently, making it amenable to parallelization. We test the performance of nPrint on a 10 GbE university Internet link using a commodity server. The server has 36 Intel Xeon 6154 CPUs running at 3 GHz and 376 GB of RAM. We observe that by load balancing the traffic across multiple receive queue/CPU pairs using Receive-Side Scaling (RSS[1]) we can run multiple, parallel nPrint processes without incurring penalties for moving data between cores. We verified that nPrint is able to run on live traffic at 10 GbE with roughly zero loss using 16 queues and 16 nPrint processes. Given this performance, nPrint should be capable of processing higher rates by leveraging further parallelization and optimized packet libraries, such as zero-copy pf_ring [30].

Table 1 also shows the lines of code to tailor nPrint to various fingerprinting tasks, which entails assigning nPrint output to labels. Tailoring nPrint for application identification required 33 additional lines of Python to aggregate the over 6,500 PCAPs and associate each with a label.

[1]RSS is Intel-specific, depending on hardware another similar multi-queue receive technology could also be used.

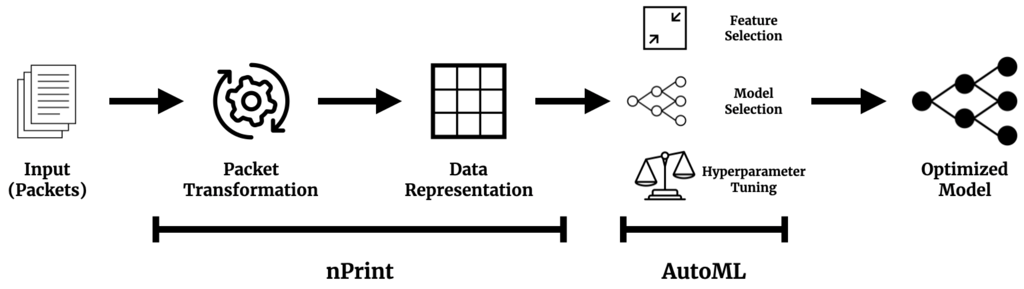


Figure 3: *nPrint produces a standard network traffic representation that can be combined with AutoML tools to create a standard, largely automated, traffic analysis pipeline.*

5 nPrint in Practice

In this section, we first explain how nPrint can be included in a machine learning pipeline with AutoML and describe the metrics used to evaluate the optimized models. We then evaluate nPrint's effectiveness in practice by integrating it into ML pipelines for three distinct applications: passive OS detection, active device fingerprinting, and application identification.

5.1 Including nPrint in ML Pipelines

Many past works have used a pipeline similar to the one seen in Figure 3. Packets are transformed into features developed by experts over the course of days, weeks, or years, and ultimately trained on one, or a small set of models that experts believe will work best on the developed features. Finally, the models are tuned either by hand or through a structured search process. We highlight an opportunity to simplify the standard pipeline in Figure 3 through nPrint and new AutoML systems.

First, we designed nPrint to be directly used in a machine learning pipeline, standardizing the tedious feature development process for a large set of packet-level traffic analysis problems. Next, we notice the opportunity for nPrint to be directly combined with AutoML tools to abstract the second half of the standard pipeline.

AutoML tools are designed to automate feature-selection, model selection, and hyperparameter tuning to find an optimized model for a given set of features and labels. Rather than running hyperparameter optimization on one model, AutoML tools use optimization techniques to perform combined algorithm selection and hyperparameter optimization, searching for the highest performing model over a larger set of possibilities than is possible by hand.

We follow the same intuition that models can extract the best features for each task and allow AutoML to determine the best type of model and hyperparameters for that problem. This decision provides multiple benefits: 1) we can train and test a wider variety of model types, 2) we can optimize the hyperparameters for *every* model we train, and 3) we are ensured that the best model is chosen for a given representation.

5.2 AugoGluon-Tabular Automl

We use AutoGluon-Tabular to perform feature selection, model search, and hyperparameter optimization for all three problems we evaluate [11]. We choose AutoGluon as it has been shown to outperform many other public AutoML tools given the same data. While many AutoML tools search a set of models and corresponding hyperparameters, AutoGluon achieves higher performance by ensembling multiple single models that perform well. AutoGluon-Tabular allows us to train, optimize, and test over 50 models for each problem stemming from 5 different base model classes:

- Deep Neural Networks
- K-neighbors
- Random Forest
- Extra Trees
- LightGBM
- CatBoost

The highest performing model for each problems we examine is an ensemble of two of the base model classes above.

AutoGluon has a `presets` parameter that determines the speed of the training and size of the model versus the overall predictive quality of the models trained. We set the `presets` parameter to `high_quality_fast_inference_only_refit`, which produces models with high predictive accuracy and fast inference. There is a quality preset of "best quality" which can create models with slightly higher predictive accuracy, but at the cost of ~10x-200x slower inference and ~10x-200x higher disk usage. We make this decision as we believe inference time is an important metric when considering network traffic analysis. Most importantly, we use the same preset for every problem. By selecting a high quality model setting, each model is bagged with 10 folds, decreasing the bias of individual models.

		p0f						nPrint					
		1 Packet		10 Packets		100 Packets		1 Packet		10 Packets		100 Packets	
Host	p0f Label	P	R	P	R	P	R	P	R	P	R	P	R
Mac OS	Mac OS x 10.x	1.00	0.05	1.00	0.28	1.00	0.88	0.99	0.99	1.00	0.99	1.00	0.99
Web Server	Linux 3.11 and newer	1.00	0.01	1.00	0.25	1.00	0.74	0.99	0.99	0.99	1.00	0.99	1.00
Ubuntu 14.4 32B		1.00	0.04	1.00	0.20	1.00	0.69						
Ubuntu 14.4 64B		1.00	0.04	1.00	0.20	1.00	0.65						
Ubuntu 16.4 32B		1.00	0.05	1.00	0.19	1.00	0.68						
Ubuntu 16.4 64B		1.00	0.04	1.00	0.24	1.00	0.79						
Ubuntu Server		1.00	0.05	1.00	0.25	1.00	0.74						
Windows 10	Windows 7 or 8	0.99	0.00	0.98	0.02	0.98	0.09	1.00	1.00	1.00	1.00	1.00	1.00
Windows 10 Pro		0.99	0.01	0.98	0.04	1.00	0.14						
Windows 7 Pro		1.00	0.04	1.00	0.23	1.00	0.71						
Windows 8.1		0.99	0.05	0.99	0.25	0.99	0.77						
Windows Vista		1.00	0.01	1.00	0.27	1.00	0.71						
Kali Linux	No output	-	-	-	-	-	-	-	-	-	-	-	-

Table 2: *Performance of nPrint vs. p0f for passive OS fingerprinting. nPrint is capable of finer granularity for OS fingerprinting and achieves near perfect precision and recall when compared directly to p0f. While p0f never provides any OS estimate for Kali Linux, we include it for testing fine-grained OS classification with nPrint.*

We set no limit on model training time, allowing AutoGluon to find the best model, and split every dataset into 75% training and 25% testing datasets. Finally, we set the evaluation metric to `f1_macro`, which represents a F1 score that is calculated by calculating the F1 Score for each class in a multi-class classification problem and calculating their unweighted mean. This decision leads AutoGluon to tune hyperparameters and ensemble weights to optimize the F1-macro score on validation data.

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 balanced 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 present macro AUC 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. The ROC AUC is computed by calculating the ROC AUC for each class and taking their unweighted mean. F1 scores represent a weighted average of precision and recall. In multi-class classification task we report a macro F1 score that is calculated in the same manner as optimized during training.

5.3 Example Applications of nPrint

In this section, we highlight the versatility and performance of nPrint by applying it to three distinct inference tasks. nPrint can match or beat the performance of existing bespoke approaches for each of these problems.

5.3.1 Passive OS Fingerprinting

We study nPrint in the context of passive OS fingerprinting: determining the operating system of a device from network traffic. We compare the performance of a learning pipeline that uses nPrint and AutoML to p0f, one of the most well-known and commonly used passive OS fingerprinting tools. Ultimately, we find that nPrint can perform OS detection with much higher recall rate than p0f, using much smaller packet sequences. Further, we find that nPrint can uncover finer-grained differences in OSes compared to p0f.

p0f uses an array of 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 fingerprints from device traffic and looks for direct matches in its database in order to identify the OS.

Input (Packets). We use the CICIDS2017 intrusion detection evaluation dataset, which contains PCAPs of over 50 GB of network traffic over the course of 5 days [37]. 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 usable devices are seen in the first column of Table 2.

Packet Transformation and Data Representation. We vary the amount of traffic available to p0f and nPrint to compare performance in different settings. We use the first 100,000 packets seen for each device and split them into sets of 1, 10, and 100 packet samples (*i.e.*, 100,000 1-packet PCAPs, 10,000 10 packet PCAPS, etc.) to be used with both. This results in three separate classification problems, each problem using the same traffic, but varying amounts of information the classification technique accesses in each sample.

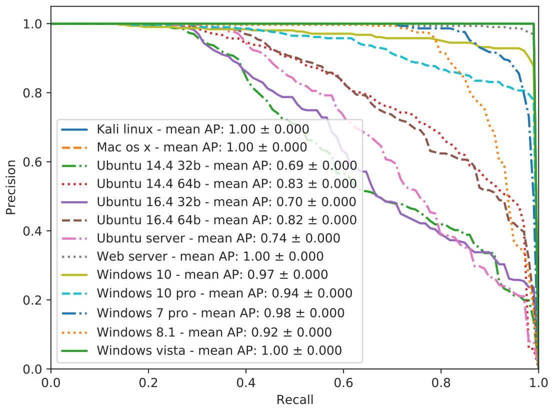
p0f extracts several fields from each packet and compares extracted values against a fingerprint database to find a matching OS for the extracted fields. We run p0f, without modifications, on each of the 1, 10, and 100 packet samples created from each device's traffic. In the case of nPrint, we take each 1, 10, and 100 packet traffic sample and transform it into a nPrint. We remove the IP source address, IP destination address, TCP source and destination ports, and TCP sequence and acknowledgement numbers from each nPrint to avoid learning direct identifiers of specific devices rather than general operating system characteristics. Further, we configure nPrint to only output the IP and TCP headers to fairly compare nPrint with p0f. For multi-packet nPrints (10 and 100 packet samples), we concatenate nPrints.

p0f outputs an operating system guess only on packets that directly match a fingerprint in p0f's 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 2 shows the precision and recall for each separate experiment.

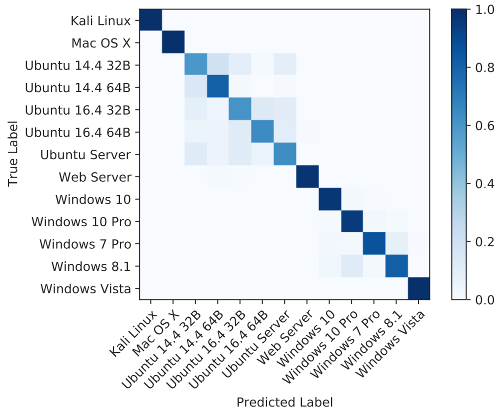
Table 2 illustrates the relatively coarse-grained output generated by p0f. For example, p0f classifies all Ubuntu devices, and the web server, as "Linux 3.11 and newer", and all of the Windows devices as "Windows 7 or 8". Table 2 shows that p0f generally increases in performance when given access to more packets in a sample, and that it is precise when it does make an estimate. Unfortunately, we also see that p0f's recall is generally quite low until given access to 100 packet samples for a device, as p0f does not offer any estimates for many samples. Finally, p0f never outputs a single vote for the Kali Linux traffic samples.

nPrint can perform OS classification with higher recall and fewer packets. To directly compare nPrint to p0f, we use the "p0f Label" in Table 2 as a label for each host and train each classifier using the 3 coarse-grain classes on the same samples that p0f was executed on. Table 2 shows the results of this classification. As shown, models trained on nPrint achieve nearly the same precision as p0f while drastically improving upon p0f's recall. Interestingly, with only one packet, a model trained on nPrint can outperform p0f when given access to 100 packet samples. We also see very slight improvements in model performance when increasing the number of packets in each nPrint.

nPrint can uncover finer-grained differences in OSes than p0f. Next, we aim to uncover if models trained on nPrint can perform passive OS detection at a more fine-grained level. We take the same 1, 10, and 100 packet samples and train each model using the "Host" column of Table 2 as a label for each sample, resulting in a 13-class classification problem. Figure 4a shows the PR curve of the highest performing model using 100 packets. nPrint reveals finer-grained differences in operating systems than P0f.



(a) *nPrint PR.*

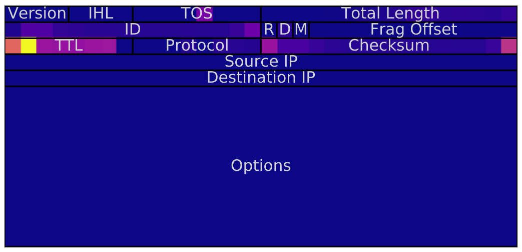


(b) *nPrint Confusion Matrix.*

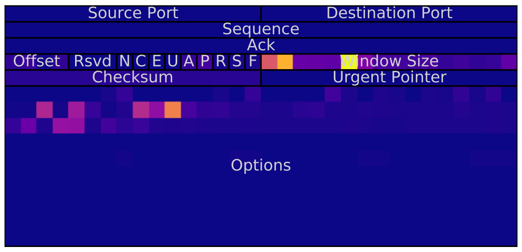
Figure 4: *Passive OS fingerprinting PR and confusion matrix for nPrint. Models trained with nPrint learn to identify operating systems at a finer granularity (e.g., versions) than p0f.*

Figure 4b examines in more detail the performance of the model trained on nPrint. The classifier learns to separate operating system characteristics, with almost all of the confusion being *within the same operating system*. The classifier can separate the devices into more fine-grained classes than p0f's original output, finding differences in Windows Vista, Kali Linux, and the Web Server that is not encoded into p0f. Figure 4b also illustrates a challenge in fine-grained OS detection, as bit version differences of the same OS tend to lead to confusion. This result is unsurprising, as we anticipate the network stack for an OS would be similar between different bit versions.

nPrint differentiates OSes, not simply devices. Finally, we seek to verify that nPrint detects operating systems generally, rather than learning to identify specific devices. We construct an experiment where we compare sets of devices



(a) *IPv4*



(b) *TCP*

Figure 5: *Per-bit feature importance for passive OS detection. 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.*

that share a common OS. We take the five Ubuntu hosts and five Windows hosts in the dataset and set up a binary classification task where 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 to memorizing individual device characteristics.

nPrint can be used to understand semantic feature importance. nPrint is unique as it is interpretable at the *bit level*. We can use this ability to map nPrint back to the semantic realm (*i.e.*, packet headers) to examine the features that are driving the performance of the model. Figure 5 shows a heatmap of the feature importance gathered from the highest performing model trained on nPrint. This visualization can help provide insight into what is differentiating the classes for a specific traffic analysis problem.

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 [5,28]. In the TCP header, the window size field is the most important feature.

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

Table 3: *The active device fingerprinting dataset.*

We also observe that certain bits in the TCP options can help determine OS as some OSes include particular options by default such as maximum segment size, window scaling, or selective acknowledgement permitted.

5.3.2 Active Device Fingerprinting

Next, we examine the utility of nPrint to extract features from packets in the context of active device fingerprinting. We compare the performance of models trained with nPrint to Nmap, perhaps the most popular device fingerprinting tool, which has been developed for over 20 years. We highlight nPrint's ability to adapt to an entirely different context and nPrint's ability to outperform hand-generated features developed for decades.

Input (Packets). We use a dataset of labeled devices and fingerprints to compare nPrint's performance to Nmap's hand-engineered features. Holland *et al.* previously used a subset of Nmap's probes to fingerprint network device vendors at Internet scale [19]. 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 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 compare the performance of nPrint with Nmap. 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, two IoT cameras and two IoT TV streaming devices through Shodan [38]. The final dataset is shown in Table 3.

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

Table 4: *Nmap's highly complex device detection tests, which are used to generate a fingerprint for each device.*

Holland *et al.*'s dataset includes the Nmap output and raw packet responses for each label in the dataset. We modify Nmap to output the raw responses for each probe and Nmap each IoT device added through Shodan labeling. We only have access to the responses to the probes that Nmap sends, not the actual sent probes, as we cannot re-scan the labeled router dataset due to the chance that the device underneath the IP address may have changed.

Packet Transformation and Data Representation. Nmap transforms the responses to each probe into a fingerprint using a series of varyingly complex tests. Table 4 outlines the full set of tests Nmap performs on the probe responses. We use this fingerprint for our evaluation in two ways. First, Nmap compares the fingerprint generated to its database, making a classification of the remote device. Second, we take the fingerprint and encode each test as a categorical feature, creating a feature vector to be used with machine learning methods. Holland *et al.* show this technique to be effective using Nmap's closed-port probes, which comprise only 6 of the 16 probes Nmap sends to each device. We consider every test Nmap conducts when transforming each fingerprint into a feature vector.

nPrint uses 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 have access to the received packets in each nPrint.

We transform every response into a nPrint, configured to include IPv4, TCP, and ICMP headers, resulting in 1,024 features per nPrint. We then aggregate all responses from each device to create a 21-packet nPrint for each device, which is accomplished in one line of python code. We point out that there are 21 rows in each nPrint, while Nmap sends only 16 probes to the device. Nmap re-sends 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 that Nmap does not use, and at worst duplicates data already in the nPrint. We fill any probe response with zeros if the device did not respond to the probe. Finally, as each probe is specifically named, we sort each aggregated nPrint by the probe name to ensure consistent order across each nPrint.

Nmap's heuristics perform poorly for some devices. Nmap compares its generated fingerprint to a hand-curated fingerprint database using a carefully-tuned heuristic. The weights of this heuristic can be seen in Table 4. Nmap either outputs a direct guess of the device, which it is more confident

	Direct		Aggressive	
Label	Precision	Recall	Precision	Recall
Adtran	0.95	0.24	0.70	1.00
Avtech	0.00	0.00	0.00	0.03
Axis	0.00	0.00	0.01	0.52
Chromecast	0.00	0.00	0.00	0.33
Cisco	0.99	0.56	0.95	0.99
Dell	0.11	0.27	0.11	0.87
H3C	0.00	0.00	0.31	0.85
Huawei	0.76	0.24	0.55	0.87
Juniper	0.99	0.48	0.72	0.98
Lancom	0.05	0.00	0.15	0.84
Mikrotik	0.00	0.00	0.04	0.47
NEC	0.00	0.00	0.60	0.99
Roku	0.00	0.00	0.00	0.00
Ubiquoss	0.00	0.00	0.00	0.00
ZTE	0.00	0.00	0.00	0.00

Table 5: *Nmap's heuristic performance only excels on Cisco devices, with significantly lower performance across IoT devices.*

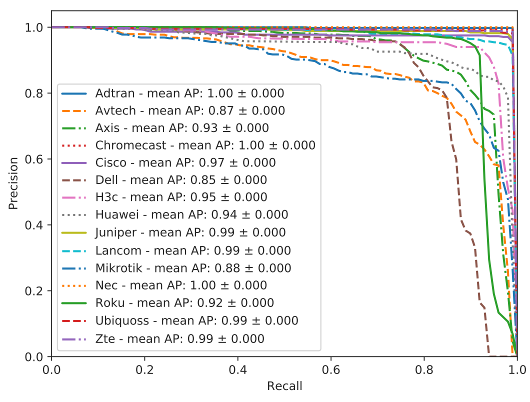
Representation	Balanced Accuracy	ROC AUC	F1
nPrint	95.4	99.7	95.5
Nmap	92.7	99.3	92.9

Table 6: *Models trained on nPrint outperform Nmap's hand-engineered features.*

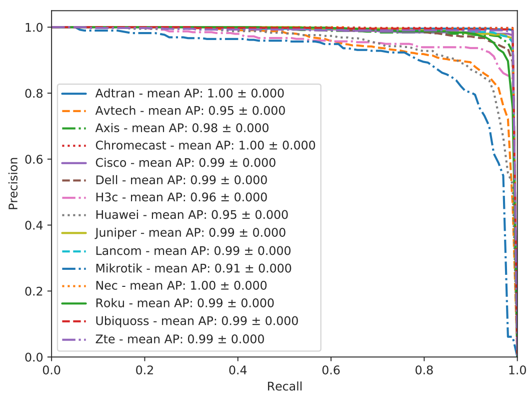
in, or an aggressive guess, where there is less confidence in the output. Table 5 shows the performance of this heuristic on the entire dataset using both direct and aggressive metrics. We see that Nmap's performance is low across the entire dataset with the exception of Cisco, Juniper, and Adtran devices, for which it has high precision with relatively low recall. We notice a trend in the devices that Nmap is accurate at classifying: older, North American based routers. Of all of the IoT devices in the dataset, Nmap's highest heuristic precision is .01.

nPrint outperforms an ML-enhanced Nmap. To better compare Nmap's decades of engineered features to nPrint, we enhance Nmap by replacing its heuristic with ML. We take each one-hot-encoded Nmap fingerprint and run the AutoGluon pipeline, creating optimized models for the features. We also run the AutoGluon pipeline on the nPrint fingerprint generated for each device.

Figure 6 shows the PR curves of the highest performing classifier for Nmap and nPrint. Immediately, we see that not only can models trained on nPrint differentiate the devices without manual feature engineering, nPrint outperforms Nmap's hand-engineered features. Table 6 further examines the best performing model for both Nmap and nPrint. We see that nPrint outperforms Nmap's long-developed features in every metric without access to the sent probes.



(a) *Nmap PR*



(b) *nPrint PR*

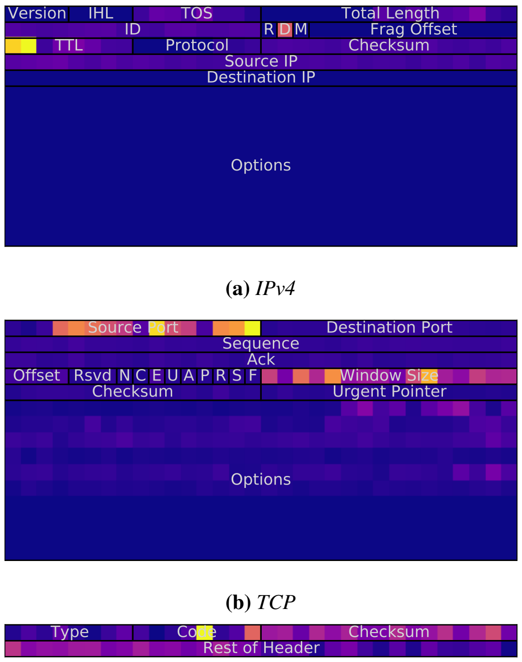
Figure 6: *nPrint achieves higher performance than expert-derived features, even without access to Nmap's sent probes.*

Table 7 examines the training and inference time of the two highest performing models for both Nmap and nPrint. We see that although the highest performing nPrint model takes longer to train than on Nmap's features, models trained on nPrint can achieve higher F1 scores than models trained on Nmap's features while having a lower training time and faster inference speed.

Representation	Model	# Models in Ensemble	Training Time (Seconds)	Inference Time (Seconds)	F1
Nmap	Weighted Ensemble	2	497	12	92.8
Nmap	Weighted Ensemble	3	630	32	92.9
nPrint	Weighted Ensemble	3	775	267	95.4
nPrint	Weighted Ensemble	2	188	3	95.2

Table 7: *models trained on nPrint can outperform Nmap in terms of F1 score while performing faster inference.*

nPrint can be used to understand semantic feature importance. We again use the ability to map nPrint back to the semantic realm to examine the features that are driving the



(a) *IPv4*

(b) *TCP*

(c) *ICMP*

Figure 7: *Per-bit feature importance for active fingerprinting. 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.*

performance of the model. Figure 7 shows a heatmap of the feature importance gathered from the model trained on nPrint.

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 the ICMP code value that the device responds with are both helpful in identifying the device.

5.3.3 DTLS Application Identification

Finally, we test the ability of nPrint to identify a set of applications through their DTLS handshakes. We aim to automatically identify the application and browser that generated a DTLS handshake with nPrint when provided with the handshake traffic. MacMillan *et al.* examined the feasibility of fingerprinting Snowflake, a pluggable transport for Tor that uses WebRTC to establish browser-to-browser connections [27], 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

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

Table 8: *The application identification dataset [27].*

Chrome. They then extract features from the handshakes to show that Snowflake is identifiable with perfect accuracy.

We are interested demonstrating nPrint's ability to automate this process entirely. This DTLS classification problem highlights both nPrint's ability to adapt to entirely new tasks and nPrint's performance in a noisy environment, as the packet traces vary in length due to retransmission in the UDP-based handshakes.

Input (Packets). Table 8 shows an overview of the nearly 7,000 DTLS handshakes in the dataset. While MacMillan *et al.* examine the classification task solely at the application level, we *further* split the classification task into the specific (browser, application) pair created the handshake, increasing the number of classes in the task from four to seven. Snowflake traffic is specific to the Firefox browser and did not have any Chrome instances.

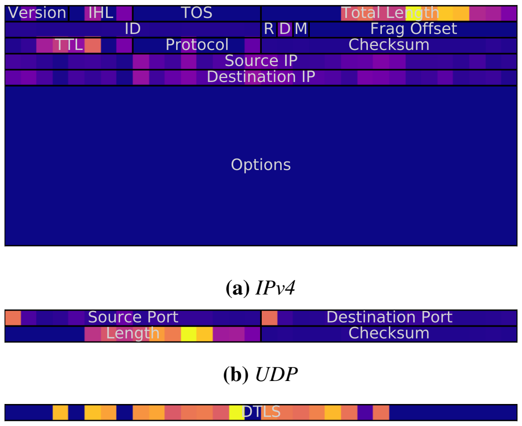
Packet Transformation and Data Representation. We take each handshake, which was captured and filtered as a PCAP file, and transform it into a nPrint, configured to consist of the IPv4, UDP, and first 10 bytes of the TLS payload. We choose the first 10 bytes as the first few bytes of the TLS handshake messages identify which type of DTLS handshake message is contained. The number of packets in the packet captures varies from 4 to 50 due to both client behavior and packet retransmissions. We pad each fingerprint with nPrints of zeros to the maximum capture size and allow the models trained on nPrint to identify important features in the noisy traffic.

nPrint can automatically detect features in a noisy environment. We run AutoGluon on the handshakes in nPrint format. The weighted ensemble classifier trained on nPrint achieves a perfect ROC AUC score, 99.8% accuracy, and 99.8% F1 score. nPrint can almost perfectly identify the (browser, application) pair that generates each handshake. While in prior work manually engineered features achieved the same accuracy on an easier version of the problem, nPrint avoids model selection and feature engineering entirely, matching the performance of manual features and models on a more difficult instance of the problem [27].

nPrint performs well across models and trains quickly. Table 9 shows the F1 score, training time, and total inference time on the testing dataset for each non-ensemble classifier trained and the weighted ensemble classifier with the highest overall performance. We find that nPrint works across models,

Model Architecture	Fit Time (Seconds)	Total Inference Time (Seconds)	F1
Random Forest	3.69	0.37	99.8
ExtraTrees	3.89	0.43	99.9
KNeighbors	3.90	8.95	96.0
LightGBM	5.21	0.15	99.8
Catboost	9.00	0.38	99.7
Weighted Ensemble	46.1	0.45	99.9
Neural Network	85.58	29.9	99.7

Table 9: *nPrint performs well across models, with training and inference times varying depending on the type of modle.*



(c) *DTLS*

Figure 8: *Per-bit feature importance for (browser, application) identification. Brighter colors are more important. Given the nPrint representation, ML models can automatically learn important features (e.g., length), as opposed to relying on manual engineering.*

with inference and training time varying. nPrint working well across different models is important as some environments require different optimizations. For example, classification in streaming environments may prefer faster inference time over the highest performing model.

nPrint can be used to understand semantic feature importance. We use nPrint to map feature importance to their semantic meaning. Figure 8 shows a heatmap of the feature importance gathered from the highest performing random forest model trained on nPrint. This visualization can help provide insight into what is differentiating the classes for a specific traffic analysis problem. In this instance, the successive lengths of the headers drive much of the performance in nPrint. We also see that the first 10 bytes of the DTLS payload help drive the performance of the classifier. We make this heatmap code public (upon publication) to allow others to analyze the bits that are driving classification performance for a given problem.

6 Conclusion

This paper presented nPrint, a standard, binary, packet-level representation of network traffic that can be applied to many network traffic analysis problems. We developed design requirements for a standard representation of network traffic, including the need for a complete, aligned, constant size, inherently normalized representation that is usable with many machine learning models. We combine nPrint with a state-of-the-art AutoML tool and evaluate nPrint's flexibility on three discrete problems: operating system detection, as compared with p0f; device identification, as compared with the widely used Nmap approach; and application identification, based on the contents of DTLS handshakes.

Our evaluation demonstrates that models trained with nPrint are at least as accurate as existing bespoke solutions to individual fingerprinting problems. In contrast to existing approaches, which involve manual feature engineering for each specific traffic analysis task, nPrint is applicable to a wide variety of tasks. nPrint's standard, protocol-agnostic nature also allows for models to be easily retrained in the face of changing operational environments and scenarios (e.g., software upgrades, adversarial evasion) without additional feature engineering or model re-design and retraining.

We have released nPrint as an open-source software tool along with reference implementations and examples, as well as extensive documentation. Our goal is ultimately to make inference problems using network traffic as seamless as possible by enabling nPrint to be deployed in a wide range of settings and making it as easy as possible to couple its output to AutoML tools. We encourage the research and operations communities to apply it to other traffic analysis problems, extending the representation as appropriate.

Our evaluation shows promise for nPrint's generality, but the true test will be its widespread use over time, across more network inference tasks of many different types. Along these lines, the current tool contains capabilities for representation that we ourselves have not yet explored; for example, nPrint can encode certain aspects of traffic such as sequencing and timing relationships (e.g., packet sequences, relative timestamps). Although we have not considered problems that use this information in this work, the existence of this information in nPrint make it a natural direction for future work.

References

[1] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé. Mobile encrypted traffic classification using deep learning: Experimental evaluation, lessons learned, and challenges. *IEEE Transactions on Network and Service Management*, 16(2):445–458, 2019.

[2] A. Agrapetidou, P. Charonyktakis, P. Gogas, T. Papadimitriou, and I. Tsamardinos. An automl application to forecasting bank failures. *Applied Economics Letters*, pages 1–5, 2020.

[3] M. AlSabah, K. Bauer, and I. Goldberg. Enhancing Tor's performance using real-time traffic classification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 73–84, New York, NY, USA, 2012. Association for Computing Machinery.

[4] J. Barker, P. Hannay, and P. Szewczyk. Using traffic analysis to identify the second generation onion router. In *2011 IFIP 9th International Conference on Embedded and Ubiquitous Computing*, pages 72–78, Oct 2011.

[5] S. M. Bellovin. A technique for counting natted hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment*, IMW '02, page 267–272, New York, NY, USA, 2002. Association for Computing Machinery.

[6] R. Beverly. A robust classifier for passive TCP/IP fingerprinting. In *Proceedings of the 5th Passive and Active Measurement (PAM) Workshop*, Apr. 2004.

[7] V. Brik, S. Banerjee, M. Gruteser, and S. Oh. Wireless Device Identification with Radiometric Signatures. In *Proceedings of the 14th ACM international conference on Mobile computing and networking*, pages 116–127, 2008.

[8] D. E. Comer and J. C. Lin. Probing TCP implementations. In *Usenix Summer*, pages 245–255, 1994.

[9] Z. Durumeric, E. Wustrow, and J. A. Halderman. Zmap: Fast internet-wide scanning and its security applications. In *Proceeedings of the 22nd USENIX Security Symposium*, pages 605–620, 2013.

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

[11] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505*, 2020.

[12] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *Advances in neural information processing systems*, pages 2962–2970, 2015.

[13] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter. Auto-sklearn: efficient and robust automated machine learning. In *Automated Machine Learning*, pages 113–134. Springer, Cham, 2019.

[14] D. Formby, P. Srinivasan, A. M. Leonard, J. D. Rogers, and R. A. Beyah. 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*, 2016.

[15] J. François, H. Abdelnur, R. State, and O. Festor. Machine Learning Techniques for Passive Network Inventory. *IEEE Transactions on Network and Service Management*, 7(4):244–257, 2010.

[16] B. Greschbach, T. Pulls, L. M. Roberts, P. Winter, , and N. Feamster. The Effect of DNS on Tor's Anonymity. In *Network and Distributed System Security Symposium*, Feb. 2017.

[17] J. Hayes and G. Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1187–1203, Austin, TX, Aug. 2016. USENIX Association.

[18] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[19] J. Holland, R. Teixeria, P. Schmitt, K. Borgolte, J. Rexford, N. Feamster, and J. Mayer. Classifying network vendors at internet scale. *arXiv preprint arXiv:2006.13086*, 2020.

[20] R.-H. Hwang, M.-C. Peng, V.-L. Nguyen, and Y.-L. Chang. An lstm-based deep learning approach for classifying malicious traffic at the packet level. *Applied Sciences*, 9(16):3414, 2019.

[21] H. Jin, Q. Song, and X. Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956, 2019.

[22] T. Kohno, A. Broido, and K. C. Claffy. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing*, 2(2):93–108, 2005.

[23] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, and K. Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *The Journal of Machine Learning Research*, 18(1):826–830, 2017.

[24] E. LeDell and S. Poirier. H2o automl: Scalable automatic machine learning. In *Proceedings of the AutoML Workshop at ICML*, volume 2020, 2020.

[25] R. Lippmann, D. Fried, K. Piwowarski, and W. Streilein. Passive operating system identification from TCP/IP packet headers. In *Workshop on Data Mining for Computer Security*, volume 40, 2003.

[26] G. F. Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, USA, 2009.

[27] K. MacMillan, J. Holland, and P. Mittal. Evaluating snowflake as an indistinguishable censorship circumvention tool. *arXiv preprint arXiv:2008.03254*, 2020.

[28] T. Miller. Passive OS Fingerprinting: Details and Techniques. http://www.ouah.org/incosfingerp.htm, Feb. 2020.

[29] S. Nayak, B. B. Misra, and H. S. Behera. Impact of data normalization on stock index forecasting. *International Journal of Computer Information Systems and Industrial Management Applications*, 6(2014):257–269, 2014.

[30] ntop. Pf_ring, high-speed packet capture, filtering and analysis. pfring.

[31] S. E. Oh, S. Sunkam, and N. Hopper. p1-fp: Extraction, classification, and prediction of website fingerprints with deep learning. *Proceedings on Privacy Enhancing Technologies*, 2019(3):191–209, 2019.

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

[33] J. Padhye and S. Floyd. On inferring TCP behavior. *SIGCOMM Comput. Commun. Rev.*, 31(4):287–298, Aug. 2001.

[34] A. Panchenko, F. Lanze, J. Pennekamp, T. Engel, A. Zinnen, M. Henze, and K. Wehrle. Website fingerprinting at internet scale. In *NDSS*, 2016.

[35] V. Paxson. Automated packet trace analysis of TCP implementations. *SIGCOMM Comput. Commun. Rev.*, 27(4):167–179, Oct. 1997.

[36] V. Rimmer, D. Preuveneers, M. Juárez, T. van Goethem, and W. Joosen. Automated website fingerprinting through deep learning. In *Network and Distributed System Security Symposium, NDSS*, San Diego, CA, USA, Feb. 2018.

[37] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *ICISSP*, 2018.

[38] Shodan. Shodan. https://www.shodan.io/, Jan. 2020.

[39] D. Singh and B. Singh. Investigating the impact of data normalization on classification performance. *Applied Soft Computing*, page 105524, 2019.

[40] P. Sirinam, M. Imani, M. Juarez, and M. Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. *arXiv preprint arXiv:1801.02265*, 2018.

[41] M. Smart, G. R. Malan, and F. Jahanian. Defeating tcp/ip stack fingerprinting. In *Usenix Security Symposium*, 2000.

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

[43] A. Sperotto, M. Mandjes, R. Sadre, P.-T. de Boer, and A. Pras. Autonomic parameter tuning of anomaly-based idss: an ssh case study. *IEEE Transactions on Network and Service Management*, 9(2):128–141, 2012.

[44] V. Thangavelu, D. M. Divakaran, R. Sairam, S. S. Bhunia, and M. Gurusamy. Deft: A Distributed IoT Fingerprinting Technique. *IEEE Internet of Things Journal*, 6(1):940–952, 2018.

[45] R. Trimananda, J. Varmarken, A. Markopoulou, and B. Demsky. Packet-level signatures for smart home devices. In *Network and Distributed System Security Symposium, NDSS*, San Diego, CA, USA, Feb. 2020.

[46] S. Venkataraman, J. Caballero, P. Poosankam, M. Kang, and D. Song. Fig: Automatic fingerprint generation. In *Network and Distributed System Security Symposium, NDSS*, 01 2007.

[47] L. Wang, K. P. Dyer, A. Akella, T. Ristenpart, and T. Shrimpton. Seeing through network-protocol obfuscation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 57–69, New York, NY, USA, 2015. Association for Computing Machinery.

[48] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective attacks and provable defenses for website fingerprinting. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 143–157, San Diego, CA, Aug. 2014. USENIX Association.

[49] T. Wang and I. Goldberg. Improved website fingerprinting on tor. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pages 201–212, 2013.

[50] T. Wang and I. Goldberg. Walkie-talkie: An efficient defense against passive website fingerprinting attacks. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 1375–1390, USA, 2017. USENIX Association.

[51] W. Wang, Y. Sheng, J. Wang, X. Zeng, X. Ye, Y. Huang, and M. Zhu. Hast-ids: Learning hierarchical spatial-temporal features using deep neural networks to improve intrusion detection. *IEEE Access*, 6:1792–1806, 2017.

[52] W. Wang, M. Zhu, J. Wang, X. Zeng, and Z. Yang. 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, July 2017.

[53] N. Williams, S. Zander, and G. Armitage. A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification. *SIGCOMM Comput. Commun. Rev.*, 36(5):5–16, Oct. 2006.

[54] Y. Yu, J. Long, and Z. Cai. Network intrusion detection through stacking dilated convolutional autoencoders. *Security and Communication Networks*, 2017, 2017.

A Appendix

A.1 Option Representation Evaluation

Section 3 outlines a set of representation requirements and pitfalls of strawman representations. One issue with a semantic representation that renders it unusable as a *standard* representation is that option ordering is not preserved. We now aim to exhibit the performance degradation that can occur when option ordering is not preserved.

We set up a classification problem using the dataset fully explained in Section 5.3.2. At a high level, the dataset consists of fingerprints for 15 classes of devices probed with Nmap. We take the TCP responses from each fingerprint and generate two representations of the TCP options in each packet, one using nPrint, and one using a semantic representation. For the semantic representation we parse all of the options and consider each TCP option as a continuous valued feature, with the name of the feature being the TCP option and the value being its corresponding value in the packet. nPrint fingerprints are the bitmap representation of the options, which preserves ordering.

Model	Representation	F1
Catboost	Semantic	75.1
ExtraTrees		72.9
LightGBM		74.4
Neural Network		68.2
Random Forest		73.6
Weighted Ensemble		75.6
Catboost	nPrint	85.2
Extra Trees		83.3
LightGBM		83.1
Neural Network		82.3
Random Forest		83.3
Weighted Ensemble		85.9

Table 10: *Option ordering increases performance across all models.*

Table 10 shows the performance degradation that occurs when ordering is lost across a wide array of models. In general, we see over 10% increase in F1 scores for nPrint over the semantic representation.