New Directions in Automated 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

Despite the prevalence of machine learning in many network traffic analysis tasks, from application identification to intrusion detection, the aspects of the machine learning pipeline that ultimately determine the performance of the model—feature selection and representation, model selection, and parameter tuning—remain manual and painstaking. This paper presents a method to automate these steps. We introduce nPrint, a tool that generates a unified packet representation that is amenable for representation learning and model training. We integrate nPrint with automated machine learning (AutoML), resulting in nPrintML, a pipeline that can quickly automate many network traffic analysis tasks. nPrintML often outperforms best known results for existing problems while automating many manual steps of the process. We have released nPrint, nPrintML, and the corresponding datasets from our evaluation to enable future work to build on these methods.

1 INTRODUCTION

Network traffic analysis often relies on machine learning (e.g., application identification [8, 25, 51], device identification [17, 26]). Although research has paid much attention to the machine learning models applied to these tasks and the performance of those models, in practice these tasks rely heavily on a pipeline that involves manually engineering features and selecting and tuning tuning models. Arriving at an appropriate combination of features, model, and model parameters is typically an iterative process. Indeed, the effectiveness of applying machine learning to network traffic analysis tasks often depends on the selection and appropriate representation of features as much as the model itself, yet this part of the process has remained exacting and manual.

Feature engineering and model selection are painstaking processes, typically requiring substantial specialized domain knowledge to engineer features that are both practical to measure or derive and result in an accurate model. Even with expert domain knowledge, feature exploration and engineering remains largely a brittle and imperfect process. Such manual analysis may omit features that either were not immediately apparent or involve complex relationships (e.g., non-linear relationships between features). Furthermore, traffic patterns and conditions invariably shift, rendering models and hand-crafted features obsolete [4, 19]. Finally, every new network detection or classification task requires re-inventing the wheel: engineering new features, selecting appropriate models, and tuning new parameters.

This paper charts a new direction for applying machine learning to problems in networking, exploring whether and how a single, standard representation of a network packet can serve as a building block for the automation of many common traffic analysis tasks. Our goal is not to retread any specific network classification problem, but rather to argue that many of these problems can be made easier—and in some cases, completely automated—with a unified representation of traffic that is amenable for input to existing automated machine learning (AutoML) pipelines [14]. To demonstrate this capability, we designed a standard packet representation, *nPrint*, that encodes each packet in an inherently normalized, binary representation while preserving the underlying semantics of each packet. nPrint enables machine learning models to automatically discover important fields of packets for each distinct classification task largely removing many of the most painstaking aspects of applying machine learning to traffic analysis tasks. This paves the way for faster iteration and deployment of machine learning algorithms for networking, lowering the barriers to practical deployment.

The integration of nPrint with AutoML, a pipeline we call nPrintML, enables automated model selection and hyperparameter tuning, enabling researchers to create entire traffic analysis pipelines with nPrint—often without writing a single line of code. Our evaluation shows that models trained on nPrint can perform finer-grained OS detection than p0f [38], achieve higher accuracy than Nmap [31] in device fingerprinting, and automatically identify applications in traffic with retransmissions. To our knowledge, we are the first to apply AutoML techniques to network traffic analysis.

This paper makes several contributions:

(1) The design of a unified packet representation that takes raw network packets and transforms them into a format that is amenable for representation learning and model training.
(2) The implementation and release of an open-source tool, nPrint, that can perform this transformation, both on

Problem Overview			nPrintML					Comparison	
Description	Dataset	# Classes	Configuration (Appendix A.5)	Sample Size (# Packets)	Balanced Accuracy	ROC AUC	Macro F1	Score	Source
Passive OS Detection	CICIDS 2017 [44]	3	-4 -t	1	99.5	99.9	99.5	81.3 (Macro-F1)	p0f [38]
				10	99.9	100	99.9		
				100	99.9	100	99.9		
		13		100	77.1	97.5	76.9	No Previous Work	
Active Device Fingerprinting	Network Device Dataset [22]	15	-4 -t -i	21	95.4	99.7	95.5	92.9 (Macro-F1)	ML-Enhanced Nmap [31]
Application Identification via DTLS Handshakes	DTLS Handshakes [32]	7	-p 25	43	99.9	99.7	99.7	99.8 (Average Accuracy)	Hand-Curated Features [32]
			-p 100		99.9	99.7	99.7		
			-4		99.8	96.9	96.6		
			-4 -u -p 10		99.8	99.9	99.8		
			-u		99.9	99.7	99.5		
			-p 10		95.0	78.8	77.4		
Determine Country of Origin for Android & iOS Application Traces	Cross Platform [42]	3	-4 -t -u -p 50	25	96.8	90.2	90.4	No Previous Work	
Malware Detection for IoT Traces	netML IoT† [6, 28]	2	-4 -t -u	10	92.4	99.5	93.2	99.9 (True Positive Rate)	NetML Challenge Leaderboard [35]
		19			86.1	96.9	84.1	39.7 (Balanced F1)	
Type of Traffic in Capture	netML Non-VPN† [6, 12]	7	-4 -t -u -p 10	10	81.9	98.0	79.5	67.3 (Balanced F1)	
			-4 -t -u		76.1	94.2	75.8		
		18			66.2	91.3	63.7	42.1 (Balanced F1)	
		31			60.9	92.2	57.6	34.9 (Balanced F1)	
Intrusion Detection	netML CICIDS 2017† [6, 44]	2	-4 -t -u	5	99.9	99.9	99.9	98.9 (True Positive Rate)	
		8			99.9	99.9	99.9	99.2 (Balanced F1)	
Identify streaming video (DASH) service via device SYN packets	Streaming Video Providers [10]	4	-4 -t -u -R	10	77.9	96.0	78.9	No Previous Work	
				25	90.2	98.6	90.4		
				50	98.4	99.9	98.6		

Table 1: *Case studies we have performed with nPrintML. nPrintML enables automated high performance traffic analysis across a wide range and combination of protocols. In many cases, models trained on nPrint are able to outperform state-of-the-art tools with no hand-derived features.*

offline traces and online, at speeds of up to 10 Gbps in software.

(3) The integration of nPrint with an automated machine learning (AutoML) pipeline to show how nPrint's unified representation, coupled with state-of-the-art AutoML techniques can automate machine learning on labeled packet traces.

(4) The application of this pipeline, nPrintML, to eight common network classification tasks, demonstrating that nPrintML can enable easy, rapid, and largely automated iteration on a wide variety of network classification problems, typically with better performance than existing models.

Table 1 highlights the performance of nPrintML, which typically outperforms the best known results from the published literature. To enable others to replicate and extend these results—and to apply nPrint and nPrintML to a broader class of problems—we have publicly released nPrint, nPrintML, and all datasets used in this work, to serve as a benchmark for the research community.

†Original netML datasets were not made public. As such, we recreated the datsets from the descriptions of the authors. In the vast majority of cases, we were able to directly match the number of samples for each class. Due to this recreation we do not have the same training and testing split as the original challenge and our results do not reflect a perfect performance comparison. Finally, we remove IP address bits from each nPrint to match the masking of IP addresses in the netML datasets.

Many problems, such as capturing temporal relationships across multiple traffic flows, remain unsolved and unexplored. In this regard, we view nPrint and nPrintML as the first chapter in a new direction for research at the intersection of machine learning and networking that allows us to focus less time on fine-tuning models and features and more time interpreting and deploying the best models in practice.

Figure 1 summarizes the following sections of this paper. Sections 2 and 3 discuss the design and implementation of nPrint. Section 4 introduces AutoML and nPrintML, which combines AutoML and nPrint. Section 5 applies nPrintML for three of the case studies in Table 1. Finally, Sections 6 and 7 examine related works and summarize our contributions.

2 DATA REPRESENTATION

For many classification problems, the data representation 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 2.1. Section 2.2 explores three possible standard representations of network traffic, including a semantic encoding, an unaligned binary representation, and a hybrid approach, which fuses a binary representation of the packet with an encoding that simultaneously preserves the semantic structure of the bits.

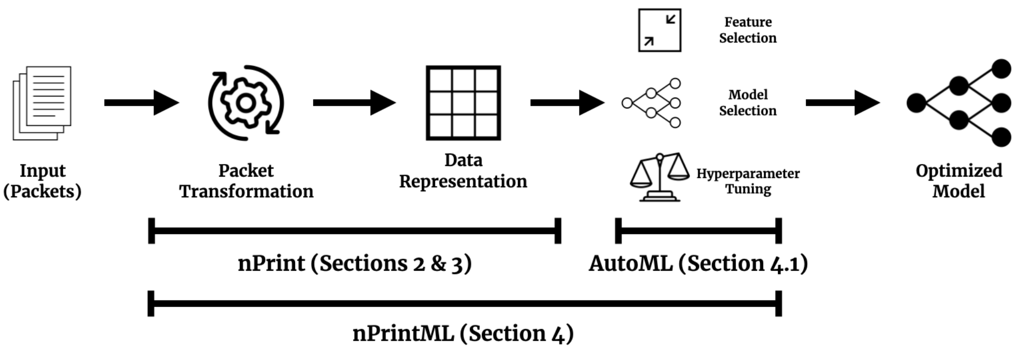


Figure 1: *nPrint produces a standard network traffic representation that can be combined with AutoML tools to create nPrintML, a standard, largely automated traffic analysis pipeline.*

2.1 Design Requirements

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

Constant size per problem. Many 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 knowledge avoids the need for multiple passes over a stored packet trace, and it is essential in data streaming contexts.

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 decreases training time and increases model stability [34, 46]. Of course, it is possible to normalize *any* feature set, but it is more convenient if the initial representation is inherently normalized.

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 in binary form, as both protocols and packets differ in length.

2.2 Building a Standard Data Representation

Network traffic can be represented in multiple ways. We discuss three options: semantic, unaligned binary, and a hybrid representation that we term nPrint.

2.2.1 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 has 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 2 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 [31, 48].

We examine two of the datasets (presented in Sections 5.1 and 5.2) and find 10 and 59 unique TCP option orderings, respectively. We further explore the effect that 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,

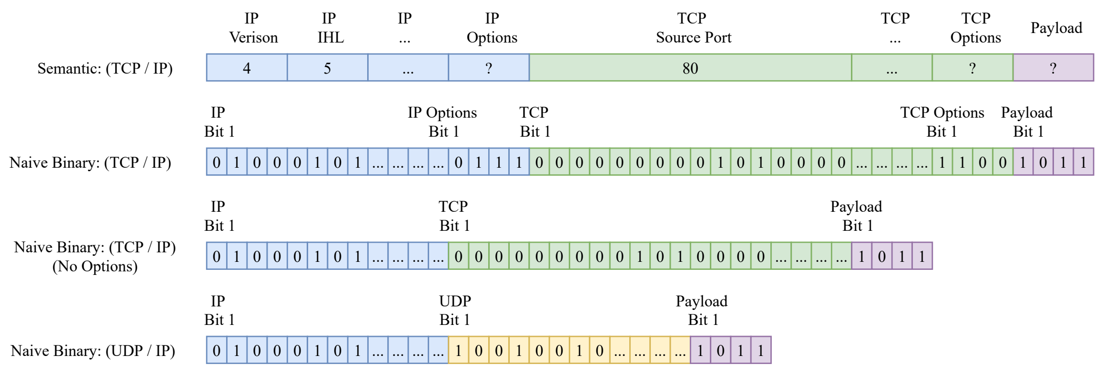


Figure 2: *Semantic and naive binary nPrint representations introduce various problems that make modeling difficult: Semantic loses ordering of specific fields such as the TCP options. Binary nPrints lack alignment, which can degrade performance due to misaligned features.*

or with a one-hot encoding. Semantic features also may need to be normalized depending on the model.

2.2.2 Naive 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 akin to an "image" of each packet. We see an example of this representation in Figure 2. However, transforming each packet into its bitmap representation ignores many intricate details, including varying sizes and protocols. These issues can cause feature vectors for two packets to have 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 2 illustrates this problem.

Such misalignment can occur between two packets of the same protocol: 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 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 it is impossible to map each bit in the representation to a semantic meaning. The ability to understand the features that determine the performance of a given model is especially important in network traffic where the underlying data has semantics, in contrast to image classification, where the underlying data is pixels.

2.2.3 Hybrid: nPrint. Figure 3 shows nPrint, a single-packet representation that can be directly provided as input to machine learning models. 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 non-existing headers with -1, each feature takes on one of three values: -1, 0, or 1. Further, filling non-existing header values with -1 allows us to differentiate between a bit being set to 0 and a header that does not exist in the packet. 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 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, nPrint, which is a single-packet representation, can be used as a building block for classification problems that require sets of packets, as we have done in several cases. If If we consider each nPrint fingerprint as a 1xM matrix, where M

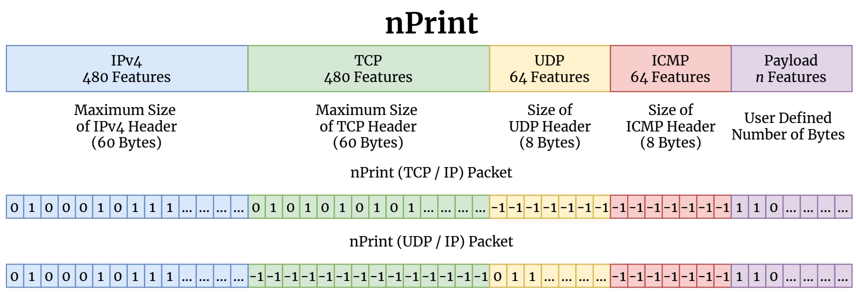


Figure 3: *nPrint, the complete, normalized, aligned packet representation. Headers that do not exist in the packet being transformed are noted with sentinels accordingly, while headers that exist but are not of maximum size are zero padded for alignment across nPrints. nPrint removes reliance on manual feature engineering while avoiding misaligned features.*

is the number of features in the fingerprint, we can build multi-packet nPrint fingerprints by concatenating them.

3 NPRINT IMPLEMENTATION

We implemented nPrint in C++ and evaluated it in different contexts, including its memory footprint and performance at line-rate. nPrint currently supports Ethernet, IPv4, fixed IPv6 headers, UDP, TCP, ICMP, and any corresponding packet payloads. Appendix A.5 shows the full configuration options available in nPrint. nPrint can either process offline packet packet capture formats such as PCAP and Zmap [13] output, or capture packets directly from a live interface. nPrint can also reverse the encoding, creating a PCAP from the nPrint format. The nPrint source code is publicly available.

nPrint transforms over 1 million packets per minute. 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. On this machine, nPrint transformed the three datasets used in Section 5, containing approximately 2.35 million, 274K, and 49K packets, in 49, 13, and 12 seconds respectively. On average, nPrint currently transforms packets at about 1.5 million packets per minute using a single thread.

nPrint has a constant 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 and TCP output (used for p0f evaluation). This configuration results in a constant memory footprint of about 295 KB. A configuration consisting of IPv4, UDP, ICMP, TCP, and 10 payload bytes yields a constant memory footprint of about 310 KB.

nPrint can be leveraged for line-rate processing. nPrint processes each packet independently, making it amenable to parallelization. We test the performance of nPrint on a 10 Gbps 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*) 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 [36].

4 NPRINTML

Much previous work has used a pipeline similar to the full pipeline seen in Figure 1. 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 1 through nPrint and new AutoML systems. To our knowledge, we are the first to combine AutoML techniques and network traffic analysis.

First, we designed nPrint to be directly used in a machine learning pipeline, standardizing the tedious feature development process for a large set of traffic analysis problems. Next, our design choices for nPrint enable it to be directly combined with AutoML tools to standardize 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 model selection and hyperparameter optimization, searching for the highest performing model in a more principled manner than by hand.

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

We build upon our guiding principle 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 more model types, 2) we can optimize the hyperparameters for *every* model we train, and 3) we are certain that the best model is chosen for a given representation.

4.1 AutoGluon-Tabular AutoML

We use AutoGluon-Tabular to perform feature selection, model search, and hyperparameter optimization for all three problems we evaluate [14]. We choose AutoGluon as it has been shown to outperform many other public AutoML tools given the same data [14]. 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 6 different base model classes which are variations of tree-based methods, deep neural networks and neighbors-based classification. The highest performing model for each problem we examine is an ensemble of the base model classes.

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. By selecting a high quality model setting, each model is bagged with 10 folds, decreasing the bias of individual models.

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.

4.2 nPrintML Implementation

We have implemented nPrintML in Python, directly combining nPrint and AutoGluon-Tabular Automl. nPrintML enables researchers to create full traffic analysis pipelines using nPrint in a single program call. Below is an example of fully recreating the passive OS detection case study found in Section 5.

```
nprintml -L os_labels.txt -a index -P traffic.pcap
-4 -t
```

A full tutorial of nPrintML is supplied in Appendix A.2. At the time of writing, nPrintML can create machine learning pipelines from a single traffic trace or an entire directory of traffic traces.

Metrics. nPrintML outputs a standard set of metrics for each classification problem, which we define here. 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. For multi-class classification, we report a macro F1 score, calculated in the same manner as optimized during training.

5 CASE STUDIES

In this section, we highlight the versatility and performance of nPrint. Table 1 presents the breadth of problems that nPrintML can be leveraged for, summarizing eight case studies tested with nPrintML. This section examines three of those case studies in depth: passive OS detection, active device fingerprinting, and application identification. We show that nPrintML can match or beat the performance of existing bespoke approaches for each of these problems.

5.1 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 nPrintML to p0f, one of the most well-known and commonly used passive OS fingerprinting tools. Ultimately, we find that nPrintML can perform OS detection with much higher recall rate than p0f, using much smaller packet sequences. Further, we find that nPrintML can uncover finer-grained differences in OSes compared to p0f. We highlight that nPrintML is able to be used in a variety of manners including single-packet OS detection and using sequences of packets.

		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		1.00	0.01	1.00	0.25	1.00	0.74						
Ubuntu 14.4 32B		1.00	0.04	1.00	0.20	1.00	0.69						
Ubuntu 14.4 64B	Linux 3.11 and newer	1.00	0.04	1.00	0.20	1.00	0.65	0.99	0.99	0.99	1.00	0.99	1.00
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		0.99	0.00	0.98	0.02	0.98	0.09						
Windows 10 Pro		0.99	0.01	0.98	0.04	1.00	0.14						
Windows 7 Pro	Windows 7 or 8	1.00	0.04	1.00	0.23	1.00	0.71	1.00	1.00	1.00	1.00	1.00	1.00
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 nPrintML vs. p0f for passive OS fingerprinting. nPrintML 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 nPrintML.*

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. We note that p0f's fingerprinting capabilities include *much* more breath than is considered in this case study, as it contains signatures for not only OS detection, but browser identification, search robots, and some commonly used applications. We specifically aim to test if models trained on the nPrint representation can exceed the performance of p0f on a common task for the tool and test if nPrintML can be used for automated, finer-grained OS detection than p0f.

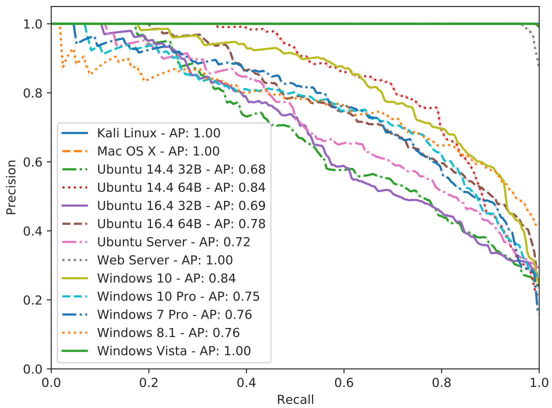
Input (Packets). We use the CICIDS2017 intrusion detection evaluation dataset, which contains PCAPs of over 50 GB of network traffic over the course of 5 days [44]. The traffic contains labeled operating systems ranging from Ubuntu to Windows to MacOS. There are 17 hosts in the IDS dataset, but we find 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 nPrintML 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.

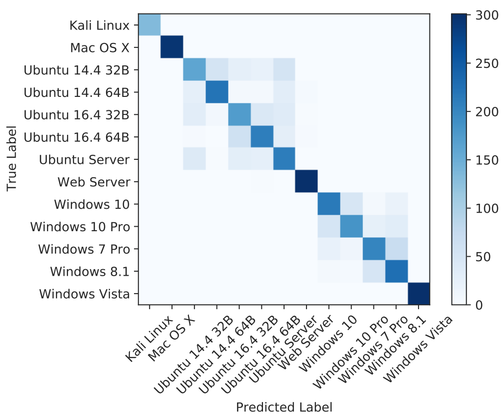
p0f extracts several fields from each packet and compares extracted values against a fingerprint database to find a matching OS for the extracted fields. We run p0f, without modifications, on each of the 1, 10, and 100 packet samples created from each device's traffic. We create separate nPrintML pipelines for the 1, 10, and 100 packet traffic samples, configuring it to remove the IP source address, IP destination address, IP identification, 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 nPrintML to only use IP and TCP headers to fairly compare nPrintML with p0f.

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



(a) *nPrintML PR.*



(b) *nPrintML Confusion Matrix.*

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

does make an estimate. Unfortunately, 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.

nPrintML can perform OS classification with higher recall and fewer packets. To directly compare nPrintML to p0f, we use the "p0f Label" in Table 2 as a label for each host and train each classifier using the 3 coarse-grained classes on the same samples that p0f was executed on. Table 2 shows the results of this classification. As shown, models trained on the nPrint representation achieve nearly the same precision as p0f while drastically improving upon p0f's recall. Interestingly, with only one packet, a model trained on the nPrint representation 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.

nPrintML can reveal finer-grained differences in OSes than p0f. Next, we aim to discover if models trained on the nPrint representation can enable automated passive OS detection at a more fine-grained level. We take the same 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.

Figure 4b examines the performance of the model trained on the nPrint representation in more detail. The classifier separates 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 and Kali Linux that are 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.

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

nPrintML 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 illustrates that models can successfully identify operating systems generally from the nPrint representation, as opposed to memorizing individual device characteristics.

5.2 Active Device Fingerprinting

Next, we examine the utility of nPrintML to extract features from packets in the context of active device fingerprinting. We compare the performance of models trained on the nPrint representation to Nmap, perhaps the most popular device fingerprinting tool, which has been developed for over 20 years. We highlight nPrintML's ability to adapt to an entirely different context in which 3 separate protocols (IP, TCP, ICMP) need to be represented and nPrintML's ability to outperform hand-generated features developed for decades.

Vendor	Device Type	Labeled Devices	Nmap Direct		Nmap Aggressive	
			Precision	Recall	Precision	Recall
Adtran	Network	1,449	0.95	0.24	0.70	1.00
Avtech	IoT	2,152	0.00	0.00	0.00	0.03
Axis	IoT	2,653	0.00	0.00	0.01	0.52
Chromecast	IoT	2,872	0.00	0.00	0.00	0.33
Cisco	Network	1,451	0.99	0.56	0.95	0.99
Dell	Network	1,449	0.11	0.27	0.11	0.87
H3C	Network	1,380	0.00	0.00	0.31	0.85
Huawei	Network	1,409	0.76	0.24	0.55	0.87
Juniper	Network	1,445	0.99	0.48	0.72	0.98
Lancom	Network	1,426	0.05	0.00	0.15	0.84
Mikrotik	Network	1,358	0.00	0.00	0.04	0.47
NEC	Network	1,450	0.00	0.00	0.60	0.99
Roku	IoT	2,403	0.00	0.00	0.00	0.00
Ubiquoss	Network	1,476	0.00	0.00	0.00	0.00
ZTE	Network	1,425	0.00	0.00	0.00	0.00

Table 3: *The active device fingerprinting set and Nmap's heuristic performance on the devices. Nmap's heuristic only excels on Cisco devices, with significantly lower performance across IoT devices.*

Input (Packets). We use a dataset of labeled devices and fingerprints to compare nPrintML'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 [22]. 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 nPrintML'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 nPrintML with Nmap. We further expand the types of devices we are testing to test the adaptability of nPrintML 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 [45]. Table 3 shows the final dataset.

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.

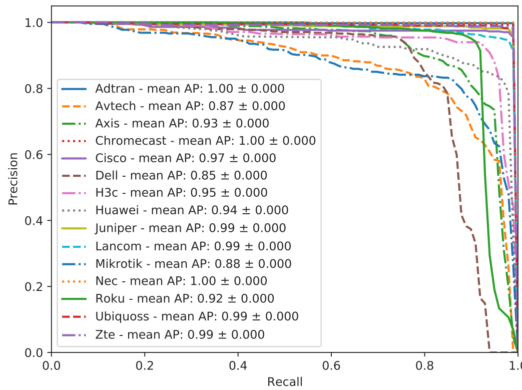
Packet Transformation and Data Representation. Nmap transforms the responses to each probe into a fingerprint using a series of varyingly complex tests developed for decades. Table 9 in Appendix A.4 outlines the full set of tests Nmap performs on the probe responses, which includes complex features computed across sets of packets. 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 that this technique is 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.

nPrintML uses the raw responses generated from the modified version of Nmap to build a nPrint for each device. nPrintML uses the same response packets that Nmap uses to build each 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.

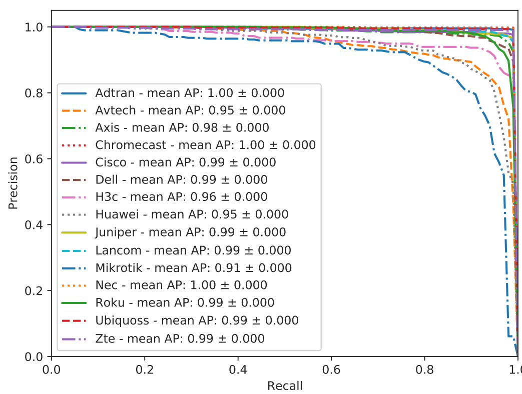
We configure nPrintML to include IPv4, TCP, and ICMP headers in each nPrint, resulting in 1,024 features for each individual packet. nPrintML aggregates all responses from each device to create a 21-packet nPrint for each device. We point out that there are 21 packets in each nPrint, while Nmap sends 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 packet 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. nPrintML fills any probe response with -1s if the device did not respond to the probe. Finally, as each probe is specifically named, we sort each aggregated nPrint created by nPrintML by the probe name to ensure consistent order across all samples.

Nmap's heuristics perform poorly for some devices. Nmap compares its generated fingerprint to a hand-curated fingerprint database using a carefully-tuned heuristic. Table 9 in Appendix A.4 shows weights of this heuristic. Nmap either outputs a direct guess of the device, which it is more confident in, or an aggressive guess, where there is less confidence in the output. Table 3 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.

nPrintML outperforms an ML-enhanced Nmap. To further compare Nmap's decades of engineered features to nPrintML,



(a) *Nmap PR*



(b) *nPrintML PR*

Figure 5: *nPrintML achieves higher performance than expert-derived features, even without access to Nmap's sent probes.*

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

Table 4: *Models trained on the nPrint representation outperform Nmap's hand-engineered features.*

we enhance Nmap by replacing its heuristic with ML. We take each one-hot-encoded Nmap fingerprint and run the AutoGluon AutoML pipeline, creating optimized models for the features. We also run nPrintML on on the nPrint representation generated for each device.

Figure 5 shows the PR curves of the highest performing classifier for Nmap and nPrintML. Immediately, we see that not only can models trained on the nPrint representation differentiate the devices without manual feature engineering,

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 5: *models trained on the nPrint representation can outperform Nmap in terms of F1 score while performing faster inference.*

nPrintML outperforms Nmap's hand-engineered features. Table 4 examines the best performing model for both Nmap and nPrintML. We see that nPrintML outperforms Nmap's long-developed features in every metric without access to the sent probes.

Table 5 examines the training and inference time of the two highest performing models for both Nmap and nPrintML. We see that although the highest performing nPrintML model takes longer to train than on Nmap's features, models trained on the nPrint representation can achieve higher F1 scores than models trained on Nmap's features with a lower training time and faster inference speed. Appendix A.3.1 shows and explains the features driving the performance of the nPrintML model.

5.3 DTLS Application Identification

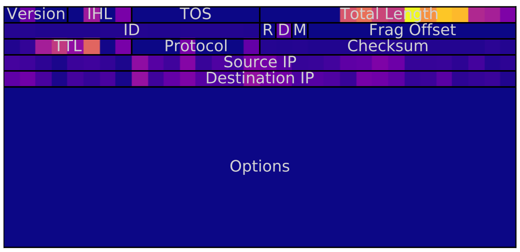
We test the ability of nPrintML to identify a set of applications through their DTLS handshakes. We aim to automatically identify the application and browser that generated a DTLS handshake with nPrintML when provided with the handshake traffic. MacMillan *et al.* examined the feasibility of fingerprinting Snowflake, a pluggable transport for Tor that uses WebRTC to establish browser-to-browser connections [32], 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 Snowflake is identifiable with perfect accuracy.

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

Input (Packets). Table 6 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.

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

Table 6: *The application identification dataset [32].*



(a) *IPv4*

(b) *UDP*

(c) *DTLS*

Figure 6: *Per-bit feature importance for (browser, application) identification, computed by summing the feature importance for each individual nPrint in the 50 packet nPrint created for each handshake. 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.*

Snowflake traffic is specific to the Firefox browser and did not contain any Chrome instances.

Packet Transformation and Data Representation. We configure nPrintML to transform each handshake, which was captured and filtered as a PCAP file, into a nPrint. We vary the configuration of nPrintML to test nPrintML under a variety of constraints. Table 1 shows these configurations. The number of packets in the packet captures varies from 4 to 50 due to both client behavior and packet retransmissions. nPrintML pads each fingerprint to the maximum capture size with -1s and allows the models trained on the nPrint representation to identify important features in noisy traffic.

nPrintML can automatically detect features in a noisy environment. We run nPrintML on the handshakes in nPrint format. The weighted ensemble classifier trained on the nPrint representation achieves a perfect ROC AUC score, 99.8% accuracy, and 99.8% F1 score. nPrintML 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, nPrintML avoids model selection and feature engineering entirely, matching the performance of manual features and models on a more difficult instance of the problem [32]. Finally, we examine models trained on the nPrint representation using different combinations of headers in each packet in the traffic. Table 1 shows that models trained on the nPrint representation can perform this task with high accuracy using *only* the UDP headers in each packet.

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

nPrintML performs well across models and trains quickly. Table 7 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. The nPrint representation works across models, 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.

nPrintML can be used to understand semantic feature importance. We use nPrintML to map feature importance to their semantic meaning. Figure 6 shows a heatmap of the feature importance gathered from the highest performing random forest model trained on the nPrint representation. 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 and the information in the DTLS payload drive much of the performance in nPrintML.

6 RELATED WORK

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

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 [15, 16, 24, 27, 29]. These tools have recently been used for model compression, image classification, and even bank failure prediction [2, 21]. 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 [14]. 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 [14].

Machine learning-based fingerprinting. Machine learning techniques have been applied to network traffic classification and fingerprinting [3, 5, 52, 58]. Wang *et al.* developed an ML-based classification model to detect obfuscated traffic [53]. 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 [49]. Trimananda *et al.* used DBSCAN to identify smart home device actions in network traffic [50].

Other work has used machine learning to identify websites visited through the Tor network [20, 40, 54, 55]. 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 [37, 43, 47]. 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 in contrast considers traffic over any network that can vary in size and protocol.

Deep learning techniques have become popular for network traffic classification problems [1, 23, 57, 59]. Yu *et al.* used convolutional autoencoders for network intrusion detection [59]. 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[56]. 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.

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 [11]. Padhye and Floyd identified differences between TCP implementations to detect bugs in public web servers [39]. Paxson passively identified TCP implementations using traffic traces [41].

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 [9, 30]. P0f passively observes traffic and determines the operating system largely based on TCP behavior [38]. Another common tool is Nmap [31], 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.* [26]. Formby *et al.* passively fingerprint industrial control system devices [18].

7 CONCLUSION

The effectiveness of applying machine learning to network traffic analysis tasks depends on the selection and appropriate representation of features as much as, if not more than, the model itself. This paper creates a new direction for automated traffic analysis, presenting nPrint, a unified packet representation that takes as input raw network packets and transforms them into a format that is amenable to representation learning and model training, thereby automating a part of the machine learning process which until now has been largely painstaking and manual.

This standard format makes it easy to integrate network traffic analysis with existing automated machine learning (AutoML) pipelines. nPrintML, the integration of nPrint with AutoML, automatically learns the best model, parameter settings, and feature representation for the corresponding task. We applied nPrintML to eight common network traffic analysis tasks, improving on the state of the art in many cases.

nPrint has demonstrated that many network traffic classification tasks are amenable to automation, though many open problems exist such as automated timeseries analysis and classification involving multiple flows. nPrint should ultimately be applied to a larger set of classification problems. To this end, we have released nPrint, nPrintML, and all datasets as a benchmark for the research community, to make it easy to both replicate and extend the results from this work.

8 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 theauthor(s) and do not necessarily reflect the views of theUnited States Air Force and DARPA.

REFERENCES

[1] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé. 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] Anna Agrapetidou, Paulos Charonyktakis, Periklis Gogas, Theophilos Papadimitriou, and Ioannis Tsamardinos. 2020. An AutoML application to forecasting bank failures. *Applied Economics Letters* (2020), 1–5.

[3] Mashael AlSabah, Kevin Bauer, and Ian Goldberg. 2012. Enhancing Tor's Performance Using Real-Time Traffic Classification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/2382196.2382208

[4] Blake Anderson and David McGrew. 2017. Machine Learning for Encrypted Malware Traffic Classification: Accounting for Noisy Labels and Non-stationarity. In *Proceedings of the 23rd ACM SIGKDD International Conference on knowledge discovery and data mining*. 1723–1732.

[5] 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

[6] Onur Barut, Yan Luo, Tong Zhang, Weigang Li, and Peilong Li. 2020. NetML: A Challenge for Network Traffic Analytics. *arXiv preprint arXiv:2004.13006* (2020).

[7] Steven M. Bellovin. 2002. A Technique for Counting Natted Hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment (IMW '02)*. Association for Computing Machinery, New York, NY, USA, 267–272. https://doi.org/10.1145/637201.637243

[8] Laurent Bernaille, Renata Teixeira, and Kave Salamatian. 2006. Early Application Identification. In *Proceedings of the 2006 ACM CoNEXT Conference*. 1–12.

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

[10] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. 2019. Inferring streaming video quality from encrypted traffic: Practical models and deployment experience. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 3 (2019), 1–25.

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

[12] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A Ghorbani. 2016. Characterization of encrypted and vpn traffic using time-related. In *Proceedings of the 2nd international conference on information systems security and privacy (ICISSP)*. 407–414.

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

[14] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data. *arXiv preprint arXiv:2003.06505* (2020).

[15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. In *Advances in neural information processing systems*. 2962–2970.

[16] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2019. Auto-sklearn: efficient and robust automated machine learning. In *Automated Machine Learning*. Springer, Cham, 113–134.

[17] 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 *Network and Distributed Systems Security Symposium*.

[18] 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

[19] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A Survey on Concept Drift Adaptation. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 1–37.

[20] 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

[21] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 784–800.

[22] 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).

[23] 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.

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

[25] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. 2005. BLINC: Multilevel Traffic Classification in the Dark. In *ACM SIGCOMM*. 229–240.

[26] 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.

[27] Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. 2017. Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. *The Journal of Machine Learning Research* 18, 1 (2017), 826–830.

[28] Stratosphere Lab. [n. d.]. Stratosphere Lab Malware Capture Facility Project. ([n. d.]). https://www.stratosphereips.org/datasets-malware iot malware.

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

[30] 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.

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

[32] Kyle MacMillan, Jordan Holland, and Prateek Mittal. 2020. Evaluating Snowflake as an Indistinguishable Censorship Circumvention Tool. *arXiv preprint arXiv:2008.03254* (2020).

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

[34] SC Nayak, Bijan B Misra, and Himansu Sekhar Behera. 2014. Impact of data normalization on stock index forecasting. *International Journal of Computer Information Systems and Industrial Management Applications* 6, 2014 (2014), 257–269.

[35] NetML. [n. d.]. NetML Network Traffic Analytics Challenge 2020. ([n. d.]). https://eval.ai/web/challenges/challenge-page/526/leaderboard/1473 netML.

[36] ntop. [n. d.]. PF_RING, High-speed packet capture, filtering and analysis. ([n. d.]). https://www.ntop.org/products/packet-capture/pf_ring/pfring.

[37] 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.

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

[39] Jitendra Padhye and Sally Floyd. 2001. On Inferring TCP Behavior. *SIGCOMM Comput. Commun. Rev.* 31, 4 (Aug. 2001), 287–298. https://doi.org/10.1145/964723.383083

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

[41] Vern Paxson. 1997. Automated Packet Trace Analysis of TCP Implementations. *SIGCOMM Comput. Commun. Rev.* 27, 4 (Oct. 1997), 167–179. https://doi.org/10.1145/263109.263160

[42] Jingjing Ren, Daniel J. Dubois, and David Choffnes. 2019. An International View of Privacy Risks for Mobile Apps. (2019). https://recon.meddle.mobi/papers/cross-market.pdf

[43] 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.

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

[45] Shodan. 2020. Shodan. https://www.shodan.io/. (Jan. 2020).

[46] Dalwinder Singh and Birmohan Singh. 2019. Investigating the impact of data normalization on classification performance. *Applied Soft Computing* (2019), 105524.

[47] 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:cs.CR/1801.02265

[48] Matthew Smart, G Robert Malan, and Farnam Jahanian. 2000. Defeating TCP/IP Stack Fingerprinting.. In *Usenix Security Symposium*.

[49] 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.

[50] 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.

[51] Thijs van Ede, Riccardo Bortolameotti, Andrea Continella, Jingjing Ren, Daniel J Dubois, Martina Lindorfer, David Choffnes, Maarten van Steen, and Andreas Peter. 2020. Flowprint: Semi-supervised Mobile-app Fingerprinting on Encrypted Network Traffic. In *Network and Distributed System Security Symposium*. Internet Society.

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

[53] 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 (CCS '15)*. Association for Computing Machinery, New York, NY, USA, 57–69. https://doi.org/10.1145/2810103.2813715

[54] 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

[55] 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 (SEC'17)*. USENIX Association, USA, 1375–1390.

[56] 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.

[57] 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.

[58] 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), 5–16. https://doi.org/10.1145/1163593.1163596

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

A APPENDIX

A.1 Option Representation Evaluation

Section 2 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 5.2. 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 8: *Option ordering increases performance across all models.*

Table 8 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.

A.2 nPrintML Example

Section 3 introduces nPrintML, our pipeline to connect nPrint and autoML. nPrintML enables researchers with network traffic and code to perform state-of-the-art traffic analysis without writing code in minutes. Here, we present an example of reproducing our results in Section 5.3 *from scratch*.

(1) Clone the repository to get the data.
(2) Uncompress the traffic traces.
(3) Write a small script to generate labels. In cases where we have labels apirori, **no code** needs to be written.
(4) Run the label generation script
(5) run nPrintML to perform traffic analysis.

These steps, instantiated.

(1) `git clone repo_name`
(2) `tar -xvf dataset.tar.gz`
(3)

```
import sys
import pathlib

# Example file name: dataset/facebook/windows_chrome_facebook_1383.pcap
# Get file paths
paths = list(pathlib.Path(sys.argv[1]).rglob('*.pcap'))
for path in paths:
    # Build label
    tokens = str(path.stem).split('_')
    label = '{0}_{1}'.format(tokens[0], tokens[1])
    print('{0},{1}'.format(path, label))
```

(4) `python gen_labels.py dataset/ > labels.txt`
(5) IPv4, UDP, first 10 payload bytes of each packet:

```
nprintML -L labels.txt -a pcap
    --pcap_dir dataset/ -4 -u -p 10
```

(6) First 100 payload bytes of each packet:

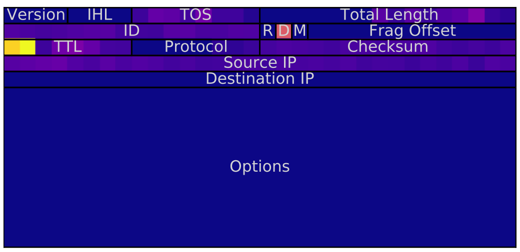
```
nprintML -L labels.txt -a pcap
    --pcap_dir dataset/ -p 100
```

(7) UDP headers only:

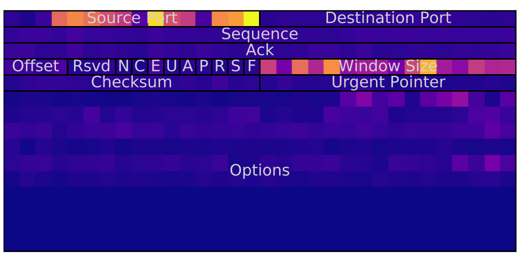
```
nprintML -L labels.txt -a pcap
    --pcap_dir dataset/ -u
```

A.3 Interpretation Heatmaps

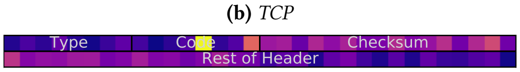
A.3.1 Device Fingerprinting Heatmap. Figure 7 shows a heatmap of the feature importance gathered from the model trained on nPrint. In this instance, we find that the TCP source port of the response probes are one of the more important features in classifying the device vendor. Upon further inspection, 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.



(a) *IPv4*



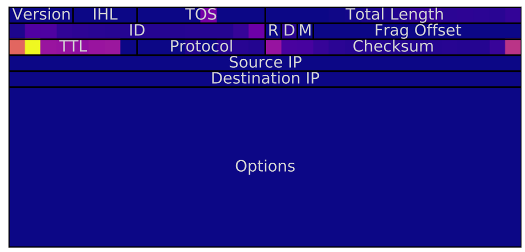
(b) *TCP*



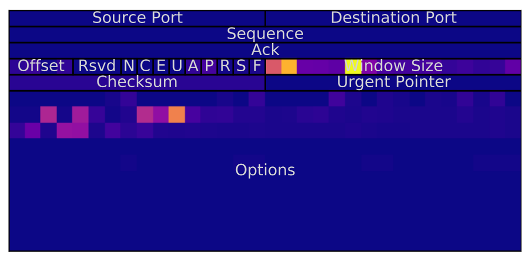
(c) *ICMP*

Figure 7: *Per-bit feature importance for active fingerprinting, computed by summing the feature importance for each individual nPrint in the 21 packet nPrint created for each device. 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.*

A.3.2 Passive OS Detection Heatmap. Figure 8 shows a heatmap of the feature importance gathered from the highest performing model trained on nPrint. 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 [7, 33]. In the TCP header, the window size field is the most important feature. 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.



(a) *IPv4*



(b) *TCP*

Figure 8: *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.*

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

A.4 Nmap Tests

```
  -4, --ipv4                 include ipv4 headers
  -6, --ipv6                 include ipv6 headers
  -A, --absolute_timestamps  include absolute timestmap field
  -c, --count=INTEGER        number of packets to parse (if not all)
  -C, --csv_file=FILE        csv (hex packets) infile
  -d, --device=STRING        device to capture from if live capture
  -e, --eth                  include eth headers
  -f, --filter=STRING        filter for libpcap
  -F, --fill_int=INT8_T      integer to fill missing bits with
  -h, --nprint_filter_help   print regex possibilities
  -i, --icmp                 include icmp headers
  -N, --nPrint_file=FILE     nPrint infile
  -O, --write_index=INTEGER  Output file Index (first column) Options:

                             0: source IP (default)

                             1: destination IP

                             2: source port

                             3: destination port

                             4: flow (5-tuple)
  -p, --payload=PAYLOAD_SIZE include n bytes of payload
  -P, --pcap_file=FILE       pcap infile
  -R, --relative_timestamps  include relative timestamp field
  -S, --stats                print stats about packets processed when finished
  -t, --tcp                  include tcp headers
  -u, --udp                  include udp headers
  -V, --verbose              print human readable packets with nPrints
  -W, --write_file=FILE      file for output, else stdout
  -x, --nprint_filter=STRING regex to filter bits out of nPrint. nprint -h for
                             details
  -?, --help                 Give this help list
      --usage                Give a short usage message
      --version              Print program version
```

A.5 nPrint Configuration Options

Here we list the current nPrint configuration options.