Beyond Accuracy: Cost-Aware Data Representation Exploration for Network Traffic Model Performance

FRANCESCO BRONZINO*, Université Savoie Mont Blanc
PAUL SCHMITT*, Princeton University
SARA AYOUBI, Nokia Bell Labs
HYOJOON KIM, Princeton University
RENATA TEIXEIRA, Inria
NICK FEAMSTER, University of Chicago

In this paper, we explore how different representations of network traffic affect the performance of machine learning models for a range of network management tasks, including application performance diagnosis and attack detection. We study the relationship between the systems-level costs of different representations of network traffic to the ultimate target performance metric—*e.g.*, accuracy—of the models trained from these representations. We demonstrate the benefit of exploring a range of representations of network traffic and present Network Microscope, a proof-of-concept reference implementation that both monitors network traffic at high speed and transforms the traffic in real time to produce a variety of representations for input to machine learning models. Systems like Network Microscope can ultimately help network operators better explore the design space of data representation for learning, balancing systems costs related to feature extraction and model training against resulting model performance.

1 INTRODUCTION

Network management tasks commonly rely on the ability to classify traffic by type or identify important events of interest from measured network traffic. Over the past 15 years, machine learning models have become increasingly integral to these tasks [3, 28, 34]. Ultimately, a model's performance depends as much on the data representation of its input as it does on the choice of model and model parameters. How data is represented also affects the cost and complexity of both capturing the necessary data, as well as the costs associated with training the model itself. To date, most network management tasks that rely on machine learning from network traffic have assumed the data to be fixed or given, typically because decisions about measuring, sampling, aggregating, and storing network traffic data are made *a priori*. As a result, a model might be trained with a sampled packet trace or aggregate statistics about network traffic, not necessarily because that data representation would result in an efficient model with good overall performance, but rather because the decision about data collection was made well before any modeling or prediction problems were considered. This problem arises in applications of supervised learning across the field, from performance diagnosis to security.

Existing network traffic measurement capabilities capture either flow-level statistics or perform fixed transformations on packet captures. First, flow-based monitoring collects coarse-grained statistics (*e.g.*, IPFIX/NetFlow collection infrastructure such as Kentik [18] and Deepfield [9]). These statistics are also often based on samples of the underlying traffic [12]. Conversely, packet-level monitoring aims to capture traffic for specialized monitoring applications [7] or triggered on-demand to capture some subset of traffic for further analysis [42]. Programmable network hardware offers potential opportunities to explore how different data representations can improve model performance; yet, previous work on programmable hardware and network data structures has typically focused on efficient ways to aggregate statistics [22] (*e.g.*, heavy hitter detection), rather than supporting different data representations for machine learning models. In all of these cases,

*Both authors contributed equally to this research.

decisions about data representation are made at the time of configuration or deployment, well before analysis takes place. Once network traffic data is collected and aggregated, it is difficult, if not prohibitive to retroactively explore a broader range of data representations that could potentially improve model performance.

From a modeling perspective, we would ideally start from raw packet traces and evaluate what transformations on those traces result in the best model performance. This approach quickly becomes unpractical. As raw traces produce massive amounts of data, collection often happens in controlled lab environments that solely provide a simplified vision of an operational network deployment. This simplified vision of the network environment fails to capture the existing inherent tradeoffs between system-level costs (*i.e.*, state, compute, bandwidth) and the achievable model performance. To overcome these limitations, we would like an approach that does not diminish our ability to explore which data representations are most effective for modeling but without losing track of the related costs. In this paper, we study a systematic approach to explore the relationship between different data representations for network traffic and a) the resulting model performance as well as b) their deployment costs on a set of classical supervised learning problems in networking.

We present Network Microscope, a proof-of-concept reference system implementation designed from the ground up to explore network data representations and evaluate the systems-related costs of these representations. To facilitate exploration, Network Microscope implements a processing pipeline that performs passive traffic monitoring and in-network feature transformations at high speed. The pipeline supports the capture of a variety of data representations and can be extended to define new ones. Further, Network Microscope integrates representation cost analysis via a profiling platform. The profiling enables the quantification of system costs, such as state and compute, for each transformation.

We use Network Microscope to demonstrate the value of exploring data representations for modeling for two types of supervised learning problems in networking: video quality inference and malware detection. We answer two questions:

- *How does data representation affect model performance?* (§4) We show that the ability to transform the data in different ways allows systems designers to make meaningful decisions involving systems costs and model performance. In some cases, for example, we find that state (*i.e.*, memory) requirements can be significantly reduced without affecting model performance, providing important opportunities for in-network reduction and aggregation. In other cases, we find that data reduction techniques do negatively affect model performance.
- *How does data representation affect costs?* (§5) We use Network Microscope to evaluate the cost of performing different transformations on traffic in real-time in deployed networks across three cost metrics: state, compute, and storage. Our results show that for the video quality inference models, state and storage requirements out-scale processing requirements. These results suggest that fine grained cost analysis can lead to different choices depending on different model performance requirements and network environments.

Overall, an important finding from this analysis is that *there is no universally ideal data representation* for these supervised learning problems, and thus there is a need for operators to explore a range of data representations that ultimately yield the right operating point for systems costs and model performance. This work lays the groundwork for new directions in applying machine learning to network modeling and prediction problems that consider not only the performance of the model, but also the appropriate ways to represent network traffic when training these models and the associated systems costs of these different representations.

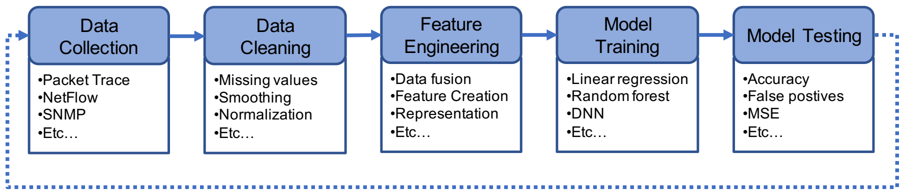


Fig. 1. *Typical pipeline for model design in network inference.*

2 THE IMPORTANCE OF DATA REPRESENTATION

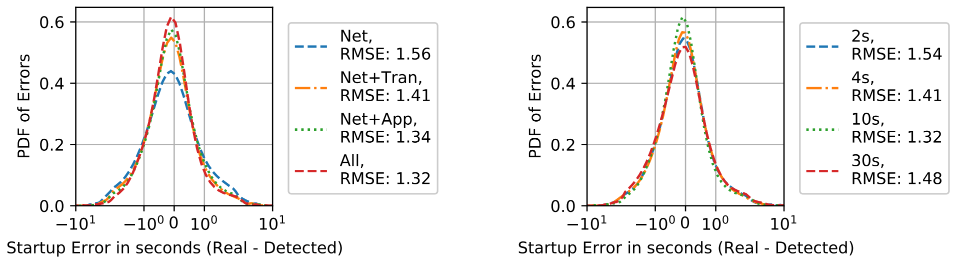
Figure 1 shows the conventional machine learning pipeline: 1) data collection; 2) data cleaning; 3) feature engineering; 4) model training; and 5) model testing. A model (step 4 in the figure) learns to predict outcomes based on some *representation* of the input data generated by the previous steps in the pipeline. In some cases, as in classical machine learning algorithms, the designer of the algorithm must decide on the data representation that will be fed to the model. For example, when designing a network-level spam or botnet detection system, the designer must start with some data (*e.g.*, a raw traffic trace, summary statistics, etc.) and compute the resulting *features* from the underlying data. The collection of features is sometimes referred to as the overall representation. In other cases, such as in representation learning (*e.g.*, autoencoders) or deep learning (*e.g.*, neural networks), the model itself learns the best representation from an existing initial representation of the data. The produced models are then evaluated (step 5 in the figure). If the obtained results are unsatisfactory, the designer is often forced to restart the pipeline cycle seeking additional or different data, a possibly onerous, if not prohibitive task.

2.1 Model Performance

The predictive performance of any machine learning algorithm, regardless of task, is impacted by the representation of the data that is offered to the model as input. Performance can be measured in a variety of ways, depending on the type of problem. In this paper, we focus in particular on supervised learning problems, where performance can be measured as prediction errors relative to some specified set of labeled data. In the case of a regression task (*e.g.*, estimating the startup delay for a streaming video), error might be represented as *mean squared error* or *mean absolute error*; in the case of a classification task (*e.g.*, detecting malware, rebuffering of a video stream), error might take the form of *prediction errors* (*e.g.*, false positives, missed detections) and their associated derivative statistics (*e.g.*, accuracy, precision, recall). Ultimately, the task of the machine learning designer is to maximize one or multiple performance metrics for the given problem.

A common approach for machine learning tasks based on network traffic involves manually designing and extracting the set of features and representations that achieve good model performance. These cases typically rely on significant domain knowledge to know the features that are most relevant to prediction, as well as how to transform those features in ways that result in good model separation. For example, in the case of device identification, a representation that is based on the first ten packets may capture a different behavior than one that is based on the first packet alone. A representation that explicitly captures packet header flags, IP addresses, or ports may also capture different aspects of traffic. Other models may rely on time-related features (*e.g.*, packet interarrival times), or data aggregation periodicity (*e.g.*, averages over a time window). Unfortunately, manual feature engineering may exclude important features that are nonobvious or have complex relationships with other features, even given domain expertise.

To exemplify how different data representations can impact model performance, we consider a common inference task: detecting video quality inference metrics — *e.g.*, startup delay — from



(a) *Impact of feature sets on startup delay.* **(b)** *Impact of time representations on startup delay.*

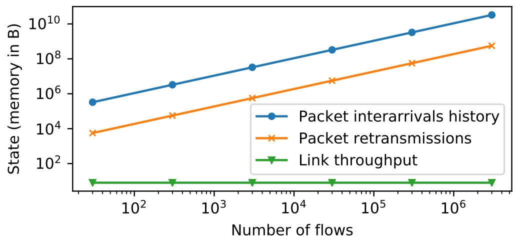
Fig. 2. *The relationship between data representations and model performance for video quality inference.*

encrypted traffic. In previous work, Bronzino et al. [4] studied how different data representations, classified as different feature sets based on network layer, impact the performance of the inference models. In particular, the work grouped features into three categories: Network Layer features as the metrics that solely rely on lightweight information available from observation of a network flow (identified by the IP/port four-tuple). Transport Layer features as those that include information extracted from the TCP header, such as end-to-end latency and packet retransmissions. Finally, Application Layer metrics as the ones that include any feature related to the application data that can be gathered by solely observing packet patterns (*i.e.*, without recurring to deep packet inspection). We replicate their work by training different models based on different networking layers for startup delay inference using the first 10 seconds of the streaming session. Figure 2a reports the distribution of inference errors confirming the original findings of the authors, *i.e.*, that transport layer features do not improve consistently the performance of the models when both network and application layer features are in use. While the initial findings are interesting, the previous work did not consider how a different representation dimension could impact model performance: the size of their time window for feature aggregation. In particular, the authors solely justify their window size choice empirically based on video streaming service behavior. To verify whether this different dimension could impact model performance, we retrain the startup delay inference models using different time window sizes (*i.e.*, two, four, 10, and 30 seconds) using the best feature set from the previous result (*i.e.*, All). The results are shown in Figure 2b. While we observe that the original choice of 10 second windows does yield the best results, we can also notice that the authors' intuition-driven choice might have significantly diminished the achievable performance of the model if they had simply chosen a different time window value. For example, using a larger window size of 30 seconds would have yield to a 12% worse performance result.

The previous example showcases the potential need of exploring a variety of different representation dimensions to maximize model performance. For many tasks; however, it can be difficult to know which features and their representations should be extracted *a priori*. This challenge poses a problem for building accurate network inference models when representation is deployed at configuration or deployment time. For example, consider an IPFIX representation, which captures aggregate statistics but not packet interarrival times. If the collection infrastructure is fixed, then exploration of the feature space that involves timing is not even possible. This limited flexibility may ultimately hinder the performance of inference models by preventing the exploration or use of features that are necessary for accurate prediction.

2.2 Model Costs

Systematizing and deploying machine learning pipelines in operational networks introduces cost constraints that are not always considered in lab settings. As a whole, deployment costs are derived



(a) *Traffic trace based analysis.*

Feature	**Memory Requirement**
Interarrivals	One timestamp per packet
Retransmissions	One TCP window size per flow
Throughput	Constant

(b) *Memory requirements for different features.*

Fig. 3. *Cost of storing in memory different traffic features based in a monitoring system.*

from the environment itself (*i.e.*, network traffic load, number of network taps, features collected) and the system resource requirements.

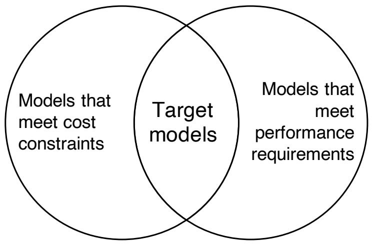
To explore the space of inference model performance and their associated costs when systematized, an operator may ultimately need to explore costs across state, processing, storage, and latency. Operationalization of a pipeline must consider not only model performance but also these systems-related costs as they may determine where and how (and whether) to deploy resources. In this paper we focus mainly on costs associated with the early stages of the machine learning (ML) pipeline: state, processing, and storage costs.

For online models, features must be stored in-memory on or near the same device that ingests network traffic until the inference model consumes them. Conversely, offline models can process features that are exported to secondary storage servers. Ultimately, feature memory requirements are based on the frequency of their generation, their representation, and their duration. Network traffic can be aggregated at multiple levels for processing. For instance, individual packets may be necessary for some models, flow-based statistics may be appropriate for others, and link-based statistics that aggregate multiple flows may be useful for another. These considerations impact the required representation and the duration that features are held.

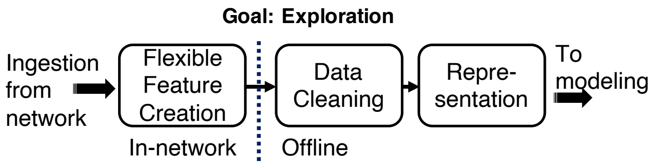
For in-network traffic monitoring purposes, state is limited by the RAM and disk space available on a server-class machine. Therefore, features must be of "reasonable" size, relative to their frequency and their representation in memory (*e.g.*, data structure design). Additionally, feature storage must compete for the limited memory with any other processes running on the system (*i.e.*, the same system may be used for feature extraction as well as inference for online models) as well as the operating system.

Figure 3a shows a basic example of the relationship of state cost to network traffic. For the purpose of this example, we use a packet trace collected in a lab network containing traffic generated by a single laptop. To emulate increasing traffic loads, we replicate the same trace multiple times. We see that features at the link level, such as throughput, require a fixed amount of memory even as the number of observed flows increase as link throughput can be represented by two counters in memory (bytes received and number of packets)[1]. Conversely, flow-based statistics like retransmission and interarrival history require memory that is proportional to either the number of packets or the number of flows that are observed. Interarrivals result in one timestamp entry per each packet received, and retransmissions require storing entries for a full TCP window's worth of transmitted packets, hence scaling storage requirements based on the number of flows. While this basic example shows the amount of memory necessary for the traffic is relatively small (*e.g.*, 10 GB maximum), required state increases as traffic increases. Monitoring a busy peering link, for example, would require substantially more memory.

[1]For this example, we assume that all counters require 4B of storage.


Models that meet cost constraints
Target models
Models that meet performance requirements

(a) *Models space split by target metrics.*



(b) *ML pipeline split to enable performance and cost exploration.*

Fig. 4. *Exploring different data representations early in the ML pipeline is critical to optimizing for both cost and model performance.*

Ultimately, the costs that arise during systematization can impact the feasibility of collecting the traffic features that are used by a model. Features that are collected by existing systems might be just a small subset of all possible features. At the other extreme, all possible features that could be extracted from network traffic may not be feasible in terms of cost given existing system constraints. Finding the features that can be collected in-network without prohibitive costs in order to increase the performance of subsequent inference models is key to design models that can be deployed in operational settings. For any given inference task, this set of features may be unique.

2.3 The Case for Exploring Representation

The pipeline presented in Figure 1 applies to many contexts where the data is "found"—that is, when a dataset exists and the model designer has no mechanism for changing the initial data that may be collected (and thus derivative representations from that data). Note that this is often the common case for many machine learning problems, where data may be trained on an existing or benchmark dataset (*e.g.*, a dataset of images, text, or data that has been previously collected).

At least in many common machine learning domains, however, the input data originates in some form that is close to "raw." For example, in the case of images, it is possible to represent the data as a matrix of pixels [20]; in the case of text, the data can be represented as a bag of words [17]. In the case of network traffic, it has often been the case that decisions about aspects of representation have been made *a priori*, without consideration of modeling. A classic example of this is aggregation, whereby the system exports data in the form of aggregate statistics (*e.g.*, IPFIX), a format from which the original raw form could never be recovered. Typically these decisions have been made based on systems-oriented constraints, without regard to what the best representation of the raw data (*i.e.*, packet traces) might be for a modeling problem.

Model designers need to iterate on different models and the features that are used as input, exploring a wide range of representations which could potentially be collected but might not be available using existing systems. As shown in Figure 4, a failure to explore the full range of possible data representations is a missed opportunity to maximize the final model performance within the constraints of what is feasible with current network technologies. First, exploring different data representations is a critical part of model design; second, jointly exploring both cost and performance considerations can result in improvements to *both* metrics. Ideally, one would *always* start with raw packet captures, from which *any* data representation could subsequently be derived. Unfortunately, doing so may impose prohibitive state, storage, bandwidth, or latency costs, and so these tradeoffs must ultimately be considered in conjunction with model performance. Doing so, however, first requires the design and implementation of a system that allows for this type of flexible representation. Solving this problem is the topic of the next section.

3 EXPLORING DATA REPRESENTATIONS WITH NETWORK MICROSCOPE

To explore network traffic feature representation and its subsequent effect on both the cost and performance of prediction models, we need a way to easily collect different representations from network traffic. While versatile, existing monitoring tools do not offer enough means to flexibly explore the impact that features and representations have on model performance and cost. As such, we implement Network Microscope, a proof-of-concept system that we have designed and implemented from the ground up to generate flexible feature representations from raw network traffic.

3.1 Goals

We design Network Microscope to provide the ability to explore network data representations and evaluate the systems-related costs of these representations. The design aims to achieve three goals.

Enable flexible and extensible feature extraction and data representation. Different network inference tasks use different models, each of which may depend on a unique set of features. For example, a model to infer video streaming performance may rely on inter-arrival times of video segments; on the other hand, a model to infer whether a link's performance is degrading may depend on measuring the number of packet retransmissions. Of course, all of these statistics are available from packet traces, but Network Microscope must compute statistics of this nature online to avoid storing complete packet captures for subsequent post-processing. Further, due to the range of models and constantly evolving applications, Network Microscope must enable customizable data representations, including making it easy to control which features are collected from the network traffic, and the time granularities at which these are collected. This requirement implies the need for extensible data collection routines that can evolve with Internet applications and the set of inference tasks. Ultimately, the system allow for customization of: (1) what data is collected; (2) for which traffic; and (3) at what time granularity. To facilitate exploration, the tool must provide a simple, extensible interface to make it easy to define new representations and features.

Integrate cost analysis. Any features that are collected will result in systems-related costs. Ideally, a system should provide a way to quantify the cost that each feature imposes, to allow a better understanding and formalization of how different data representations ultimately affect the deployability of a model in operational networks. To achieve this goal, Network Microscope must allow the user to profile each feature in isolation with fine granularity, providing a way to directly compare their effects on the collection process. We aim to enable this analysis in operational settings where such a system may ultimately be deployed. We describe how Network Microscope can be used to explore how different representations result in different costs in Section 4.

Support fast network speeds. As network speeds increase, passive traffic monitoring becomes more challenging simply due to the increasing volume and rate of traffic that must be analyzed. Even at the network edge, gigabit-level speeds in access networks can make passive traffic capture challenging. Although packet capturing at line rate in software has become more and more feasible due to tools such as PF_RING [10] and DPDK [11], the output data is often overwhelming in terms of storage: For example, a one-minute packet capture of a saturated 1 Gbps link generates 7.5 gigabytes of raw data. Thus, in addition to exploiting the available technical capabilities to *capture* traffic at high rates, Network Microscope must implement techniques to maximize its ability to ingest traffic and transform the data into features, lowering the storage burden compared to raw traffic representations. For example, the system has to efficiently select only subsets of traffic that are targeted by the inference problem being studied without resorting to sampling, which can negatively impact the inference model.

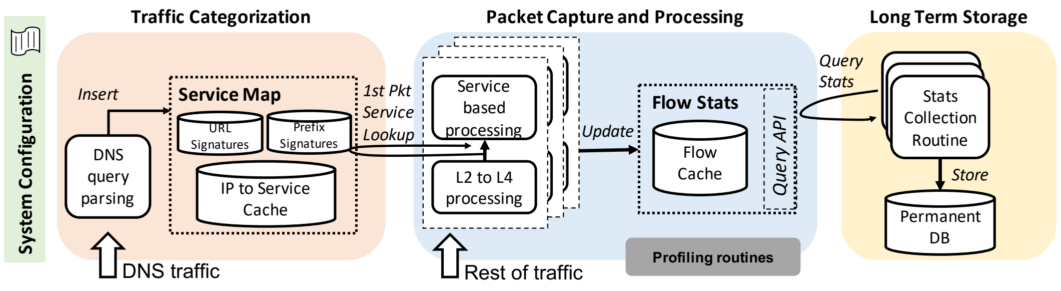


Fig. 5. *Network Microscope system overview.*

3.2 System Design and Implementation

Figure 5 shows the system design of Network Microscope. Network Microscope is implemented in Go [14] to exploit the language match of performance and flexibility, as well as its built-in benchmarking tools. The system implements three main components to support flexible collection of features from network traffic:

(1) a *traffic categorization* module responsible for associating network traffic with applications (Section 3.2.2);
(2) a *packet capture and processing* module that collects network flow statistics and tracks their state at line rate; moreover, this block implements a cache used to store flow state information (Section 3.2.3); and
(3) a *long term storage* module that queries the flow cache to obtain features and statistics about each traffic flow and stores higher-level features concerning the applications of interest for later processing (Section 3.2.4).

To support the extensible monitoring functions of the system, Network Microscope wraps a convenient configuration package that allows users to define service classes of interest as well as choose the features to extract for each one of them (Section 3.2.1). Finally, Network Microscope provides the capability to profile the feature costs through a set of profiling tools (explored in Section 5). We will open source the entire code base to the community upon publication.

3.2.1 System Configuration

We design Network Microscope to be customizable through a configuration file written in JSON format. The configuration provides a way to tune system parameters such as the interface used for capture and the definitions of service classes to monitor. A service class includes three pieces of information: (1) which flows to monitor; (2) how to represent the underlying flows in terms of features; (3) at what time granularity features should be represented. Listing 1 shows a truncated example of the configuration used to collect Netflix traffic as needed by the video inference models presented later in Section 4. Using this configuration, the system creates a customized pipeline for each targeted service class, as shown in Figure 6.

3.2.2 Signature-based traffic categorization

Network Microscope aims to minimize the processing and state for packets and flows that are irrelevant for computing the features of interest. Accordingly, it is crucial to categorize network flows based on their service so that the packet processing module, presented in the next section, can extract features solely from relevant flows, ideally without resorting to sampling traffic. Further, we seek to accurately label flows without breaking encryption or exporting private information to a remote server. To do so, Network Microscope maintains a cache to map remote IP addresses

```
{
  "Name": "Netflix",
  "Filter": {
    "DomainsString": ["nflxvideo.net", ...],
    "Prefixes": ["23.246.0.0/18", ...]
  },
  "Collect": [PacketCounters, VideoSegments],
  "Emit": 10
}
```

Listing 1. *Configuration example.*

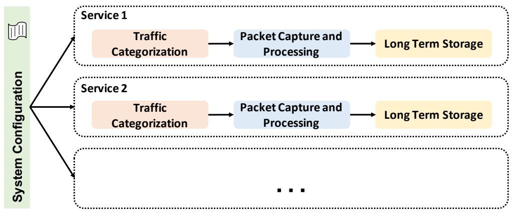


Fig. 6. *Logical split across service-driven pipelines.*

to services accessed by users. A separate process maps flows to services by tracking DNS queries and responses. When Network Microscope captures DNS packets, it inspects the hostname in DNS queries and matches these lookups against a corpus of regular expressions for domain names that we have derived for those corresponding services. For example, `(.+?\.)?nflxvideo\.net` captures a set of domain names corresponding to Netflix's video caches. For further flexibility, Network Microscope supports specifying matches between services and IP prefixes, which assists with mapping when DNS lookups are cached or encrypted.

In addition to possible performance concerns, relying on DNS to map traffic to applications and services may prove challenging in the future, as DNS becomes increasingly transmitted over encrypted transport (*e.g.*, DNS-over-HTTPS [2] or DNS-over TLS [31]). In such situations, we envision Network Microscope relying on two possible solutions: (1) the system could parse TLS handshakes for the server name indication (SNI) field in client hello messages, as this information is available in plaintext; or (2) the system could implement a web crawler to automatically generate an IP-to-service mapping, a technique already implemented in production solutions [9].

As with any signature-based matching method, an ongoing challenge is to keep the list of signatures complete and up-to-date. It is possible to semi-automatically generate signatures by capturing traffic of the target service in controlled settings to identify the set of domain names and IP prefixes to monitor. Additionally, many publicly available domain lists map domains to applications to allow network administrators to implement blocking and filtering in corporate networks. For our prototype, we created a database of signatures of a number of known over-the-top video services, video conference applications, and ads providers.

3.2.3 Packet Processing

The traffic categorization and packet processing modules both require access to network traffic. To support fast (and increasing) access network speeds, Network Microscope relies on state-of-the-art packet capture libraries: We implement Network Microscope's first two modules and integrate a packet capture interface based on PF_RING [10] and the gopacket *DecodingLayerParser* library [15]. Network Microscope also supports *libpcap*-based packet capture and replay of recorded traces.

Processing network traffic in software is more achievable than it has been in the past; yet, coupling passive network performance measurement involves developing new efficient algorithms and processes for traffic collection and analysis. Network Microscope implements parallel traffic processing through a pool of worker processes, allowing the system to scale capacity and take advantage of multicore CPU architectures. We exploit flow clustering (in software or hardware depending on the available resources) to guarantee that packets belonging to the same flow are delivered to the same worker process, thus minimizing cross-core communication. The workers store the computed state in a shared, partitioned flow cache, making it available for quick updates upon receiving new packets.

Group	Features
PacketCounters	throughput, packet counts
PacketTimes	packet interarrivals
TCPCounters	flag counters, window size, retransmissions, etc.
LatencyCounters	latency, jitter
VideoSegments	segments list

Table 1. *Current features available in Network Microscope.*

The packet processing module has two components:

State storage: Flow cache. We implement a flow cache used to store a general data structure containing state and statistics related to a network flow. The data structure allows the system to maintain both common statistics for all flows, as well as unique features and statistics for different applications using a single interface. Furthermore, it includes, if applicable, an identifier to match the services the flow belongs to. This information permits the system to determine early in the pipeline whether a given packet requires additional processing. The current version of the system implements the cache through a horizontally partitioned hash map. The cache purges entries for flows that have been idle for a configurable amount of time. In our configuration this timeout is set to 10 minutes. The general flow data structure allows for different flow types, and differing underlying statistics, to be stored in the single cache. We evaluate the performance of the cache in Section 5.

Features extraction: Service-driven packet processing. A worker pool processes all non-DNS packets. Each worker has a dedicated capture interface to read incoming packets. As a first step, each worker parses MAC, network, and transport headers, which yields useful information such as the direction of the traffic flow, the protocols, and the addresses and ports of the traffic. The system then performs additional operations on the packet depending on the service category assigned to the packet by inspecting the flow's service identifier in the cache.

Using the information specified by the configuration file, Network Microscope creates a list of feature classes to be collected for a given flow at runtime. Upon receiving a new packet and mapping it to its service, Network Microscope loops through the list and updates the required statistics. The current version of the system provides a number of built-in default features that span multiple layers of the network stack. We group features that require maintaining the same or similar state into a single class, *e.g.*, *NetworkCounter* includes packet counters as well as throughput and packets per second. Table 1 provides an overview of the features currently supported.

To account for the need to add new features to the existing collection, Network Microscope allows the user of the system to define additional, custom features. The system uses convenient flow abstraction interfaces to allow for quick implementation of collection methods for additional statistics. The flow data structure implements a single `AddPacket` function that allows a user of the system to easily process new metrics using the pre-processed information previously parsed from the packet headers. Newly implemented features can be added as a separate file in the system code structure. Upon execution, Network Microscope uses Go's language run-time reflection to capture all available feature classes and select the required ones based on the configuration file provided. For example, Listing 2 shows the implementation of the PacketCounters class which keeps track of the number of packets and bytes for flows observed by the system.

3.2.4 Long Term Storage

Network Microscope's packet processing module exposes an API that provides access to the information in the cache. Queries can be constructed based on either an application (*e.g.*, Netflix), or

```go
// PacketCounters is a data structure to collect packet and byte
    counters
type PacketCounters struct {
  InCounter   int64
  OutCounter  int64
  InBytes     int64
  OutBytes    int64
}

// AddPacket increment the counters based on information contained in
    pkt
func (c *PacketCounters) AddPacket(pkt *network.Packet) {
  if pkt.Dir == network.TrafficIn {
    c.InCounter++
    c.InBytes += pkt.Length
  } else if pkt.Dir == network.TrafficOut {
    c.OutCounter++
    c.OutBytes += pkt.Length
  }
}
```

Listing 2. *Implementing a new counters class*

on a given device IP address. Using the time representation information provided in the configuration file, Network Microscope initializes a time-driven process that extracts the information of each service class at the given time intervals. In the current version of the system, we implement the module to periodically query the API to dump all collected statistics to a temporary file in the system. We then use a separate system service to periodically upload the collected information to a remote location, where it can be used as input to models.

3.3 Prototype and System Performance

Evaluating systems-related costs of data representation requires accurately measuring these costs. Although such costs can sometimes be simulated, these approaches often abstract details that may ultimately affect the metrics. Instead, we build a prototype implementation and profile it on live traffic. This approach not only allows us to empirically measure systems-related costs of data representation, such as the state-related costs that we explore in this paper, but it also allows us to demonstrate that the flexibility we advocate for developing network models is, in fact, achievable in practice.

To demonstrate that the system can support feature packet processing at high network speeds, we evaluate Network Microscope's ability to process traffic in real time. In particular, we examine the traffic processing capacity of Network Microscope collecting the features required to infer video quality metrics in real time [4] for 11 video services. We deploy the system on a commodity server equipped with 16 Intel Xeon CPUs running at 2.4 GHz, and 64 GB of memory running Ubuntu 18.04. The server has a 10 GbE link that receives mirrored traffic from an interconnect link carrying traffic for a consortium of universities[2]. The link routinely reaches nearly full capacity (*e.g.*, roughly 9.8 Gbps) during peak times each day during the school year. We evaluate Network Microscope on the link over several days in October, 2020. We leverage the PF_RING packet library with zero-copy enabled in order to access packets at line-rate. While we did not specifically engineer or optimize our prototype for high speed processing in production environments, this setup offers a realistic deployment representation in an ISP network where a network operator might use

[2]All captured traffic has been anonymized and sanitized to obfuscate personal information before being used. No sensitive information has been stored at any point. Our research has been approved by the university's ethics review body.

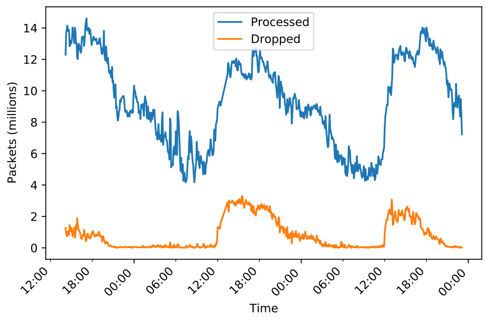


Fig. 7. *Network Microscope performance on the server.*

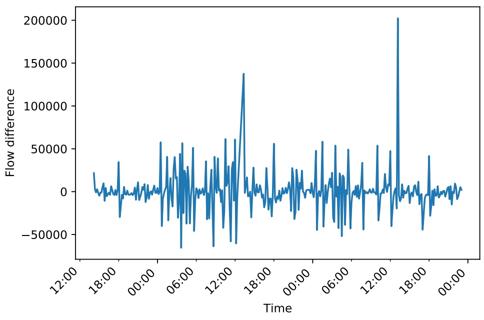


Fig. 8. *Flow cache size changes over time.*

Network Microscope to deploy its monitoring models, and enables us to better understand potential system bottlenecks.

System performance. Figure 7 shows the number of packets processed and the number of packets dropped in 10 second windows over the course of a few days. We see that traffic tends to show a sharp increase mid-day, which coincides with an increase in the rate of packet drops. Overall, Network Microscope is able to process roughly one million packets per-second (10M PPS per 10 second window in the figure) without loss, while higher rates trigger loss. We posit that this performance would likely be improved upon using a more modern server with higher specs and more CPUs, which would allow for a higher number of parallel traffic processing workers.

We further investigate the cause of packet drops in order to understand bottlenecks in the Network Microscope code. The system's flow cache is a central component that gets continuously updated concurrently by the workers that process traffic. We study the ability of the system's flow cache to update the collected entries upon the receipt of new incoming packets. We implement benchmark tests that evaluate how many update operations can the flow cache perform each second in isolation from the rest of the system. We test two different scenarios: first, we evaluate the time to create a new entry in the cache, *i.e.*, the operation performed upon the arrival of a newly seen flow. Second, we repeat the same process but for updates to existing flows in the cache. Our results show that new inserts take one order of magnitude more time than a simple update: roughly 6000 nano seconds versus 200 nano seconds. These numbers confirm that the flow cache can not support the creation of more than about 150,000 new flows per second.

We confirm this result by looking at the arrival of new flows in our deployment. Figure 8 shows the difference in the size of the flow cache between subsequent windows over the observation period. Negative values mean that the size of the flow cache decreased from one timestamp to the next. As shown, there are sudden spikes (*e.g.*, greater than 100,000 new flows) in the number of flow entries in the cache around noon on two of the days, times that correspond with increases in packet drops. Recall that the flow cache maintains a data structure for every flow (identified by the IP/port four-tuple). Therefore, the spikes are a result of Network Microscope processing a large number of previously unseen flows. This behavior helps explain the underlying causes for drops. Packets for flows that are not already in the flow cache cause multiple actions: First, Network Microscope searches the cache to check whether the flow already exists; Second, once it finds no entries, a new flow object is created and placed into the cache, which requires locks to insert an entry into the cache data structure. This leads us to believe that performance could be improved (*i.e.*, drop rates could be lowered) by using a lock-free cache data structure and optimizing for sudden spikes in the number of new flows, which would enable Network Microscope to be deployed in higher-speed production networks. We leave system optimization for future work.

4 HOW REPRESENTATION AFFECTS MODEL PERFORMANCE

In this section, we study the relationship between model performance and systems-related costs with two case studies: online video quality inference and network-based malware detection. In both cases, we build on previous work in the area, using previously developed models but explicitly exploring how *data representation* affects model performance in each case. Our goal is to better understand the relationships between how network data is represented to models and the relationships of those representations both to the performance of the model *and* to the associated systems-level costs that would need to be considered if either model were to be deployed in practice.

In this regard, our investigations in this section constitute an important re-assessment of previous results, not only replicating the results from previous work for a particular data representation, but also exploring how these models perform given different data representations. This exploration serves several purposes. At a basic scientific level, it explores the robustness of previously published results. From a practical standpoint, the results in this section also speak to practical, systems-level deployment considerations, and how those considerations might ultimately affect these models in practice. This style of evaluation serves as a significant departure from much previous work in this area and we hope can act as a rubric for evaluation of other models that rely on machine learning for prediction and inference in the future.

In this section, we focus on state-related costs as these costs are practical to study empirically with Network Microscope. Interestingly, we find that, in some cases, the relationship between state and model performance is sometimes not proportional, and that it is often possible to significantly reduce the state-related requirements of a model without significantly compromising prediction performance. We also find that depending on the problem and the performance metrics, that there may not be a single "best" data representation for network traffic, further bolstering the case for systems like Network Microscope that allow for flexible data representations.

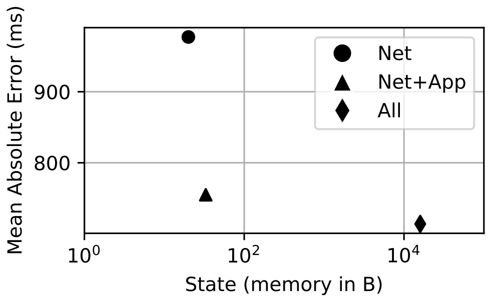
4.1 Online Video Quality Inference

With end-to-end encryption becoming more common, recent work has studied the feasibility of using machine learning models to infer streaming video quality metrics from encrypted traffic [4, 19, 25]. These approaches gather features from network traffic at recurring time intervals to then feed models that can infer a number of metrics related to user-perceived quality of experience, such as startup delay and resolution.

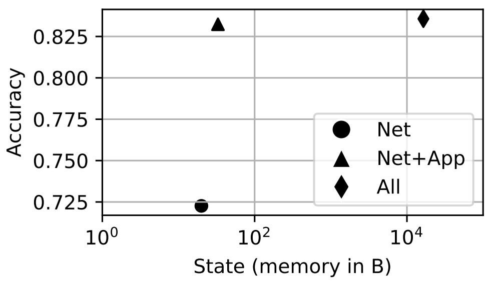
The selection of a video quality inference method requires understanding which metrics provide the best tradeoffs between collection overhead and detection accuracy. As the presented methods work online, they are required to maintain in memory the state necessary to compute the features. Some metrics can easily be computed in the network path using counters (*e.g.*, throughput), whereas others require maintaining per-flow state for the entire time interval used for the inference (*e.g.*, per packet data bytes distribution). The performance and cost are then impacted by two design choices: (a) which features are used for prediction and (b) how frequently the predictions are performed.

4.1.1 The relationship between representation and model performance

As first presented in Section 2.1, Bronzino *et al.* [4] categorized useful features for video quality inference into three categories that correspond to layers of the network stack: Network, Transport, and Application Layer features. To understand the relationship between memory overhead and inference accuracy, we use the dataset of more than 13k sessions presented in [4] to train six inference models for the two studied quality metrics: startup delay and resolution. For our analysis, we use the random forest models presented in [4]; in particular, random forest regression for startup delay and random forest multi-class classifier for resolution. Further, we use the same inference interval size, *i.e.*, 10 second time bins. To quantify the state costs, we calculate the amount of



(**a**) *Startup delay (lower is better).*



(**b**) *Resolution (higher is better).*

Fig. 9. *The relationship between features state cost and model performance for video streaming quality inference.*

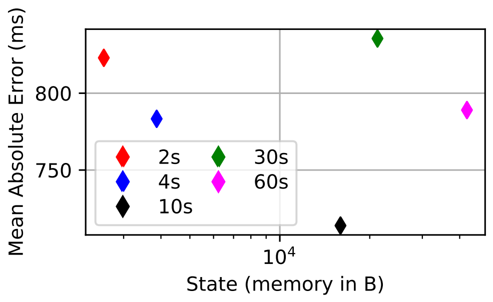
memory required to store each class of features for each time interval of the collection procedure. For the purpose of this analysis, we assume that for each counter in use, four bytes are sufficient to maintain the state without loss of precision.

Figure 9 shows the relationship between model performance and state costs. We observe that network features alone can provide a lightweight solution to infer both startup delay and resolution but this yields the lowest model performance. Surprisingly, adding application layer features contributes to a very small additional memory overhead. This is due to the fact that for the vast majority of time slots, relatively few segments are downloaded even though a high number of flows are active (not necessarily to download video). This result is particularly important for resolution where models with video segments alone perform basically as well as the others. Further, we observe that adding transport features (labeled "All" in the figure) provides very small benefits in terms of added performance — 40 ms on average lower errors for startup delay and less than 0.5% higher accuracy for resolution. Even for startup delay where using transport features can improve the mean absolute error by a bigger margin, this comes at the cost of two orders of magnitude higher memory usage. This is due to the fact that many transport-layer features require maintaining long-running per-flow state information (*e.g.*, per packet data bytes transmitted).

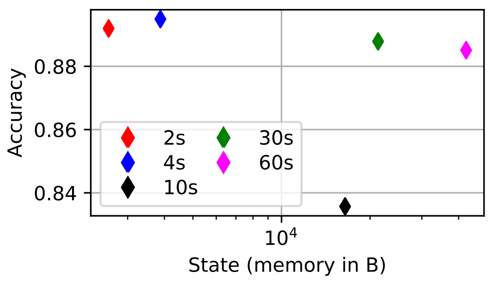
4.1.2 The relationship between time granularity and model performance

State of the art inference techniques (Bronzino *et al.* [4] and Mazhar and Shafiq [25]) employ 10 second time bins to perform the prediction of the features. This decision is justified as a good trade-off between the amount of information that can be gathered during each time slot, *e.g.*, to guarantee that there is at least one video segment download in each bin, and the granularity at which the prediction is performed. For example, small time bins—*e.g.*, two seconds— can have a very small memory requirement but might incur in lower prediction performance due to the lack of historical data on the ongoing session. On the other hand, larger time bins—*e.g.*, 60 seconds—could benefit from the added information but would provide results that are just an average representation of the ongoing session quality. These behaviors can be particularly problematic for startup delay, a metric that would benefit from using exclusively the information of the time window during which the player is retrieving data before actually starting the video reproduction.

Similarly to Section 4.1.1, we train different random forest models with increasing time bin sizes of two, four, 10, 30, and 60 seconds. To fully understand the possible memory impact of the different time bins, we use all features (All) to train the models. Figure 10 shows the obtained results. From the figure, we observe different outcomes for the two quality metrics. For startup delay, the results show that 10 second windows can indeed provide a good trade-off between memory and prediction, achieving a minimum of 70 ms better predictions than all other time granularities. This results



(a) *Startup delay (lower is better).*



(b) *Resolution (higher is better).*

Fig. 10. *The relationship between time granularity state costs and model performance for video quality inference.*

shows that 10 seconds are an acceptable trade-off between gathering enough information at the beginning of the session without gathering data from instants in time that are not relevant anymore to the prediction, *i.e.*, from data retrieval that happens after the video has started.

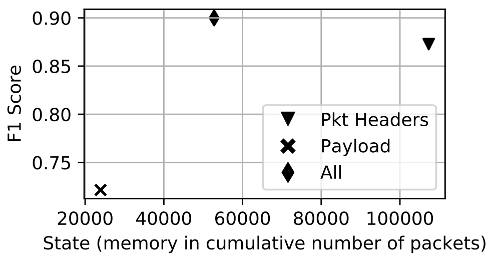
Interestingly, the results for resolution inference show that 10 second windows perform the worst among all studied cases. This might be the product of multiple factors. In particular, we observe that the change of the inference window size not only changes how much information it is used for prediction but that it also affects the granularly of the inference, possibly modifying the underlying problem. Among the different time bin sizes we have different extremes ranging from two second windows which are shorter than the video segment size itself for all services, as well as 60 seconds time windows that could contain many video quality changes within the time slot caused by the download of multiple video segments.

4.2 Malware Detection

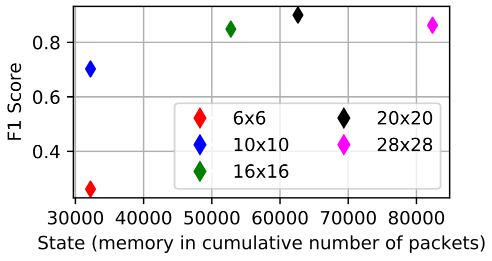
Traffic classification is of fundamental importance to a number of network management tasks, including network security. Most existing machine-learning based traffic classifiers rely on features engineered by domain experts [28]. Recently, several works on traffic classification explored the application of deep learning on raw network traffic [23, 24, 37, 38] to learn the features. The deep-learning based solutions show results that slightly outperform “shallow” machine learning models without the need for expert-crafted features. The deep-learning based solutions consist of feeding a Convolutionary Neural Network (CNN) with raw traffic data, either as a normalized sequence of bytes [23, 24] or by converting the bytes into a gray-scale image [37, 38]. These methods also differ in the choice of the length of the sequence of bytes and the layer in the network stack from which the sequence is extracted. For example, for traffic flow classification, [38] uses the first 784 bytes of each flow, whereas DeepMAL [24] uses the first 100 bytes of payload of the first two packets of each flow. A key deciding factor (that was overlooked in these works) is the impact of these different decisions on the deployment costs and model performance. Applying such traffic classifiers online requires to keep the sequence of bytes in memory until the desired length is achieved before feeding this sequence to the CNN for inference. The amount of memory used increases depending on (a) how many protocol layers are used to perform the inference (*i.e.*, how many features are used) and (b) how many packets are required to build the sequence of bytes into a gray-scale image.

4.2.1 *The relationship between features and model performance*

To understand the relationship between memory cost and inference accuracy, we use ~171k flows from the CIC-IDS2017 dataset [33] to train a CNN to classify traffic flows as either benign



(a) *Layers in the network stack (higher is better).*



(b) *Image size (higher is better).*

Fig. 11. *The relationship between state costs and model performance for CNN-based malware detection.*

or malware. The CIC-IDS2017 consists of five days worth of pcap traces that contain a mix of lab-generated benign and malware traffic. We train 15 models in total to explore the performance-cost relationship while varying the layers of the network stack used to extract the sequence of bytes: header-fields only ("Pkt Headers"), payload only ("Payload"), or header-fields and payload (All). We also explore the model cost overhead when varying the size of the sequence of bytes fed to the CNN. We follow the same approach as [38] by converting the sequence of bytes extracted into gray images before feeding them to the CNN; hence, we refer to the size of the sequence of bytes as the image size hereinafter. We perform the cost-performance analysis on a test set of 19k flows. For the cost metric, we use the number of packets needed to generate the required image which linearly relates to the amount of memory needed to maintain the required information.

Figure 11a presents the relationship between the layers in the network stack used and the inference F1 score. We obtain these results by feeding a CNN with a 20x20 image generated using packet headers only, payload only, or both. The choice of the image size is justified as the one that rendered the highest performance across all layers. Note that the resultant image size is the same regardless of the layers used, however, the number of packets needed to assimilate the image varies depending on the choice of layers. Using payload only yields the lowest cost since typically a single packet per flow contains enough bytes to generate the desired image size; but it also yields the lowest F1 score due to encryption. Packet headers yield a higher accuracy but come at a much higher cost, since more packets are needed to assimilate the image size needed from header-fields bytes. Generating the image using all layers strikes a good balance between cost and performance. That is because when the payload is not encrypted, it provides useful insights on the nature of the flow (note that the CIC-IDS2017 consists of a mix of encrypted and unencrypted traffic flows).

4.2.2 The relationship between image size and model performance

Next, we explore the relationship between image size and inference performance. Figure 11b presents the F1 score obtained as we vary the image size. Here, we show the results using all layers in the network stack. Image size has a clear impact on the model performance and the associated memory cost. The F1 score improves as the image size increases. However, this improvement flattens as the image size increases from a 20x20 to 28x28, whereas the induced cost in terms of memory cost maintained per-flow almost doubles. This result supports the need to explore the performance-cost relationship, as maintaining more information does not always lead to higher performance.

5 HOW REPRESENTATION AFFECTS COST

Given our findings in Section 4, we argue that operators should holistically consider systems-related costs *in addition to* model performance for their tasks. Network Microscope aims to provide an intuitive and automated platform to evaluate the cost effects of different data representations. To do so, we leverage Go's built-in benchmarking features and implement dedicated tools to profile different costs intrinsic to the collection process. Our profiling method allows for quickly iterating through the collection of different features in isolation and providing a fair comparison across different service classes. We find that while some features add relatively little state (*i.e.*, memory), processing, or long-term storage costs, others require substantially more resources.

5.1 Cost Profiling Method

State costs. We use Go's `pprof` profiling tool to track the amount of in-use memory. Using this tool, the system can output at a desired instant a snapshot of the entire in-use memory of the system. We extract from this snapshot the amount of memory that has been allocated by each service class at the end of each iteration of the collection cycle, *i.e.*, the time the long term storage module gathers the data from the cache. We choose this instant because it corresponds to the memory usage peak for each cycle. To avoid interference across different classes, we evaluate each class separately by extracting from each trace only the traffic generated by the service of interest and injecting it into the profiling routine. At the beginning of each profiling iteration, we pre-load in memory the entire trace to be used for the evaluation. We then iterate over all packets updating for each the corresponding flow entry in the cache. At the end of each session we collect all snapshots and create a profiling output file with the obtained results for the features under study.

Processing costs. To evaluate the CPU usage for each feature class, we aim to monitor the amount of time required to extract the feature information from each packet, leaving out any operation that shares costs across all possible classes, such as processing the packet headers or reading/writing into the cache. To do so, we build a dedicated pipeline that removes all additional operations from the execution. For each feature class of interest, the pipeline uses the preprocessed information to then execute all feature extraction operations on the packets of the trace. This method is similar in spirit to Go's built-in benchmarking feature but allows for using custom packet traces for evaluation.

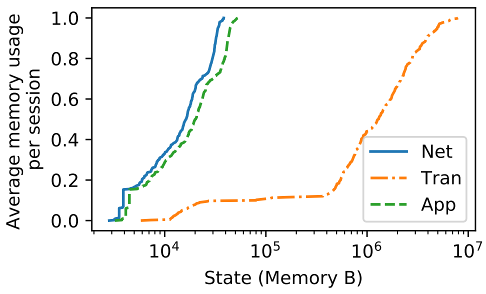
Storage costs. Storage costs can be compared by observing the size of the output generated over time during the collection process. The current version of the system stores this file in json format without implementing any optimization on the representation of the extracted information. While this solution can provide a general overview of the amount data produced by the system, we expect that this feature will be further optimized for space in the future. For reference, we provide in the following section a comparison of the amount of data generated by the current version of the system with alternative monitor tools: pcap traces and NetFlow.

5.2 Online Video Quality Inference

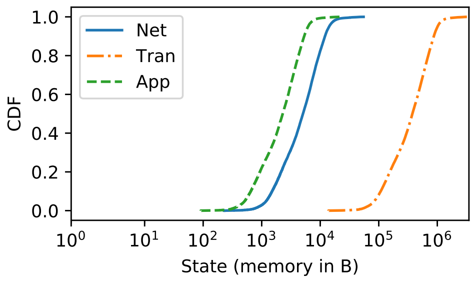
We analyze the same video inference use case presented in Section 4.1 to understand the costs of the associated representations. To do so, we use 1,000 packet traces from video sessions split across four major video services (Netflix, YouTube, Amazon Prime Video, and Twitch). We evaluate state, processing, and storage costs focusing our analysis on the three classes of features generally used for video quality inference: network, transport, and application features.

5.2.1 State Costs

We study the state costs as the amount of in-use memory required by the system at the end of each collection cycle—*i.e.*, the periodic interval at which the cache is dumped into the external storage. Figure 12a shows the memory distribution in Bytes across all analyzed sessions. The reported results highlight how collecting transport layer features can heavily impact the amount of memory used by the system. In particular, we observe that collecting such features can require up



(a) *Per streaming session.*



(b) *For streaming 10 minute intervals on a university link.*

Fig. 12. *State required for different features used for video inference models*

to three orders of magnitude more memory compared to network and application features. This result confirms our analytical study previously presented in Figure 9.

We also observe that, in contrast to what was previously observed in Section 4.1, application features have a slightly larger memory footprint than network features. Upon inspection, we realized that this is due to the basic Go data structure used to store video segments in memory, *i.e.*, a slice, which requires extra memory to implement its functionality. This additional memory was not accounted for in our initial analysis. The result further demonstrates the additional insights that can be obtained by studying costs on a running system.

State costs at scale. To provide insight on the impact of the results obtained on data representation costs at scale, we measure memory costs of an inference pipeline on a link that provides a mirror of all traffic, including traffic from student residences, at a university edge router. We implement feature collection to track all Netflix traffic on the link. The link operates at 10 Gbps and our measurements were conducted for nine days from April 30, 2020 through May 8, 2020[3].

Figure 12b shows CDFs of the memory required to monitor the video flows, with features categorized by their corresponding layer of the network stack. The memory is measured using a collection cycle of 10 seconds and a cache purge timeout of 10 minutes. We observe a median of 634 video flows and 763 video segments in each interval. As shown, features from the network layer and the application layer (*e.g.*, video segment interarrivals) require considerably less memory compared with transport layer features. This follows from the findings in Figure 12a that show the state cost of transport features compared with network features as transport features require historical flow information (*e.g.*, all packets) versus simple counters. We see that the video flow features require a median of a few hundred MB in memory. Busier networks such as ISP peering links would obviously require significantly more memory in order to maintain the necessary state for the video inference pipeline we tested.

5.2.2 Processing Costs

Collecting features on a running system measuring real traffic provides the ability to quantify the processing requirements for each target feature class. We represent the processing cost as the average processing time required to extract a feature set from a captured packet. Figure 13 shows distributions of the time required to process the features for the same three feature classes relevant to the video streaming quality inference. We observe that collecting simple network counters reduces processing time, followed by application and transport features.

[3]Note that this measurement was conducted during the COVID-19 pandemic. While a sizeable portion of the campus population was not present, many students maintained residency and were active on the network.

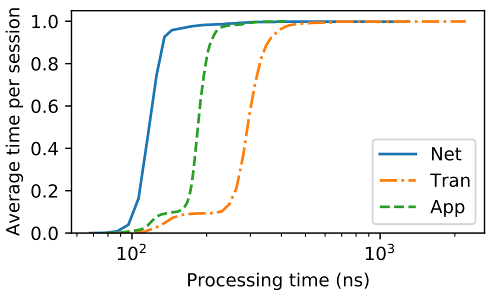


Fig. 13. *Processing costs for different representations used for video inference models.*

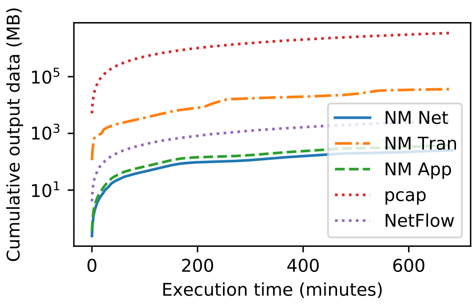


Fig. 14. *Storage costs for different representations used for video inference models.*

While there are differences among the three classes, we observe that the difference is relatively small and within the same order of magnitude. These results highlight how all feature classes considered for video inference are relatively lightweight in terms of processing requirements. Hence, we would not expect them to have major impact on the system's ability to collect them in real time from inside the network. Further, this conclusion demonstrates that, for this particular service class, state costs have a much larger impact on the ability of collecting them in an operational network.

5.2.3 Storage Costs

Feature retrieval at scale can also generate high costs due to the need to move the collected data out of the measurement system and to the location where it will be ingested for processing. Figure 14 shows the amount of data produced by Network Microscope when collecting data for the three feature classes relevant to the video streaming quality inference. This data is produced by the execution on the interconnect mirrored link used in Section 3.3. For comparison, we also include the same information for two different approaches to feature collection: a) pcap, which collects an entire raw packet trace; b) NetFlow (configured using defaults), which collects aggregated per flow data volume statistics; this roughly corresponds to the same type of information collected by Network Microscope for the network layer features.

From the results we observe that storage costs follow similar trends than the state ones presented in the previous section. This is not surprising as the exported information is a representation of the state contained in memory. More interesting outcomes can be observed by comparing our system output to existing systems. We notice that, even if not optimized, our current implementation produces less data than NetFlow, even when exporting similar information, *i.e.*, network features. While this result mostly reflects the different verbosity levels of the configurations used for each system, it confirms our Section 4 outcome: having additional flexibility in exporting additional features, *i.e.*, video segments information, can have great performance benefits at a low additional cost. Finally, as expected, we observe that raw packet traces generate a possibly unbearable amount of data and if used at scale can quickly generate terabytes of data. This result supports our claim that collecting traces at scale quickly becomes unpractical.

6 RELATED WORK

Over the past few years, machine learning models have become an increasingly integral component in solving a number of network management tasks, spanning from performance inference to detection of networks events and more. Our work takes inspiration from the large corpus of previous work on supervised machine learning methods applied to networking [3, 28, 34] and analyzes the need for better understanding the relationship that different data representations used as input have on the final accuracy of the models as well as on the costs of collecting the data.

Constructing features requires monitoring network traffic. Passive traffic measurement is a well-established area. Tools such as NetFlow [5], IPFIX [6], and SNMP allow for capturing summaries of traffic flows, statistics, or events. Yet, flow statistics are often too coarse to exhaustively support the development of effective models; data manipulation performed by the collector itself results in few knobs to turn, providing little or no flexibility for feature creation and data representation by a user. Such constraints make it hard to experiment with different representations that can improve inference models or reduce system-level costs.

On the other hand, packet-level measurements such as libpcap [35] are too high-volume to collect. Tshark [36], the terminal-based Wireshark [29], which builds on top of libpcap and is used for creating data formats that are fit for various tasks, including ML-based inference, suffers from similar limitations. Such capture is challenging to scale up to handle high-speed traffic. Tshark also lacks the ability to aggregate traffic based on time windows or bins. Our approach based on Network Microscope uses caches, modules, and parallel programming techniques to effectively handle high-speed traffic input. Further, it also provides modules that help identify specific traffic classes or services (*e.g.*, video segments) automatically for the user.

Advanced network monitoring and analysis tools such as Tstat [13, 26], Bro [30], and Snort [32] share a similar spirit to Network Microscope in that they have the goal of capturing network traffic and executing transformations on the data for later use. Tstat is an open source passive monitoring tool that can monitor network traffic and output logs, statistics, and histograms with different granularities: per-packet, per-flow, or aggregated. Bro and Snort are network intrusion detection systems that rely on regular expressions to identify the subset of packets to inspect and execute specific tasks based on the class of traffic. Ultimately, while these tools are versatile traffic monitoring tools they would need to be adapted to achieve the goals that our work aims to achieve, *i.e.*, flexible exploration of pre-existing or new features, which Network Microscope provides by default. Additionally, Network Microscope has been designed from the ground up for the purpose of providing researchers with the ability to not only explore the network traffic feature space across a range of different representations for any given problem, but to also integrate the ability to evaluate the related costs for such collection.

Finally, streaming analytics platforms [1, 8, 16, 27, 41] and algorithms (*e.g.*, "Sketches") [21, 22, 39, 40] provide a line of work complementary to ours. Streaming platforms allow operators to express queries on streaming traffic data. But they are primarily designed to collect low-level statistics on a backbone router or switch, or a programmable data-center switch, which operate at very high speeds. As such, they typically support a more limited set of queries that are constrained by the hardware they are designed to support. Similarly, sketches are custom data structures that support specific counting and aggregation tasks on traffic streams at high speeds. Findings from the analysis provided by our approach could be used to export compatible collection tasks to programmable network elements in high speed networks.

7 CONCLUSION

The performance of any machine learning model often depends as much on input data representation as it does on the model itself. However, existing network traffic data representations are typically determined by decisions made at the time of deployment without regard for future use in ML pipelines. In this paper, we create Network Microscope to facilitate the exploration of multiple possible data representations. Network Microscope enables the study of the relationship between network data representations and their subsequent effects on both the cost and the inference model performance on live network traffic. We show that such exploration is worthwhile and fruitful, finding that: (1) there is typically no single best representation of network traffic data that yields the best performance for all models and network prediction tasks, and (2) in some cases, there exists a data representation that both improves model performance and reduces the state or processing costs associated with producing that representation.

These results point to many promising future directions concerning the design of network measurement systems that allow the machine learning modeling process to more easily explore a range of data representations for any given modeling problem. Furthermore, our work paves the way for holistic approaches to model evaluation that include not only model performance but also systems-related costs. While we have considered the state, processing and state costs of different data representations, the approach we introduce can and should be extended to consider a broader range of costs, including other systems-related costs (*e.g.*, latency), as well as model training time and complexity.

REFERENCES

[1] Kevin Borders, Jonathan Springer, and Matthew Burnside. 2012. Chimera: A Declarative Language for Streaming Network Traffic Analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. USENIX, Bellevue, WA, 365–379. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/borders

[2] Kevin Borgolte, Tithi Chattopadhyay, Nick Feamster, Mihir Kshirsagar, Jordan Holland, Austin Hounsel, and Paul Schmitt. 2019. How DNS over HTTPS is Reshaping Privacy, Performance, and Policy in the Internet Ecosystem. *Performance, and Policy in the Internet Ecosystem (July 27, 2019)* (2019).

[3] Raouf Boutaba, Mohammad A Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M Caicedo. 2018. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications* 9, 1 (2018), 16.

[4] 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.

[5] Benoit Claise. 2004. *Cisco systems netflow services export version 9*. Technical Report.

[6] Benoit Claise, Brian Trammell, and Paul Aitken. 2013. Specification of the IP flow information export (IPFIX) protocol for the exchange of flow information. RFC 7011.

[7] corelight 2019. Corelight. https://corelight.com/.

[8] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. 2003. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 647–651.

[9] deepfield 2019. Deepfield. https://deepfield.com/.

[10] Luca Deri et al. 2004. Improving passive packet capture: Beyond device polling. In *Proceedings of SANE*, Vol. 2004. Amsterdam, Netherlands, 85–93.

[11] dpdk 2018. DPDK, Data Plane Development Kit. https://www.dpdk.org/.

[12] Cristian Estan and George Varghese. 2002. New Directions in Traffic Measurement and Accounting. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Pittsburgh, Pennsylvania, USA) *(SIGCOMM '02)*. ACM, New York, NY, USA, 323–336. https://doi.org/10.1145/633025.633056

[13] Alessandro Finamore, Marco Mellia, Michela Meo, Maurizio M Munafò, and Dario Rossi. 2010. Live traffic monitoring with tstat: Capabilities and experiences. In *International Conference on Wired/Wireless Internet Communications*. Springer, 290–301.

[14] golang 2020. Go language. https://golang.org/.

[15] gopacket 2020. Go Packet Library. https://godoc.org/github.com/google/gopacket.

[16] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven Streaming Network Telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. ACM, New York, NY, USA, 357–371. https://doi.org/10.1145/3230543.3230555

[17] Thorsten Joachims. 1998. Text categorization with support vector machines: Learning with many relevant features. In *European conference on machine learning*. Springer, 137–142.

[18] kentik 2019. Kentik. https://kentik.com/.

[19] Vengatanathan Krishnamoorthi, Niklas Carlsson, Emir Halepovic, and Eric Petajan. 2017. BUFFEST: Predicting Buffer Conditions and Real-time Requirements of HTTP (S) Adaptive Streaming Clients. In *Proceedings of the 8th ACM on Multimedia Systems Conference*. ACM, 76–87.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[21] Abhishek Kumar, Minho Sung, Jun Jim Xu, and Jia Wang. 2004. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 32. ACM, 177–188.

[22] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 101–114.

[23] Gonzalo Marín, Pedro Casas, and Germán Capdehourat. 2018. Rawpower: Deep learning based anomaly detection from raw network traffic measurements. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. 75–77.

[24] Gonzalo Marín, Pedro Casas, and Germán Capdehourat. 2020. DeepMAL–Deep Learning Models for Malware Traffic Detection and Classification. *arXiv preprint arXiv:2003.04079* (2020).

[25] M. Hammad Mazhar and Zubair Shafiq. 2018. Real-time Video Quality of Experience Monitoring for HTTPS and QUIC. In *INFOCOM, 2018 Proceedings IEEE*. IEEE.

[26] Marco Mellia, Andrea Carpani, and Renato Lo Cigno. 2003. Tstat: TCP statistic and analysis tool. In *International Workshop on Quality of Service in Multiservice IP Networks*. Springer, 145–157.

[27] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 85–98.

[28] Thuy TT Nguyen and Grenville Armitage. 2008. A survey of techniques for internet traffic classification using machine learning. *IEEE communications surveys & tutorials* 10, 4 (2008), 56–76.

[29] Angela Orebaugh, Gilbert Ramirez, and Jay Beale. 2006. *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier.

[30] Vern Paxson. 1999. Bro: a system for detecting network intruders in real-time. *Computer networks* 31, 23-24 (1999), 2435–2463.

[31] Tirumaleswar Reddy, Dan Wing, and Prashanth Patil. 2017. Dns over datagram transport layer security (dtls). *RFC 8094* (2017).

[32] Martin Roesch et al. 1999. Snort: Lightweight intrusion detection for networks.. In *Lisa*, Vol. 99. 229–238.

[33] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. 2018. Toward generating a new intrusion detection dataset and intrusion traffic characterization.. In *ICISSP*. 108–116.

[34] Jayveer Singh and Manisha J Nene. 2013. A survey on machine learning techniques for intrusion detection systems. *International Journal of Advanced Research in Computer and Communication Engineering* 2, 11 (2013), 4349–4355.

[35] tcpdump 2020. tcpdump and libpcap. https://www.tcpdump.org/.

[36] tshark 2020. Tshark: terminal-based Wireshark. https://www.wireshark.org/docs/wsug_html_chunked/AppToolstshark.html.

[37] 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.

[38] Wei Wang, Ming Zhu, Xuewen Zeng, Xiaozhou Ye, and Yiqiang Sheng. 2017. Malware traffic classification using convolutional neural network for representation learning. In *2017 International Conference on Information Networking (ICOIN)*. IEEE, 712–717.

[39] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 561–575.

[40] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 29–42.

[41] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. 2017. Quantitative Network Monitoring with NetQRE. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. ACM, New York, NY, USA, 99–112. https://doi.org/10.1145/3098822.3098830

[42] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. 2015. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 479–491.