Traffic Refinery: Cost-Aware Traffic Representation for Machine Learning in Networks

Francesco Bronzino[1*], Paul Schmitt[2*], Sara Ayoubi[3],
Hyojoon Kim[2], Renata Teixeira[4], Nick Feamster[5]

[1]Université Savoie Mont Blanc [2]Princeton University
[3]Nokia Bell Labs [4]Inria [5]University of Chicago

arXiv:2010.14605v2 [cs.NI] 28 Jan 2021

ABSTRACT

Ever more frequently network management tasks apply machine learning on network traffic. Both the accuracy of a machine learning model and its effectiveness in practice ultimately depend on the representation of raw network traffic as features. Often, the representation of the traffic is as important as the choice of the model itself; furthermore, the features that the model relies on will ultimately determine where (and even whether) the model can be deployed in practice. This paper develops a new framework and system that enables a joint evaluation of both the conventional notions of machine learning performance (*e.g.*, model accuracy) and the systems-level costs of different representations of network traffic. We highlight these two dimensions for a practical network management task, video streaming quality inference, and show that the appropriate operating point for these two dimensions depends on the deployment scenario. We demonstrate the benefit of exploring a range of representations of network traffic and present Traffic Refinery, a proof-of-concept reference implementation that both monitors network traffic at 10 Gbps and transforms the traffic in real time to produce a variety of feature representations for machine learning models. Traffic Refinery both highlights this design space and makes it possible for network operators to easily explore different representations for learning, balancing systems costs related to feature extraction and model training against the 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, 35]. To build a machine learning model based on network traffic, a common approach requires designing and extracting a set of features that achieve good model performance. This process typically requires 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 separation of classes in the underlying dataset. Figure 1 shows a typical pipeline, from measurement to modeling: The process begins with data (*e.g.*, a raw traffic trace, summary statistics produced by a measurement system); features are then derived from this underlying data. The collection of features and their computed statistics is often referred to as the data *representation* that is used as input to the model. Even for cases where the model itself learns the best representation based on its input (*e.g.*, representation learning or deep learning), the designer of the algorithm must determine the *initial* representation of the data that is provided to model. In practice, a model's effectiveness 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 affects the performance of the model as well as the cost and complexity of capturing the necessary data.

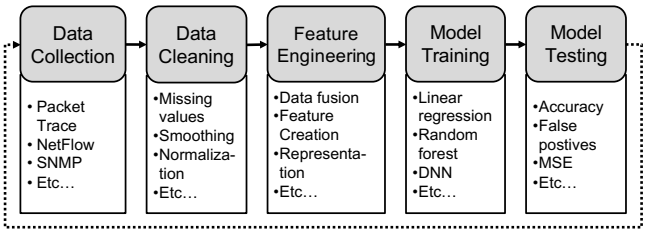


Figure 1: *Typical pipeline for model design in network inference.*

Unfortunately, with existing network traffic measurement systems, the first three steps of this process—collection, cleaning, and feature engineering—are often out of the pipeline designer's control. 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

*Co-first authors.

about data collection was made well before any modeling or prediction problems were considered.

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 or collection infrastructure such as Kentik [19] and Deepfield [10]). These statistics are also often based on samples of the underlying traffic [13]. Conversely, packet-level monitoring aims to capture traffic for specialized monitoring applications [8] 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 [21] (*e.g.*, heavy hitter detection), rather than supporting different data representations for machine learning models. In all of these cases, 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 impossible, to retroactively explore a broader range of data representations that could potentially improve model performance.

A central premise of the work in this paper is introducing additional flexibility into the first three steps of this pipeline for network management tasks. On the surface, raw packet traces would seem to be an appealing starting point: Any networking operator or researcher knows full well that raw packet traces offer maximum flexibility to explore transformations and representations that result in the best model performance. Yet, unfortunately, capturing raw packet traces often proves to be impractical on large networks as raw packet traces produce massive amounts of data, introducing storage and bandwidth requirements that are often prohibitive. Many controlled laboratory experiments (and much past work) that demonstrate a model's accuracy turn out to be non-viable in practice because the systems costs of deploying and maintaining the model are prohibitive. An operator may ultimately need to explore costs across state, processing, storage, and latency to understand whether a given pipeline can work in its network.

Evaluation of a machine learning model for network management tasks must also consider the operational costs of deploying that model in practice. Such an evaluation requires exploring not only how data representation and models affect model accuracy, but also the systems costs associated with different representations. Sculley *et al.* refer to these considerations as "technical debt" [34] and identified a number of hidden costs that contribute to building the technical debt of ML-systems, such as: unstable sources of data, underutilized data, use of generic packages, among others. This problem is vast and complex, and this paper does not explore all dimensions of this problem. For example, we do not investigate practical considerations such as model training time, model drift, the energy cost of training, model size, and many other practical considerations. In this regard, this paper scratches the surface of systemization costs, which we believe deserves more consideration before machine learning can be more widely deployed in operational networks.

As an initial step in this direction, we develop and publicly release a systematic approach to explore the relationship between different data representations for network traffic and (1) the resulting model performance as well as (2) their associated costs. We present Traffic Refinery (§3), a proof-of-concept reference system implementation designed to explore network data representations and evaluate the systems-related costs of these representations. To facilitate exploration, Traffic Refinery implements a processing pipeline that performs passive traffic monitoring and in-network feature transformations at traffic rates of up to 10 Gbps in software (§4). The pipeline supports capture and real-time transformation into a variety of common feature representations for network traffic; we have designed and exposed an API so that Traffic Refinery can be extended to define new representations, as well. In addition to facilitating the transformation itself, Traffic Refinery performs profiling that quantifies system costs, such as state and compute, for each transformation, to allow researchers and operators to evaluate not only the accuracy of a given model but the associated systems costs of the resulting representation.

As a proof-of-concept, we use Traffic Refinery to demonstrate the value of jointly exploring data representations for modeling and their associated costs for a supervised learning problem in networking: video quality inference from encrypted traffic. We study two questions:

- *How does the cost of feature representation vary with network speeds?* (§5.2) We use Traffic Refinery 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-pace processing requirements as traffic rates increase. These results suggest that fine-grained cost analysis can lead to different choices for traffic representation depending on different model performance requirements and network environments.
- *Can systems costs be reduced without affecting model accuracy?* (§5.3) 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. 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.

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.

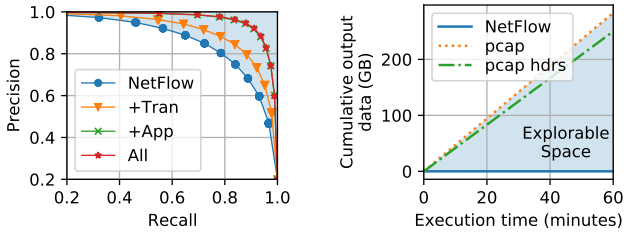
2 COST-AWARE REPRESENTATION EXPLORATION

Model designers typically need to iterate on different models and the representation of the data that is provided to these models. Exploring the wide range of possible data representations can help improve model performance within the constraints of what is feasible with current network technologies. Doing so, however, requires a system that has been designed with the goal of *joint exploration* of both cost and model performance in mind—a main contribution of this paper. With this goal in mind, we highlight what requirements a system must meet to support this joint exploration.

2.1 Flexible Feature Extraction

Different network inference tasks use different models, each of which may depend on a unique set of features. A model designed to infer whether a link's performance is degrading may rely on common traffic features such as measuring the number of packet retransmissions; on the other hand, a model designed to infer application quality over time might have to rely on ad hoc features.

Consider the task of inferring the quality of a video streaming application from encrypted traffic (*e.g.*, resolution). This problem is well-studied [5, 18, 22, 23]. The task has been commonly modeled using data representations extracted from different networking layers at regular intervals (*e.g.*, every ten seconds). For instance, Bronzino *et al.* [5] grouped data representations from different networking layers into different feature sets: Network, Transport, and Application layer features. Network-layer features consist of lightweight information available from observing network flows (identified by the IP/port four-tuple) and are typically available in monitoring systems (*e.g.*, NetFlow) [10, 19]. Transport-layer features consist of information extracted from the TCP header, such as end-to-end latency and packet retransmissions. Such features are widely used across the networking space but can require significant resources (*e.g.*, memory) to collect from large network links. Finally, application-layer metrics are the ones that include any feature related to the application data that can be gathered by solely observing packet patterns (*i.e.*, without resorting to deep packet inspection). These features



(a) *The relationship between data representations and model performance for video quality inference*

(b) *Storage cost for collecting a one hour of traffic across different monitoring system on a 10 Gbps link.*

Figure 2: *Balancing traffic data exploration and storage cost.*

capture a unique behavior of the application and have been designed specifically for this problem.

We replicate the work of Bronzino *et al.* [5] by training multiple machine learning models to infer the resolution of video streaming applications over time using the three aforementioned data representations. Figure 2a shows the precision and recall achieved by each representation. As previously presented by the authors, we observe that the performance of a model trained with Network Layer features only (NetFlow in the figure) achieves the poorest performance. Hence, relying solely on features offered by the existing infrastructure would have produced the worst performing models. On the other hand, combining Network and Application features results in more than a 10% increase in both precision and recall. This example showcases how limiting available data representations to the ones typically available from existing systems (*e.g.*, NetFlow) can inhibit potential gains in model performance achievable through representation exploration (depicted by the blue-shaded area in Figure 2b).

Due to the range of models and constantly evolving applications, a system must then enable the collection of customizable data representations. This requirement implies the need for extensible data collection routines that can evolve with Internet applications and the set of inference tasks. Any network monitoring and analysis system should make it easy to control which features are collected from the network traffic as well as offering the ability to the user to specify new features that would not otherwise be known *a priori*.

2.2 Integrated System Cost Analysis

The previous section motivated the need for flexibility in deriving features from raw network traffic. Of course, any representation is possible if packet traces are the starting point, but raw packet traces can be prohibitive in operational networks, especially at high speeds. We demonstrate the amount of storage required to collect traces at scale by collecting a one-hour packet capture from a live 10 Gbps link. As shown in Figure 2b, we observe that this generates almost 300 GB of raw data in an hour, multiple orders of magnitude

more than aggregate representations such as IPFIX/NetFlow. Limiting the capture to solely storing packet headers reduces the amount of data generated, though not enough to make the approach practical. To compute a variety of statistics that would not be normally available from existing systems we would require an online system capable of avoiding the storage cost requirements presented by raw packet captures.

Deploying an online system creates a number of practical challenges caused by the volume and rate of traffic that must be analyzed. Failing to support adequate processing rates (*i.e.*, experiencing packet drops) impacts the correctness of the produced features, potentially invalidating generated models. Fortunately, packet capture at high rates in software has become increasingly feasible due to tools such as PF_RING [11] and DPDK [12]. Thus, in addition to exploiting the available technical capabilities to *capture* traffic at high rates, the system should implement techniques to maximize its ability to ingest traffic and lower the overhead of system processing. For example, the system has to efficiently limit heavyweight processing associated with certain features to subsets of traffic that are targeted by the inference problem being studied without resorting to sampling, which can negatively impact model performance.

Any feature transformation will introduce systems-related costs. A network monitoring system should make it possible to quantify the cost that such transformations impose. Thus, to explore the space of model performance and their associated systems costs, the system should provide an *integrated* mechanism to profile each feature.

3 EXPLORING DATA REPRESENTATIONS WITH TRAFFIC REFINERY

To explore network traffic feature representations and its subsequent effect on both the performance of prediction models and collection cost, we need a way to easily collect different representations from network traffic. Existing monitoring tools do not offer ways to explore the effect that features and representations have on model performance and cost. To enable such exploration, we implement Traffic Refinery, a proof-of-concept system that we have designed and implemented from the ground up to generate flexible feature representations from raw network traffic.

Traffic Refinery works both for *data representation design*, helping network operators explore the accuracy-cost tradeoffs of different data representations for an inference task; and for *customized data collection in production*, once the feature set for an inference task has been defined, Traffic Refinery can be deployed online as the collection tool to extract custom features for a given task.

Data representation design has three steps. First, network operators or researchers define a superset of features worth

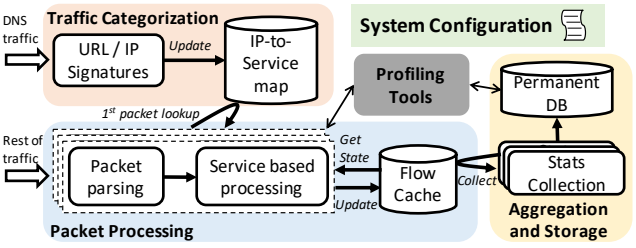


Figure 3: *Traffic Refinery system overview.*

exploring for the task and configure Traffic Refinery to collect all these features for a limited time period. Second, during this collection period, the system profiles the costs associated with collecting each individual feature. Finally, the resulting data enables the analysis of model accuracy versus traffic collection cost tradeoffs.

This section first describes the general packet processing pipeline of the system (Section 3.1) and how a user can configure this pipeline to collect features specific to a given inference task (Section 3.2). These two functionalities are useful both for data representation design and customized data collection in production. We then present how to profile the costs of features for data representation design (Section 3.3).

3.1 Packet Processing Pipeline

Figure 3 shows an overview of Traffic Refinery. Traffic Refinery is implemented in Go [15] to exploit performance and flexibility, as well as its built-in benchmarking tools. The system design revolves around three guidelines: (1) Detect flows and applications of interest early in the processing pipeline and avoid unnecessary overhead; (2) Support state-of-the-art packet processing while minimizing the entry cost for extending which features to collect; (3) Aggregate flow statistics at regular time intervals and store for future consumption. The pipeline has three components:

(1) a *traffic categorization* module responsible for associating network traffic with applications;
(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; and
(3) an *aggregation and 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.

Traffic Refinery is customizable through a configuration file written in JSON format. The configuration provides a way to tune system parameters (*e.g.*, which interfaces to use for capture) as well as 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

```
{
  "Name": "ServiceName",
  "Filter": {
    "DomainsString": ["domain.x", ...],
    "Prefixes": ["10.0.0.0/18", ...]
  },
  "Collect": [FeatureSetA, FeatureSetB, ...],
  "Emit": 10
}
```

Listing 1: *Configuration example.*

should be represented. Listing 1 shows the JSON format used. Using this configuration file, Traffic Refinery establishes a logical pipeline to collect the specified feature sets for each targeted service class, as shown in Figure 4.

3.1.1 *Traffic Categorization*

Traffic Refinery aims to minimize overhead generated by the processing and state of packets and flows that are irrelevant for computing the features of interest. Accordingly, it is crucial to categorize network flows based on their service early so that the packet processing pipeline can extract features solely from relevant flows, ideally without resorting to sampling traffic. To accurately identify the sub-portions of traffic that require treatment online without breaking encryption or exporting private information to a remote server, Traffic Refinery implements a cache to map remote IP addresses to services accessed by users. The map supports identifying the services flows belong to by using one of two methods: (1) *Using the domain name of the service:* similarly to the approach presented by Plonka and Barford [31], Traffic Refinery captures DNS queries and responses and 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. (2) *Using exact IP prefixes:* For further flexibility, Traffic Refinery supports specifying matches between services and IP prefixes, which assists with mapping when DNS lookups are cached or encrypted.

Using 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 [32]). In such situations, we envision Traffic Refinery 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 systems [10].

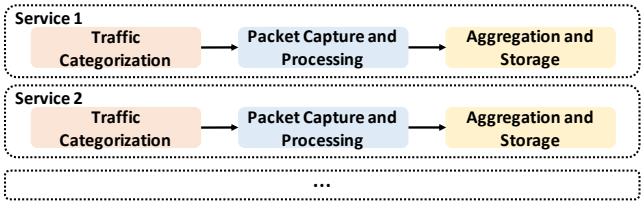


Figure 4: *Logical split across service-driven pipelines.*

3.1.2 *Packet Capture and Processing*

The traffic categorization and packet processing modules both require access to network traffic. To support fast (and increasing) network speeds, Traffic Refinery relies on state-of-the-art packet capture libraries: We implement Traffic Refinery's first two modules and integrate a packet capture interface based on PF_RING [11] and the gopacket *DecodingLayerParser* library [16]. Traffic Refinery 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. Traffic Refinery 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.

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 general flow data structure allows storing different flow types, and differing underlying statistics 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.

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 pre-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, Traffic Refinery 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, Traffic Refinery loops through the list and updates the required statistics.

3.1.3 *Aggregation and Storage*

Traffic Refinery exports high-level flow features and data representations at regular time intervals. Using the time representation information provided in the configuration file, Traffic Refinery initializes a timer-driven process that extracts the information of each service class at the given time intervals. Upon firing the collection event, the system loops through the flows belonging to a given service class and performs the required transformation (*e.g.*, aggregation or sampling) to produce the data representation of the class. Traffic Refinery's packet processing module exposes an API that provides access to the information stored in the cache. Queries can be constructed based on either an application (*e.g.*, Netflix), or on a given device IP address. In the current version of the system, we implement the module to periodically query the API to dump all collected statistics for all traffic data representations to a temporary file in the system. We then use a separate system to periodically upload the collected information to a remote location, where it can be used as input to models.

```
1  // PacketCounters is a data structure to collect
       packet and byte counters
2  type PacketCounters struct {
3    InCounter  int64
4    OutCounter int64
5    InBytes    int64
6    OutBytes   int64
7  }
8
9  // AddPacket increment the counters based on
       information contained in pkt
10 func (c *PacketCounters) AddPacket(pkt *network.
       Packet) {
11   if pkt.Dir == network.TrafficIn {
12     c.InCounter++
13     c.InBytes += pkt.Length
14   } else if pkt.Dir == network.TrafficOut {
15     c.OutCounter++
16     c.OutBytes += pkt.Length
17   }
18 }
19
20 // PacketCountersOutput is a data structure that
       contains the features to output
21 type PacketCountersOutput struct {
22   KbpsUp    float64
23   KbpsDw    float64
24   PpsUp     float64
25   PpsDw     float64
26 }
27
28 // CollectFeatures calculates the features to output
       at given time intervals
29 func (c *PacketCounters) CollectFeatures(slotSize
       float64) PacketCountersOutput{
30   return PacketCountersOutput{
31     KbpsUp:  float64(c.OutBytes) / (slotSize * 128.0)
32     KbpsDw:  float64(c.InBytes) / (slotSize * 128.0)
33     PpsUp: float64(c.OutBytes) / slotSize
34     PpsDw: float64(c.InBytes) / slotSize
35   }
36 }
```

Listing 2: *Implementing a new counters class.*

3.2 User-Defined Traffic Representations

The early steps of any machine learning pipeline involve the application of an engineer's domain knowledge to design features that could result in good model performance. Starting from these features, the model designer can explore and evaluate which representations and models work best for the problem under study. We design Traffic Refinery with the goal of enabling its users to design and collect such features and explore how they affect model performance and collection cost.

To support the collection of a variety of features for each service class, we design Traffic Refinery to use convenient flow abstraction interfaces to allow for quick implementation of collection methods for features and their aggregated statistics. Each flow data structure implements two functions that define how to handle a packet in the latter two steps of the processing pipeline: (1) an `AddPacket` function that defines how to update the flow state metrics using the pre-processed information parsed from the packet headers; (2) a `CollectFeatures` function that allows the user to specify how to aggregate the features collected for output when the collection time interval expires.

Implemented features are added as separate files to the system code structure. Traffic Refinery uses the configuration file to obtain the list of service class definitions and the features to collect for each one of them. Upon execution, the system uses Go's language run-time reflection to load all available feature classes and select the required ones based on the system configuration. The implemented functions are then executed respectively during the packet processing step or during representation aggregation. We detail in Section 5 how the system can be configured to flexibly collect features at deployment time for a video inference use case.

As an example, we show in Listing 2 our implementation of the PacketCounters feature class. This collection of features, stored in the `PacketCounters` data structure, keeps track of the number of packets and bytes for observed flows. To do so, the `AddPacket` function uses the pre-processed information which is stored in the `Packet` provided as input (*i.e.*, the direction of the packet and its length). Upon triggering of the collection interval, the system uses the structure to

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

Table 1: *Current common features available in Traffic Refinery.*

output throughput and packets per second statistics, *i.e.*, the `CollectFeatures` function and the output data structure `PacketCountersOutput`. The current release of the system provides a number of built-in default features commonly collected across multiple layers of the network stack. Table 1 provides an overview of the features currently supported.

Design considerations. We took this design approach to offer full flexibility in defining new features to collect while also minimizing the amount of knowledge required of a user about the inner mechanics of the system. We made several compromises in developing Traffic Refinery. First our design focuses on supporting per flow statistics and output them at regular time intervals. This approach enables the system to exploit established packet processing functions (*e.g.*, clustering) to improve packet processing performance. Conversely, this solution might limit a user's ability to implement specific types of features, such as features that require cross flows information or based on events. Second, the software approach for feature calculation proposed in Traffic Refinery might encourage a user to compute statistics that are ultimately unsustainable for an online system deployed in an operational network. To account for this possibility, the next section discusses how the system's cost profiling method provides a way to quantify the cost impact that each feature imposes on the system. Ultimately, this analysis should provide feedback to a user in understanding whether such features should be considered for deployment.

3.3 Cost Profiling

A detailed analysis on each feature's cost and effect on the collection pipeline can instruct model designers on the feasibility of a given feature's collection and ultimately provide a framework to analyze the tradeoffs between model performance and its costs. Towards this goal, Traffic Refinery aims to provide an intuitive and automated platform to evaluate the system cost effects of the user defined data representations presented in the previous section. To do so, we use Go's built-in benchmarking features and implement dedicated tools to profile different costs intrinsic to the collection process. At data representation design time, users employ the profiling method to quickly iterate through the collection of different features in isolation and provide a fair comparison for three cost metrics: state, processing, and storage.

State costs. We aim to collect the amount of in-use memory over time for each feature class independently. To achieve this, we use Go's `pprof` profiling tool. 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 aggregation and storage module gathers the data from the cache. We choose this instant because it corresponds to the memory usage peak for each time interval cycle.

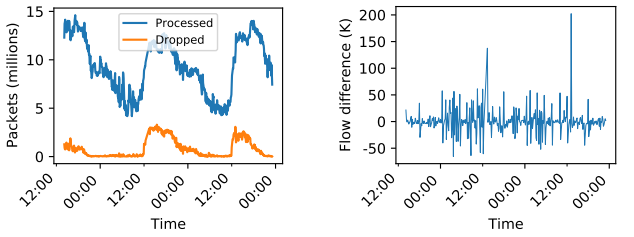
Processing costs. To evaluate the CPU usage for each feature class, we aim to monitor the amount of time required to extract the feature information from each packet, leaving out any operation that shares costs across all possible classes, such as processing the packet headers or reading/writing into the cache. To do so, we build a dedicated time execution monitoring function that tracks the execution of each `AddPacket` function call in isolation, collecting running statistics (*i.e.*, mean, median, minumum, and maximum) over time. This method is similar in spirit to Go's built-in benchmarking feature but allows for using raw packets captured from the network for evaluation over longer periods of time.

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 Section 5.2 a comparison of the amount of data generated by the current version of the system with alternative traffic representations: pcap traces and NetFlow.

Cost profiling analysis. Traffic Refinery supports two modes for profiling feature costs: (1) Profiling from live traffic: in this setting the system captures traffic from a network interface and collects statistics for a configurable time interval; and (2) Profiling using a offline traffic traces: in this setting profiling runs over recorded traffic traces, which enables fine-grained inspection of specific traffic events (*e.g.*, a single video streaming session) as well as repeatability and reproducibility of results. Similarly to Go's built-in benchmarking tools, our profiling tools run as standalone executables. To profile sets of user-defined features (as described in Section 3.2), the profiling tool takes as input the same system configuration file used for executing the system. Upon execution, the system creates a dedicated measurement pipeline that collects statistics over time.

4 FEASIBILITY EVALUATION

To examine the traffic processing capacity of Traffic Refinery, we deploy the system on a commodity server equipped with



(a) *Packets processed vs drops* **(b)** *Flow cache size changes over time.*

Figure 5: *Traffic Refinery performance on the server.*

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.[1] The link routinely reaches nearly full capacity (*e.g.*, roughly 9.8 Gbps) during peak times each day during the school year. We evaluate Traffic Refinery on the link over several days in October 2020. We use the PF_RING packet library with zero-copy enabled in order to access packets with minimal overhead. Although 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 an operator might use Traffic Refinery to deploy its monitoring models, and enables us to better understand potential system bottlenecks.

Figure 5a shows the number of packets processed and the number of packets dropped in 10 second windows over the course of a few days collecting the features required to infer video quality metrics in real time for eleven video services (more details on the use case are presented in the next section). We see that traffic tends to show a sharp increase mid-day, which coincides with an increase in the rate of packet drops. Overall, Traffic Refinery can process roughly one million packets per-second (10M PPS per ten-second window in the figure) without loss, while higher rates trigger loss. We believe 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 to understand bottlenecks in the Traffic Refinery 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 6,000 nanoseconds versus 200 nanoseconds. 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 5b 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). The spikes are thus a result of Traffic Refinery 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, Traffic Refinery 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 result 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 Traffic Refinery to be deployed in higher-speed production networks. We leave such system optimization engineering for future work.

5 USE CASE: VIDEO QUALITY INFERENCE

We demonstrate the potential of Traffic Refinery by prototyping a common inference task from the literature: streaming video quality inference (in particular, using the methods from Bronzino *et al.* [5]). This not only allows us to empirically measure systems-related costs of data representation for this problem, 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.

The rest of the section presents how we configure and use Traffic Refinery for the three phases of the data representation design: (1) definition and implementation of a

[1] 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.

```
{
  "Name": "Netflix",
  "Filter": {
    "DomainsString": ["netflix.com","nflxvideo.net","nflximg.net","nflxext.com","nflximg.com","nflxso.net"],
    "Prefixes": "23.246.0.0/18", "37.77.184.0/21", "45.57.0.0/17", "64.120.128.0/17", "66.197.128.0/17", "108.175.32.0/20", "185.2.220.0/22", "185.9.188.0/22", "192.173.64.0/18", "198.38.96.0/19", "198.45.48.0/20", "208.75.79.0/24", "2620:10c:7000::/44", "2a00:86c0::/32"]
  },
  "Collect": [PacketCounters, TCPCounters, VideoSegments],
  "Emit": 10
}
```

Listing 3: *Configuration to capture video features for Netflix.*

superset of candidate features (Section 5.1); (2) feature collection and evaluation of system costs (Section 5.2); and finally, (3) analysis of the cost performance tradeoffs (Section 5.3).

5.1 Traffic Refinery Customization

As first presented in Section 2, Bronzino *et al.* [5] categorized useful features for video quality inference into three groups that correspond to layers of the network stack: Network, Transport, and Application Layer features. In their approach, features are collected at periodic time intervals of ten seconds. The first ten seconds are used to infer the startup time of the video, while remaining time intervals are used to infer the ongoing resolution of the video being streamed.

We add approximately 100 lines of Go code to implement in Traffic Refinery the feature calculation functions to extract application features (*i.e.*, `VideoSegments`). Further, we use built-in feature classes to collect network (*i.e.*, `PacketCounters`) and transport (*i.e.*, `TCPCounters`) features. We use these classes to configure the feature collection for 11 video services, including the four services studied in [5]: Netflix, YouTube, Amazon Prime Video, and Twitch. As an example, Listing 3 shows the configuration used to collect Netflix traffic features.

5.2 Data Representation Costs

Model performance is not the lone concern for network operators; they must also account for system-related costs for any deployment scenario. We evaluate costs focusing on those related to the three classes of features used for video quality inference problem: network, transport, and application features. First we use Traffic Refinery's profiling tools to quantify the fine-grained costs imposed by tracking video streaming sessions. To do so, we profile the per-feature state and processing costs for pre-recorded packet traces with 1,000 video streaming sessions split across four major video streaming services (Netflix, YouTube, Amazon Prime Video, and Twitch). Then, we study the effect of collecting the different classes of features at scale by deploying the system in a 10 Gbps interconnect link.

We find that while some features add relatively little state (*i.e.*, memory) and long-term storage costs, others require substantially more resources. Conversely, we observe that processing requirements are within the same order of magnitude for all three classes of features.

5.2.1 *State Costs*

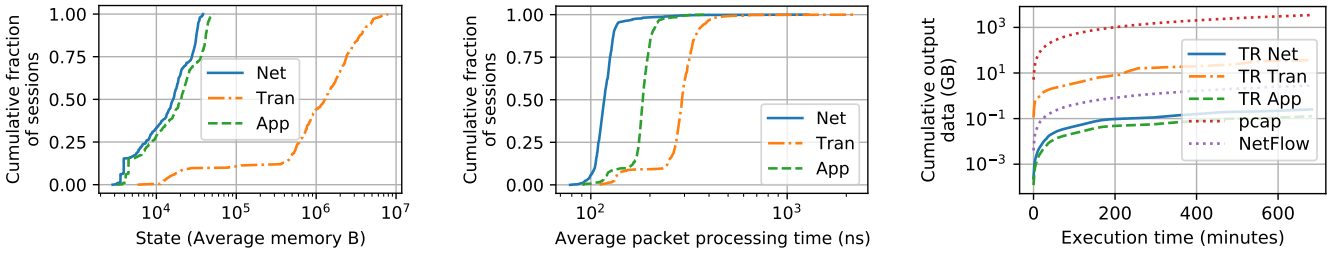
We study the state costs as the amount of in-use memory required by the system at the end of each collection cycle—*i.e.*, the periodic interval at which the cache is dumped into the external storage. Figure 6a shows the cumulative distribution of memory in Bytes across all analyzed video streaming sessions. 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 transport features can require up to three orders of magnitude more memory compared to network and application features. Transport features require historical flow information (*e.g.*, all packets) in contrast with network features that require solely simple counters.

Further, we see that the video flow features require a median of a few hundred MB in memory on the monitored link, with a slightly larger memory footprint than network features. At first glance, we assumed that this additional cost was due to the need for keeping in memory the information about video segments being streamed over the link. Upon inspection, however, we realized that streaming protocols request few segments at a time per time slot (across the majority of time slots the number of segments detected was lower than three), which leads to a minimal impact on memory used. We then concluded that this discrepancy was instead 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.

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 6b shows distributions of the time required to process different feature classes. We observe that collecting simple network counters requires the least processing time, followed by application and transport features.

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, for



(a) *State required for different features.* **(b)** *Processing costs for different representations.* **(c)** *Storage costs for different representations.*

Figure 6: *Cost profiling for video inference models.*

this particular service class, state costs have a much larger impact than processing cost on the ability of collecting features in an operational network.

5.2.3 *Storage Costs*

Feature retrieval at scale can 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 6c shows the amount of data produced by Traffic Refinery when collecting data for the three feature classes relevant to the video streaming quality inference on the monitored link. 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 (*e.g.*, five minutes sampling), which collects aggregated per flow data volume statistics; this roughly corresponds to the same type of information collected by Traffic Refinery for the network layer features.

We observe that storage costs follow similar trends as the state costs previously shown. 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 see that raw packet traces generate a possibly untenable amount of data and if used continuously can quickly generate terabytes of data. This result supports our claim that collecting traces at scale and for long periods of time quickly becomes impractical. Next, 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 that having additional flexibility in exporting additional features, *e.g.*, video segments information, may introduce low additional cost. In the next section, we demonstrate that having such features available may result in significant model performance benefits.

5.3 Model Performance

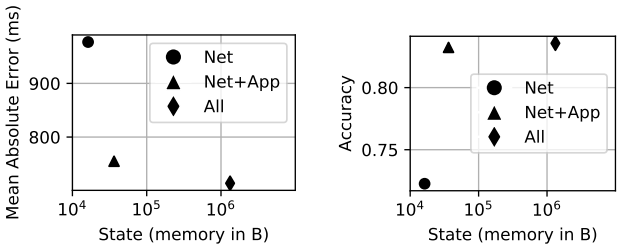
In this section, we study the relationship between model performance and system costs for online video quality inference. We use previously developed models but explicitly explore how *data representation* affects model performance.

Our investigation in this section constitutes 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. Elevating data collection costs as a primary model evaluation metric serves as a significant departure from much previous work in this area. We hope these results can act as a rubric for evaluation of other models that rely on machine learning for prediction and inference from network traffic in the future.

In this section, we focus on state-related costs (*e.g.*, memory), as for video quality inference, state costs mirror storage costs and the differences in processing costs of the feature classes is not significant (Section 5.2). Interestingly, we find that the relationship between state and model performance is not proportional. More importantly, we find that it is often possible to significantly reduce the state-related requirements of a model without significantly compromising prediction performance, further bolstering the case for systems like Traffic Refinery that allow for flexible data representations.

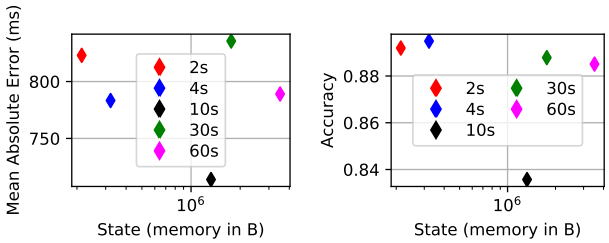
5.3.1 *Representation vs. Model Performance*

To understand the relationship between memory overhead and inference accuracy, we use the dataset of more than 13k sessions presented in [5] 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 [5]; in particular, random forest regression for startup delay and random forest multi-class classifier for resolution.



(a) *Startup delay (lower is better).* **(b)** *Resolution (higher is better).*

Figure 7: *The relationship between features state cost and model performance for video streaming quality inference.*



(a) *Startup delay (lower is better).* **(b)** *Resolution (higher is better).*

Figure 8: *The relationship between time granularity state costs and model performance for video quality inference.*

Further, we use the same inference interval size, *i.e.*, ten-second time bins.

Figure 7 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. Adding application layer features contributes to a very small additional memory overhead. 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.

5.3.2 *Time Granularity vs. Model Performance*

State of the art inference techniques (*i.e.*, Bronzino *et al.* [5] and Mazhar and Shafiq [24]) employ ten-second time bins to perform the prediction of the features. This decision is justified as a good tradeoff 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.

As in Section 5.3.1, we train different random forest models with increasing time bin sizes of 2, 4, 10, 30, and 60 seconds. To understand the possible memory impact of the different time bins, we use all features (All) to train the models. In Figure 8, we observe different outcomes for the two quality metrics. For startup delay, the results show that ten-second windows can indeed provide a good tradeoff between memory and prediction accuracy, achieving a minimum of 70 ms better predictions than all other time granularities. This result shows that ten seconds is an acceptable tradeoff between gathering enough information at the beginning of a session without adding too much data from segment downloads that happens after the video has started.

Interestingly, the results for resolution inference show that ten-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 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 about the length of the shortest video segments across all services, as well as 60-second time windows, which could contain many video quality changes within the time slot caused by the download of multiple video segments.

6 RELATED WORK

Machine learning models have become an increasingly integral component in solving a number of network management tasks [3, 28, 35], spanning from performance inference to detection of network events and more.

Collecting input data to build models for network management tasks can be typically achieved with passive network monitoring tools, such as packet captures (*e.g.* libpcap [36] and its derivative applications Wireshark [29] and Tshark [38]) or flow captures (*e.g.* NetFlow [6], IPFIX [7]). However, this set of network monitoring tools inhibits model designers from exploring the space of possible data representations. On one hand, packet captures generate massive volume of data which makes them a none-viable approach for large networks. On the other, flow captures produce statistical information that are too coarse grained to enable

full exploration of all possible data representations. Traffic Refinery strikes a balance by providing model designers the ability to define custom representations of the data they wish to collect, and profile the systems-related costs associated with the data collection.

Streaming analytics platforms [1, 9, 17, 26, 41] and algorithms (*e.g.*, "Sketches") [20, 21, 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.

Advanced network monitoring and analysis tools such as Tstat [14, 25], Bro [30], and Snort [33] are closest in spirit to Traffic Refinery 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, these tools would need to be adapted to achieve custom feature representation, data representation exploration, and profiling data collection costs.

Additionally, a number of commercial solutions that apply machine learning to netwok traffic, have recently emerged to offer a variety of network management solutions, ranging from network analytics to customer support. Such solutions (*e.g.*, Nokia's Traffica [37] and Deepfield [10], NIKSUN's NetCVR [27]), focus on black box approaches for answering targeted questions. On the other hand, Traffic Refinery focuses on providing an open solution for jointly evaluating model performance and features collection costs in the early stages of the model development.

Finally, few recent work considered the costs associated with ML-systems [4, 5, 34]. Bronzino *et al.* addressed the problem of inferring the quality of video streaming applications from encrypted traffic. The authors classified the possible set of features based on their corresponding layer in the network stack. This categorization enabled the authors to reason about the cost associated with each features sets, and thus design performant models based on lightweight features only. Sculley *et al.*, [34] investigate the hidden "technical debt" incurred by fast-paced development and deployment of ML systems. The authors reveal a number of system-level factors that build up the maintenance costs of real-world ML systems overtime; *e.g.*, unstable or undertutilized data, dependencies on proprietary packages, entanglement of input signals, to name a few. Breck *et al.* [4] built on the work in [34] to propose the "ML Test Score": a scoring system based on 28 actionable tests designed to measure how production ready ML-systems are. The ultimate goal of this scoring system is to lower their long-term maintenance costs. Among the list of proposed tests, the authors highlight the importance of measuring features cost in terms of added inference latency, RAM usage, as well as data dependencies and potential instabilities associated with each feature. Traffic Refinery inherently supports these aforementioned best practices to support the exploration of data representations for a wide range of network management tasks.

7 CONCLUSION

The performance of any machine learning model often depends as much on the input data representation as it does on the model itself. Yet, existing network traffic data representations (e.g., IPFIX) are typically determined far before the data itself is used for input to machine learning pipelines. Recognizing that there are a range of representations that may be appropriate for machine learning pipelines, this paper introduces Traffic Refinery, which allows operators to define custom representations that consider both model accuracy and the systems-related costs of collecting the associated features. We present the design and implementation of a packet processing pipeline that allows for such customization, show how users can customize the pipeline and profile the systems costs associated with customized features, and demonstrate the promise of Traffic Refinery on a case study involving video quality performance inference. Several open problems remain, including automating the exploration of the design space to find the optimal operating point for model accuracy and systems costs.

Although 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. Additionally, other deployment considerations exist when different models require different representations, as well: such a scenario may require, for example, joint optimization, or deploying multiple instances of Traffic Refinery, from multiple switch ports or even at multiple vantage points across the network. Traffic Refinery enables this exciting line of research by demonstrating both the feasibility and utility of exploring a range of traffic representations, beyond the current, limited dichotomy of packet captures and IPFIX. To enable the community to explore these benefits on a wider range of problems, we have both released Traffic Refinery as open-source software, as well as the evaluation in this paper.

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] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D Sculley. 2017. The ml test score: A rubric for ml production readiness and technical debt reduction. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 1123–1132.

[5] 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.

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

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

[8] corelight 2019. Corelight. https://corelight.com/. (2019).

[9] 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.

[10] deepfield 2019. Deepfield. https://deepfield.com/. (2019).

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

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

[13] 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 (SIGCOMM '02)*. ACM, New York, NY, USA, 323–336. https://doi.org/10.1145/633025.633056

[14] 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.

[15] golang 2020. Go language. https://golang.org/. (2020).

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

[17] 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 (SIGCOMM '18)*. ACM, New York, NY, USA, 357–371. https://doi.org/10.1145/3230543.3230555

[18] Craig Gutterman, Katherine Guo, Sarthak Arora, Xiaoyang Wang, Les Wu, Ethan Katz-Bassett, and Gil Zussman. 2019. Requet: Real-time qoe detection for encrypted youtube traffic. In *Proceedings of the 10th ACM Multimedia Systems Conference*. 48–59.

[19] kentik 2019. Kentik. https://kentik.com/. (2019).

[20] 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.

[21] 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.

[22] Tarun Mangla, Emir Halepovic, Mostafa Ammar, and Ellen Zegura. 2019. Using session modeling to estimate HTTP-based video QoE metrics from encrypted network traffic. *IEEE Transactions on Network and Service Management* 16, 3 (2019), 1086–1099.

[23] M Hammad Mazhar and Zubair Shafiq. 2018. Real-time video quality of experience monitoring for https and quic. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 1331–1339.

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

[25] 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.

[26] 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.

[27] netvcr 2020. NIKSUN NetVCR. https://www.niksun.com/product.php?id=110. (2020).

[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] David Plonka and Paul Barford. 2011. Flexible traffic and host profiling via DNS rendezvous. In *Workshop Satin*.

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

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

[34] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. In *Advances in neural information processing systems*. 2503–2511.

[35] 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.

[36] tcpdump 2020. tcpdump and libpcap. https://www.tcpdump.org/. (2020).

[37] traffica 2020. Nokia Traffica. https://www.nokia.com/networks/products/traffica/. (2020).

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

[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 (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.