Efficient Network Monitoring Applications in the Kernel with eBPF and XDP

Marcelo Abranches*, Oliver Michel†, Eric Keller*, Stefan Schmid‡
*University of Colorado Boulder †Princeton University ‡TU Berlin, University of Vienna, and Fraunhofer SIT

Abstract—**Continuous traffic monitoring and analytics are fundamental to the operation of today's networks. Network telemetry allows for performing fine-grained analytics on network flow or packet records for various use cases including intrusion detection and traffic engineering. While some analytics tasks can be offloaded to programmable switches, ultimately, telemetry data needs to be processed by analytics applications in software. These applications are highly specialized, and running many such applications concurrently to achieve high coverage is expensive. To reduce the resource footprint of software network analytics, we present a novel network monitoring primitive that consolidates logic which all monitoring applications require. The primitive can (partially) be offloaded to a SmartNIC and triggers applications only when required based on high-level traffic metrics, avoiding unnecessary and redundant computations. We identify eBPF and XDP as a natural fit for this task, and implement a prototype of our system on top of this novel technology. Our evaluation shows that the combination of conditional execution of analytics tasks and the use of modern packet I/O technologies not relying on expensive busy polling (e.g., as in DPDK) significantly reduces the resource footprint of performing continuous network analytics.**

I. Introduction

Continuous, fine-grained traffic monitoring is essential to the operation of today's reliable communication networks. In a nutshell, network monitoring and analytics describe the process of extracting information from network devices in the form of statistics or traffic records and transforming this data into meaningful insights to be used for network management decisions. This enables operators to detect changing demands, performance issues, or attacks and subsequently reconfigure the network or scale network functions [1]–[4].

In today's networks, switches and routers continuously export data about the traffic that traverses the network to analytics applications running on general-purpose servers [3, 5]. These applications detect problems or calculate metrics for a variety of use cases. Programmable switches allow for some applications to be partially offloaded to the network [1, 2, 5].

Data centers and wide area networks carry hundreds of millions of packets per second requiring significant processing performance to enable fine-grained analytics [6, 7]. Offloading parts of the analytics pipeline to programmable switches can significantly reduce the load of the software-based stream processing backend; but even then, the rate of events to be analyzed in software is often still on the order of several million events per second per application [2]. This is because many, especially complex or state-intensive, tasks can only be partially offloaded due to the limited memory and compute resources and constrained programming model offered by line rate hardware [6]. As a result, performing analytics in software using either general-purpose stream processors or specialized packet analytics frameworks is indispensable to deploying complex, parallel, and dynamic network analytics.

Deploying network-wide, fine-grained, software-based analytics in a resource-efficient manner is still a major challenge. In particular, we identified two main issues with the state-of-the art in software-based network analytics. First, operators need to run multiple different network analytics tasks in parallel in order to achieve high coverage across possible failures and attacks, and to have continuous, detailed insight into different aspects of the operation of their infrastructure [5]. Running multiple applications in parallel at all times is costly. Many network conditions (e.g., attacks), however, can be identified by shared lightweight logic and simple, high-level metrics (e.g., overall connection count) that can then be used to trigger finer-grained analysis only when required.

Second, most existing network analytics systems leverage kernel-bypass frameworks for packet input [6, 8, 9]. Kernel-bypass can achieve high packet rates but is expensive as at least one CPU core is always entirely dedicated to packet input alone due to busy polling on the network interface card (NIC) [8]. This is wasteful as these CPU cycles cannot be used for the actual task of performing analytics. While receiving high volumes of packets via sockets is also inefficient or impossible, the novel eXpress Data Path (XDP) [10] technology not only provides a resource-efficient way to ingest millions of packets per second without using busy polling; it also provides abstractions particularly suited for orchestrating multiple packet processing applications and efficiently running them in parallel.

To address these challenges, this paper presents a network monitoring architecture designed around a novel primitive which consolidates logic all monitoring applications require. The XDP technology is a natural fit to realize our architecture, and we show that the resulting system provides several performance and architectural advantages over the state-of-the art. Our paper makes the following contributions:

1) We identify the opportunity to consolidate monitoring system tasks in a novel network monitoring primitive. The primitive efficiently computes high-level metrics and can (partially) be offloaded to a SmartNIC.
2) We propose an architecture for network analytics systems that allows for dynamic orchestration of analytics appli-

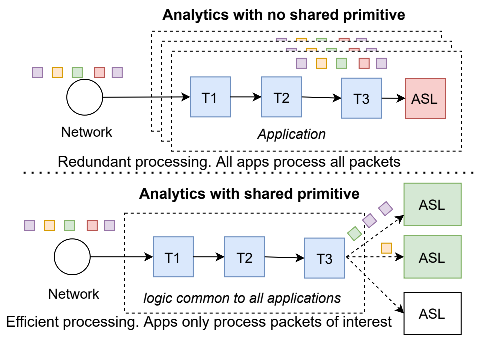

Analytics with no shared primitive
Network
T1
T2
T3
ASL
Application
Redundant processing. All apps process all packets
Analytics with shared primitive
Network
T1
T2
T3
logic common to all applications
ASL
ASL
ASL
Efficient processing. Apps only process packets of interest

Figure 1. Efficient analytics with shared primitive. T1: Receive and select records, T2: compute high-level statistics, T3: conditionally execute app specific logic, ASL: Application-specific logic

cations based on high-level metrics and policies.

3) We implement a prototype of this architecture on a Netronome NFP SmartNIC [11] and for the Linux kernel using eBPF and XDP to demonstrate its feasibility.
4) Using benchmarks, we show the high performance and small resource footprint of our approach using three example applications.

In the remainder of this paper, we motivate our work by elaborating on the challenges in software network analytics in Section II. We then present an architecture for efficient and practical network analytics in the kernel in Section III. Section IV describes our prototype and challenges we encountered during its implementation. We demonstrate the performance of our system in Section V before discussing related work and concluding in Sections VI and VII, respectively.

II. Motivation

As previously described, two main challenges in operating software-based network analytics are related to (a) reducing the system resource footprint when running analytics tasks in parallel and (b) efficiently ingesting and processing high rates of network records. We will now elaborate on both challenges.

Efficiently Deploying Parallel Applications. Analytics applications are usually highly specialized and focus on one particular type of scenario, such as a specific attack or common performance problem [1, 2, 4]. Accordingly, applications must be used in parallel to achieve high coverage across monitoring tasks such as intrusion detection [3], analyzing performance issues [1, 9], or traffic classification [12]. Even monitoring a network for just the most common attacks or anomalies therefore requires continuously running many specialized applications thus incurring high cost [5].

Despite the heterogeneity of analytics tasks, all share common tasks and logic. All applications read in network records from a NIC and decide which records carry measurements and which are control (or other) traffic. Then, especially those applications detecting some condition (e.g., an attack or performance anomaly), are often designed to be triggered based on shared, high-level metrics involving packet, Byte, or flow counters [13]. Such metrics may be used for several applications allowing for deduplicating logic and dynamically enabling and disabling more expensive, finer-grained analysis. As a result, (1) performing shared tasks, (2) computing metrics relevant to all applications, and (3) conditionally executing applications based on these metrics can be consolidated at the system-level; an overview of this is shown in Figure 1.

For example, a SYN flood toward a particular host would manifest itself not only in the particular pattern of a high amount of unanswered SYN+ACK segments, but also initially in an increase of overall flows. This basic higher-level metric can efficiently be computed on all traffic using, for example, probabilistic data structures. A change in a metric can then trigger the activation of a series of more fine-grained analytics applications that are designed to mine the required and more useful information, such as the origin of the attack to subsequently configure filtering. Today, we are missing an architecture including a common primitive that consolidates tasks needed by all analytics tasks and enables the use of high-level metrics to dynamically enable, disable, and orchestrate downstream analytics applications.

Efficiently Ingesting High-volume Packet Streams. To cope with high traffic rates in software, existing software frameworks leverage kernel-bypass technology (e.g., by using DPDK [8]) to ingest network packets at high rates [6]. While this provides high input rates, the use of busy polling in these implementations causes significant CPU consumption leaving fewer cycles for actual analytics. Furthermore, using kernel-bypass renders all in-kernel network processing capabilities (e.g., use of routing tables, firewalls, sockets) useless on the particular NIC in use and makes integration with other legacy applications challenging or impossible.

Both issues are especially problematic for distributed telemetry frameworks, such as SwitchPointer [9] where in-network measurements are processed and stored on all hosts in the network. Using kernel-bypass here would unnecessarily waste CPU cycles on all servers. Also, in this architecture, the telemetry sinks are not dedicated analytics servers and must also perform their regular purpose requiring NIC access via sockets and kernel network processing.

To enable high-performance user-defined packet processing while integrating with the OS and still allowing socket-based applications on the same NIC, the eXpress Data Path (XDP) [10] has been introduced in the Linux kernel. XDP allows to attach Extended Berkeley Packet Filter (eBPF) programs early in the kernel's packet processing path, enabling programmability at performance close to kernel-bypass technologies while leaving the kernel's packet processing functions usable. eBPF programs can be loaded and dynamically chained at runtime; they support stateful processing and offloading to compatible NICs to further boost performance.

While this novel technology is promising for a wide range of packet processing applications, we believe that eBPF and XDP are particularly useful as a platform for the practical deployment of high-performance network analytics applications and can solve many of the above outlined challenges for several reasons. First, common analytics functionality can be implemented in a shared eBPF program that can even be offloaded to a SmartNIC. This common logic can easily be changed and written using the same language and programming model (eBPF programs in C) as the analytics tasks themselves, simplifying development and adoption. Second, monitoring tasks implemented as eBPF programs can be injected and activated at runtime from user space. Multiple such applications can be orchestrated as a chain where each task feeds its results into the next one. Third, this mechanism of high-performance packet processing is lightweight and saves CPU cycles compared to kernel-bypass solutions [10]. It does not take ownership of the NIC and is transparent to kernel-based packet processing and user space networking applications, making this technology particularly suitable for distributed measurement systems.

III. A Primitive for Network Monitoring Systems

At the heart of our proposed system lies a novel network monitoring primitive that manages a set of monitoring applications and conditionally executes them based on a set of basic metrics in conjunction with operator-specified policies. Performing these tasks at the system-level rather than in each individual application has several advantages. First, application developers can write slimmer applications that focus on the measurement task and do not require logic for input/output, decapsulation of records, and computation of triggers to decide whether a record needs to be processed or not. Second, it provides a simple abstraction for orchestrating and triggering chains of applications reducing the complexity required to build practical, high-coverage monitoring systems, increasing network security and reliability. Finally, consolidating these operations avoids duplicated logic and ultimately saves resources which increases performance and saves cost.

Network Monitoring Applications. Before presenting our system in more depth, we first define what a *network monitoring application* is and elaborate on when and how an operator might want to run a specific application. A network monitoring application is a piece of logic that transforms a high-volume stream of measurements collected in the network into a lower-volume stream of data that provides useful insights for the operator. An insight is useful if it provides enough detail for the operator to make network management decisions with the goal of improving or restoring the correct and reliable operation of the network or to perform other required tasks, such as billing. Network management decisions usually result in reconfiguring a network function (e.g., a router or firewall) or adding, scaling, or removing functions. As previously explained, applications serve diverse use cases ranging from intrusion detection and traffic engineering to profiling and debugging. While the applications are highly specialized, their logic alone is often relatively simple; many applications add, update, or delete state based on some logic for each received measurement and generate an event if a condition is met.

As applications and their measurement and analytics tasks are diverse, when and how an application should run can also differ depending on the use case. We identified three main cases how an operator might want to deploy an application. First, an application might need to run at all times. This is the case for lightweight monitoring applications that go beyond basic device-level counters, such as a traffic accounting and billing application in a cloud setup. Second, an application can run only when explicitly activated by the operator. This mode allows for, for example, ad-hoc queries or other profiling and debugging tasks, such as detecting an imbalance in ECMP routing. Finally, an application can be automatically triggered by a higher-level condition, such as a sudden increase in connections or overall traffic volume. Here, just knowing about the increase in a metric is not sufficient to apply configuration changes to the network (e.g., block a host). As a result, more fine-grained applications need to be deployed rapidly to mine more facts, such as the set of hosts affected.

Receiving and Filtering Records. We now describe the three main components of our primitive and explain how an operator uses their respective APIs to configure and orchestrate a set of monitoring applications. The overall system architecture is depicted in Figure 2; the three components of the monitoring primitive which we will explain now are labeled A, B, and C. For the remainder of this paper we focus on network traffic records, in particular formatted per-packet records where each telemetry packet represents one network packet that traversed the monitored device. Each record contains the original packet's IP 5-tuple, ingress switch port number and queue depth, μs-timestamp, packet size, IP-ID, and (if applicable) TCP flags. The records are 32 Bytes in length. This is an example format for the purpose of this discussion and for our prototype; other formats can easily be supported through minor changes in the packet parsing logic.

The receiver component is the entry point to our system. Streaming telemetry records are usually encapsulated in UDP datagrams and sent to a specific IP and port combination on the analytics server. There, a monitoring system needs to differentiate telemetry traffic from control or other traffic destined for the respective machine. Which traffic should be considered monitoring traffic can be specified using filter rules that perform an exact match on the IP 5-tuple of the received packets (not the carried record). Unmatched packets are passed to the kernel to be received by any other application (e.g., the host's SSH server) or dropped at this point.

Packets for the analytics system are then decapsulated and later required metadata fields are prepended. The metadata fields include a list of monitoring application identifiers that will later be populated for record routing and a field for monitoring data derived from a hash computed over the records' IP 5-tuple (see Section IV). These steps are stateless operations and can efficiently be offloaded to programmable hardware, e.g., in our case, a SmartNIC.

Our architecture supports various types of input records including formatted packet or flow records, mirrored packets or headers, and regular data packets that may carry telemetry data (i.e., in-band network telemetry, INT [14]). Regular packets would, of course, have to be injected into the normal kernel path after all analytics tasks have been completed.

High-level Traffic Monitoring. The high-level monitor computes statistics relevant to all applications and required for the subsequent routing process. It runs at all times and

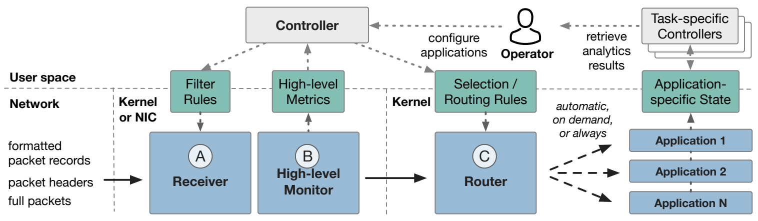
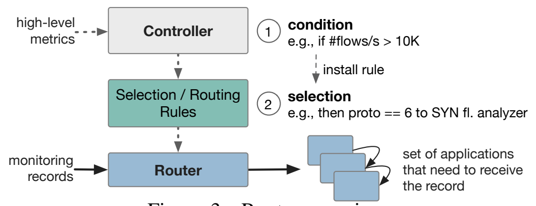


Figure 2. System architecture overview



high-level metrics
Controller
1 condition
e.g., if #flows/s > 10K
install rule
Selection / Routing Rules
2 selection
e.g., then proto == 6 to SYN fl. analyzer
monitoring records
Router
set of applications that need to receive the record

Figure 3. Router overview

can also serve as a baseline network monitor, e.g., if no further analytics tasks are currently required. The metrics are scalar values aggregated over a time period (e.g., a second). Our system computes 8 basic counters: the number of packets and Bytes per interval for all records and for TCP, UDP, and ICMP traffic, respectively. This is useful, for example, to detect a shift in the ratio between the protocol types as it would occur in various flooding attacks. It also computes the number of unique flows (as per IP 5-tuple) seen in each interval.

As this module is executed for every received record, it is important that the computation is lightweight. While the basic counters can be incremented efficiently, for example, computing the number of flows would usually require a more heavyweight set data structure. Our prototype leverages a HyperLogLog sketch (HLL) [15] to estimate the number of unique flows. The counters as well as the sketch data structures are stored in memory shared between the controller and data path. The monitor writes into this memory for each packet while the controller reads the values and resets all state after each time period. As this process requires stateful computations, not every type of metric calculation can efficiently be offloaded to a SmartNIC (see more details in Section IV).

Routing Records to Applications. Finally, the routing component, conceptually depicted in Figure 3, is responsible for determining the set of analytics applications that should receive a particular record. The inputs to the router are the incoming record stream, the current snapshot of previously computed high-level metrics, as well as policies defined by the operator. The policies describe which records under which condition should be sent to a specific application.

At the core of the router is a series of match+action tables for different subsets of the packet's header space (e.g., destination IP address or a combination of fields). An action is a list of monitoring applications that should receive the record. Each record contains the previously initialized list of applications in its metadata. After each match, the router appends the list of applications in the matched entry to the list in the record's metadata and the record will subsequently traverse all applications in this list. The match+action tables are populated by the controller and then read by the router to construct the list of application for the respective record. This allows for a two-step process where first the controller checks whether an operator-defined *condition* is true and then populates the tables for *selection* of the relevant records.

The condition determines when an application should receive a record based on operator policies and previously computed metrics (i.e., when an application should be triggered). The selection step then defines which records, in terms of matches on header fields, are relevant to the triggered application. For example, an application detecting out-of-order TCP segments should not receive UDP traffic at all. The controller provides the operator access to the current state of high-level metrics and exposes an API to add and remove selection rules. The conditions can be implemented either directly in the controller or through an external component (e.g., a script) that consumes metrics and installs rules accordingly.

Above, we outlined three different modes of how and when an operator might want to run an application. Our primitive supports these three modes. First, an application that needs to run at all times, simply has a permanent entry which is installed at system initialization that specifies which slice of the network traffic should always be sent to the application. Second, an application can run only when explicitly activated by the operator (e.g., for ad-hoc queries or debugging). This is possible as the match+action tables can be modified at runtime. Adding or removing an entry does not incur downtime or disruption in the monitoring system. Finally, an application can be automatically triggered by a condition over high-level metrics. A condition is a logical expression, e.g., number of flows per second greater than 10K and is checked after each time interval. If a condition is true, the respective application (or set of applications) is activated for the next time interval. This mechanism is powerful as it allows to (a) execute more computationally expensive applications only when required in order to save resources and (b) autonomously analyze an ongoing issue by deploying operator-defined profiles of more fine-grained applications. More complex conditional execution, e.g., using automated anomaly detection or separate conditions for when an application should be deactivated again are possible in our model but beyond the scope of this work.

IV. Implementation

We implemented our system and three example applications in approximately 1800 lines of code [1]. The data plane components consisting of the monitoring primitive and the individual applications' data plane parts are written as eBPF programs in C. The main controller and the application-specific controllers operating in user space are written in C++. We will now present the technical details of our implementation, focusing on the computation of high-level metrics, the routing system, and the example applications.

Efficient Computation of High-level Metrics. As we compute a set of high-level metrics for each received telemetry record, this computation must be efficient and not incur unnecessary overheads. Netronome SmartNICs support offloading

[1] Our code is available at: https://github.com/mcabranches/xdp-netmon

XDP programs and can be used to maintain simple statistics like counters directly on the NIC without using host CPU cycles. These counters can then be accessed by the controller through an eBPF map. Additionally, our system uses a HyperLogLog (HLL) sketch to estimate the number of unique flows per time interval. The HLL algorithm estimates the cardinality of large sets with negligible memory use [15]. HLL's main idea is that if the binary representation of an element in a set is random and uniformly distributed (e.g., a hash), the number of leftmost zeros in this representation can be used to estimate the cardinality of the set. In this manner, HLL requires the computation of the hash of a given key of interest (e.g., 5-tuple) for every packet in our system.

To reduce variance of the estimation, the hashes are divided in buckets, where the b leftmost bits of the hash value are used as the bucket index on an array. The final cardinality estimation uses the harmonic mean of the cardinality in each bucket. To find the cardinality of a bucket, the HLL algorithm gets the remainder of the hash, calculates the number of leading zeros, and checks if this number is greater than the previously recorded one for the respective bucket. If so, the bucket is updated and the HLL algorithm uses this value to estimate the new cardinality of the set. A smaller b reduces memory requirements for HLL but also reduces its accuracy. For example, by using $b = 8$, HLL can estimate the cardinality of distinct 5-tuples consuming only 768 Bytes with an accuracy of $\sim$93.5% (see [15] for details).

Performing the HLL operations is not computationally cheap, so we leverage the processing power available on a Netronome SmartNIC to accelerate them. Despite the limitations of our SmartNIC XDP offloads (i.e., inefficient and slow map update operations from the data plane [16]) we were able to design an efficient division of work between the SmartNIC and the host. In this case, all the stateless operations for the HLL are executed on the SmartNIC and the stateful ones are executed on the host. This is possible as we are able to enrich the packet metadata (using *bpf_xdp_adjust_head()*) with precomputed HLL data on the SmartNIC (bucket index and number of zeros for a given hash), and this will be carried with the packet for further processing on the host XDP layer (HLL state updates) and controller (cardinality estimation).

Routing Packets through eBPF Programs. The match+action tables required for record routing are implemented as eBPF hash maps offloaded to the SmartNIC and populated from the controller to indicate which packets should be processed by each of the applications (traffic selection). When a match occurs in each table, we enrich the packet metadata with a list of file descriptors (FD) that indicates which applications should process the packet in which order. The first field in the file descriptor list is a pointer to the FD of the application that should receive the record next. This FD is used as an index (key) in a `BPF_MAP_TYPE_PROG_ARRAY` on the host's XDP layer to get the memory address of the application to be called using the *bpf_tail_call()* function. Before jumping to the desired application, the pointer is incremented so that the router knows when the chain of applications for a packet has reached the end. After application processing, the packet is returned to the router via a tail call. The router then determines whether the packet needs to be forwarded to the next application or whether it has reached the end of the chain where a XDP action can be applied (e.g., drop, pass, etc.). Figure 4 shows an example of this mechanism where a TCP segment is sent through the traffic accounting application and the TCP half-open connection analyzer.

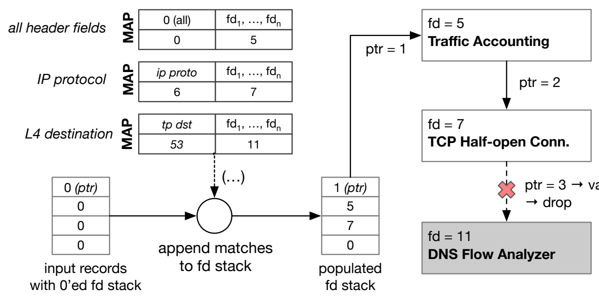


Figure 4. Example of record router for a TCP packet

Example Applications. We implemented three example applications to demonstrate the flexibility of our system. All applications consist of a kernel space (eBPF) component and a user space component. The two components communicate via eBPF maps. The user space component is a standalone application that can access the eBPF maps configured in the kernel space counterpart as soon as the kernel portion is loaded. In order to access a map, the user space application only needs to know a custom identifier string of the respective map set in the eBPF program. We will now briefly describe the applications we used in our evaluation.

Traffic Accounting. This application counts the number of Bytes and packets per destination IP address. This is useful for billing purposes, e.g., in a public cloud. The application performs a lookup and subsequent value increment in a eBPF hash table (`BPF_MAP_TYPE_HASH`) for every single record. The aggregation key and potential filtering can easily be changed to adapt the query to the operator's needs. We envision this application running at all times.

Half-open TCP Connections. A high number of half-open TCP connections are an indicator for various TCP-related attacks, in particular a SYN flood. This application keeps track of all ongoing TCP handshakes with a timestamp of the SYN segment in a hash map indexed by the IP 5-tuple. If the handshake completes, i.e., when the client sends an ACK, the previously installed state for this connection is removed. If an entry spends longer than a configurable threshold (e.g., 5s) in this map, the flow is reported as a half-open connection.

DNS Flow Analyzer. The DNS flow analyzer can be used to confirm a suspected DNS-related attack. It collects the number of packets and Bytes and the timestamp of the first packet for each DNS flow in an eBPF map. This information can be used to block flows where the request rate exceeds a threshold.

V. Evaluation

We will now evaluate the efficiency and scalability of our system by measuring CPU consumption and throughput as we vary the number of applications deployed and the number of cores assigned to the system. We also evaluate the ability of our sketch-based high-level monitor to detect an attack.

Experimental Setup. We set up two 12-core servers (Intel Xeon E5-2620v3 at 2.40GHz) with 64 GB of RAM. The first server runs our system and is equipped with a 10 Gbit/s Netronome Agilio CX SmartNIC [11]. The second server has a 10 Gbit/s Intel 82599ES NIC. The servers run kernel versions 5.8 and 5.4, respectively. We use DPDK's pktgen [8] to replay a PCAP file with telemetry records generating up to 12.5 million packets per second (Mpps) between the machines.

Efficiency in CPU Utilization as we add Applications. Our system saves resources (i.e., CPU) by leveraging shared high-level metrics that drive our monitoring application routing decisions. To evaluate this, in our first experiment we apply a conservative load on our system (2 Mpps) and run it on just one core. We also gradually increase the number of monitoring applications that each packet traverses. We can see in Figure 5 that as each packet traverses a longer chain of monitoring applications (x-axis), the average CPU utilization on the system only increases in small steps (y-axis) due to our primitive and XDP. With the SmartNIC offloads (see §IV), our system is even more efficient and able to comfortably process the applied load with all applications enabled, without ever reaching over 60% CPU utilization. This is important for energy efficiency, and also leaves more CPU cycles available for extra monitoring applications and other processes. Systems using DPDK would consume 100% CPU at all times.

Scalability in Terms of Throughput. Now we look at our system scalability in terms of throughput as we add applications. Here, to show a different perspective, we focus on the main limiting factor of throughput in our monitoring applications - the number of eBPF map accesses (i.e., lookups/writes). As we describe in Sections III and IV, most network monitoring applications interact with some state (often in a hash table) to implement their core functionality. For example, our three applications each perform up to two map accesses (lookup and/or write) for each packet.

To evaluate how the number of map accesses affects the system throughput, we gradually increase the number of map lookup/writes on an eBPF hash map for each packet. This XDP application is set to run on just one core, and we can see it as a chain of applications of variable length and of different complexities. Figure 6 shows that as we have more map lookups/writes, the system's throughput starts to degrade. By leveraging our system's primitive, monitoring applications can be turned on and off, ensuring that they will only execute when needed, which in turn allows more packets to be processed by the monitoring applications that are running at a given time.

Scalability as we add Resources. Now we show how our system throughput scales with the number of processing cores. In Figure 7, we run our system on one and two cores (one and two NIC queues) and set IRQ affinity for each queue/core. The y-axis shows the system throughput in Mpps. Our SmartNIC distributes traffic among cores based on the contents of the telemetry packet (in our case, the 5-tuple of the record). We set pktgen to send traffic at its maximum rate for our configuration (12.5 Mpps) and run the primitive alone and with each of the monitoring applications (x-axis). Here, we can see that our offloads can speed up our system by accelerating stateless operations, as we describe in Section IV. The primitive with one core has slightly higher throughput than with two cores. This is likely caused by memory access contention between multiple memory queues and our logic on the NIC. This cost is, however, amortized as the applications with two cores have higher throughput than those with one core.

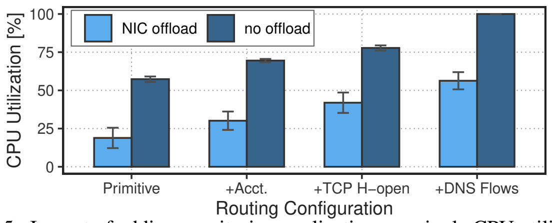


Figure 5. Impact of adding monitoring applications on single CPU utilization.

High-level Monitoring Accuracy. Finally, to show the accuracy of the HLL-based flow count estimator, we demonstrate a practical example of how our implementation was able to detect an artificially injected flooding attack. In this experiment, we used a WAN trace collected by CAIDA [17] and added an attack packet for each existing packet with probability 0.1 during a 60 second time window. The attack packet had a randomly sampled 5-tuple increasing the number of distinct flows. We computed the exact number of distinct flows seen in every 1 second time interval and used our HLL implementation with $b = 8$ to obtain an estimate. Figure 8 shows that the estimate closely follows the ground truth; it also immediately reacts to the sudden increase in flow count making this approach suitable for our use case.

VI. Related Work

The literature around network telemetry and monitoring is vast, yet few works propose solutions for parallel and dynamic queries and analytics tasks. *Flow [5] is a hardware-based telemetry system that partitions an analytics pipeline and performs only those tasks in hardware that are relevant to all queries enabling arbitrary and concurrent queries in software. Jetstream [6] complements *Flow with a fast software processor; the system, however, falls short in providing a server-side routing and dynamic orchestration mechanism for analytics tasks. NetQRE [4] allows for dynamic queries but is not designed for concurrent measurement. BeauCoup [13] is designed for dynamic and concurrent queries at switch line rate but only supports a single class of query (count-distinct).

Most work on orchestrating packet processing functions focuses on network function virtualization (NFV) for a variety of use cases, e.g., [18]–[20]. NFV systems and service chains span several hosts and generally dedicate full servers for NF processing often with all CPU resources being blocked for heavyweight packet I/O frameworks. Our system focuses specifically on network monitoring and is designed to save system resources in order to be deployed alongside other applications and services, e.g., at the network edge.

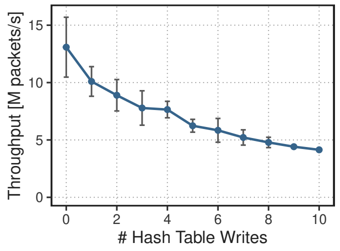


Figure 6. Performance impact of per-packet hash table lookups and writes.

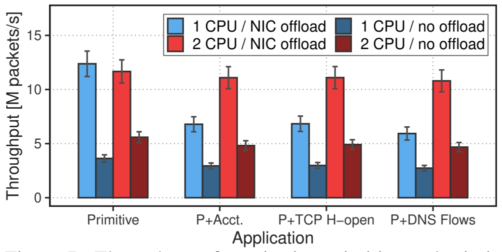


Figure 7. Throughput of monitoring primitive and primitive plus individual applications using 1 and 2 CPU cores.

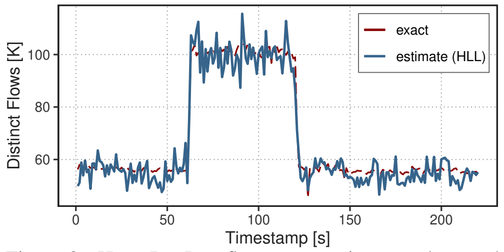


Figure 8. HyperLogLog flow count estimate and ground truth during a flooding attack.

Over the past years, eBPF and XDP have attracted significant interest in the research community as well as industry. XDP has first been presented in [10] and, since then, the technology has been used for a variety of use cases; most of them revolve around network virtualization [21], load balancing [22], or packet filtering and DDoS mitigation at end hosts [23]. For example, Cassagnes et. al. propose using XDP for DDoS detection and filtering on end hosts [24]. This system, however, also only serves this single use case and does not support orchestration of monitoring tasks.

The perhaps most related work, Polycube [25], goes beyond a single application and provides a framework for realizing general NFV service chains using XDP. Our work focuses on telemetry-based network monitoring applications and is designed and optimized for this use case. Our application router also uses tail calls to enable chaining and dynamic loading eBPF/XDP applications, but different from Polycube, routing decisions on our system can be based on shared high-level monitoring metrics. Furthermore, our work adds and evaluates offloading XDP logic to a SmartNIC.

VII. Conclusion

We presented a practical software-based network monitoring framework that significantly reduces resource consumption of network analytics by consolidating tasks relevant to all applications and triggering applications only when required. Our implementation leverages modern kernel-level packet processing capabilities improving efficiency and reducing energy consumption over existing kernel-bypass approaches.

Acknowledgements

This work has been supported by the Austrian Science Fund (FWF) D-A-CH project ADVISE (I 4800-N), 2020-2023, in part by NSF as part of CAREER award number 1652698, and the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

References

[1] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *Proc. ACM SIGCOMM*, 2017.

[2] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven Streaming Network Telemetry," in *Proc. SIGCOMM*, 2018.

[3] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller, "An overview of ip flow-based intrusion detection," *IEEE communications surveys & tutorials*, vol. 12, no. 3, pp. 343–356, 2010.

[4] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur *et al.*, "Quantitative network monitoring with netqre," in *Proc. ACM SIGCOMM*, 2017.

[5] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With *Flow," in *Proc. USENIX ATC*, 2018.

[6] O. Michel, J. Sonchack, G. Cusack, M. Nazari, E. Keller, and J. M. Smith, "Software packet-level network analytics at cloud scale," *IEEE Trans. on Network and Service Management*, vol. 18, no. 1, 2021.

[7] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: a better netflow for data centers," in *Proc. USENIX NSDI*, 2016.

[8] DPDK Project, "The data plane development kit," retrieved June 16, 2021 from https://www.dpdk.org.

[9] P. Tammana, R. Agarwal, and M. Lee, "Distributed network monitoring and debugging with switchpointer," in *Proc. USENIX NSDI*, 2018.

[10] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert *et al.*, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proc. ACM CoNEXT*, 2018.

[11] Netronome Systems Inc., "Netronome nfp-4000 flow processor product brief," retrieved June 17, 2021 from https://www.netronome.com/media/documents/PB_NFP-4000-7-20.pdf.

[12] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[13] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "Beaucoup: Answering many network traffic queries, one memory update at a time," in *Proc. ACM SIGCOMM*, 2020.

[14] P. Manzanares-Lopez, J. P. Muñoz-Gea, and J. Malgosa-Sanahuja, "Passive in-band network telemetry systems: The potential of programmable data plane on network-wide telemetry," *IEEE Access*, vol. 9, 2021.

[15] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," in *AofA: Analysis of Algorithms*, ser. DMTCS Proceedings, Jun. 2007.

[16] "Ever deeper with bpf – an update on hardware offload support - nov. 2018," retrieved July 5, 2021 from https://www.netronome.com/blog/ever-deeper-bpf-update-hardware-offload-support/.

[17] "The caida ucsd anonymized internet traces - mar. 2018," retrieved July 5, 2021 from https://www.caida.org/catalog/datasets/passive_dataset.

[18] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. USENIX NSDI*, 2012.

[19] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood *et al.*, "Nfvnice: Dynamic backpressure and scheduling for nfv service chains," in *Proc. ACM SIGCOMM*, 2017.

[20] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "Nfp: Enabling network function parallelism in nfv," in *Proc. ACM SIGCOMM*, 2017.

[21] Z. Ahmed, M. H. Alizai, and A. A. Syed, "Inkev: In-kernel distributed network virtualization for dcn," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 3, Jul. 2018.

[22] P. Enberg, A. Rao, and S. Tarkoma, "Partition-aware packet steering using xdp and ebpf for improving application-level parallelism," in *Proc. ACM ENCP*, 2019.

[23] G. Bertin, "Xdp in practice: integrating xdp in our ddos mitigation pipeline - netdev 2.1," retrieved June 28, 2021 from https://legacy.netdevconf.info/2.1/session.html?bertin.

[24] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State, "The rise of ebpf for non-intrusive performance monitoring," in *Proc. NOMS*, 2020.

[25] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu, "A framework for ebpf-based network functions in an era of microservices," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, 2021.