Feasibility of Application Layer Header Parsing in eBPF and P4

Ashwin Kumar, Abhik Bose, Khushboo Tiwari, Arnav Mishra, Abhishek Dixit, Abuhujair Khan, Mythili Vutukuru
Department of Computer Science and Engineering, Indian Institute of Technology Bombay
{ashkumar, abhik, khushboo, arnav, abhishekdixit, abuhujairkhan, mythili}@cse.iitb.ac.in
All authors contributed equally to this work.

***Abstract*—Recent advances like P4 programmable hardware switches in the network, and eBPF programs in the endhost network stack, have significantly improved the ability to customize packet processing pipelines in middleboxes and endhosts, and have enabled the offload of some simple application layer processing to the network. These programming frameworks are typically used for parsing mostly fixed-format network or transport layer headers, and are considered unsuitable for parsing complex application layer headers with variable formats. This paper characterizes the feasibility and limits of parsing complex application layer messages within the restrictive programming environments of P4 switches and eBPF kernel programs, a question that has not received much attention in prior work. We evaluate the feasibility of parsing different types of application message formats on the forwarding path using existing parsing techniques and quantify the overhead of such parsing on the application performance using our optimized implementations of the parsing techniques in eBPF and P4. We use on-path application telemetry, where application layer metrics are extracted by parsing application headers on the packet forwarding path, as a case study to evaluate our implementation. Our evaluation shows that, within the limits of feasibility, on-path network telemetry is more efficient and makes the metrics available sooner than state-of-the-art off-path telemetry systems that mirror packets and analyze them in userspace software.**

***Index Terms*—eBPF, P4, L7 parsing**

I. Introduction

Networked applications communicate with each other using complex application layer messages that are encapsulated in various standardized network protocol headers. Networking elements on the forwarding path of the packet (endhost network stacks, routers and switches, and various types of middleboxes like NATs and firewalls) parse information present in these network protocol headers and perform designated actions. Several network telemetry frameworks also analyze packet headers to compute network-level performance metrics that provide insights about the network to the operators. Two recent advances in networking, P4-programmable network switches [1] and eBPF programs [2] on the endhost network stack, have enabled flexible and efficient packet header processing on the forwarding path of the packet. P4 switches enable network operators to configure custom packet header parsing, and custom actions for packets that match certain patterns. eBPF allows similar programmability of packet processing pipeline in the endhost kernel network stack. Prior work has offloaded the functionality of various middleboxes and network telemetry frameworks to P4 enabled programmable switches [3], [4] or to eBPF programs [5], [6], [7], [8].

Given the ease of embedding custom packet processing in P4 and eBPF, recent work has also found merit in offloading various L7 applications to P4 switches [9], [10], [11] and eBPF programs [12], [13] in order to accelerate the performance of applications running on the endhosts. Parsing application layer headers is more complicated than parsing standardized network headers because the fields of application headers are often of variable width, in variable order, and sometimes optional. Parsing such headers is difficult within the P4 programming framework because the P4 parser tries to extract header information into pre-defined headers. Similarly, the eBPF verifier, which ensures that eBPF code running in the kernel does not crash the kernel itself, limits the number of conditional statements and the number of codepaths in the program to limit verification time. As a result of such restrictions, prior work has managed to offload only a small subset of the application logic, or processing of a few simple messages, to P4 switches or eBPF programs. Application layer processing that requires parsing complex application layer headers in the payload, e.g., HTTP messages with embedded JSON, has so far not been attempted within P4 switches or eBPF programs to the best of our knowledge.

In this paper, we consider the problem of parsing complex application layer messages in P4-based programmable switches and in eBPF programs on endhosts, the two common frameworks used for flexible packet processing on the forwarding path. While prior work has always assumed that such parsing is infeasible, or has a prohibitive cost, our work seeks to quantify the costs and feasible limits of application layer parsing. We believe that exploring the feasibility of such parsing is interesting for several reasons. For example, knowing the limits of feasibility could let us offload more complex application layer processing to P4 switches or eBPF programs in the future. More interestingly, the ability to easily parse application layer headers in switches or the kernel enables use cases like *on-path* application layer telemetry. Existing telemetry systems deal with the collection and analysis of network/transport layer metrics, but do not usually collect application layer metrics, due to the difficulties of parsing application headers efficiently on the packet forwarding path. Instead, application layer metrics are extracted directly within applications or are computed *off-path*, wherein packets are sampled and forwarded to a specialized analysis server, which parses and extracts useful information from the packets. However, on-path application layer telemetry is better than the other options when the extraction of such metrics is feasible on the packet forwarding path, because the metrics are available

ISBN 978-3-903176-63-8 ©2024 IFIP

in real time as compared to off-path telemetry, without the overhead of mirroring packets to the analysis server. Further, because applications can be treated as blackboxes, network operators do not need to depend on application vendors to expose suitable metrics to optimize network operations. A compelling example of on-path application layer telemetry is 5G analytics, where a mobile network operator can analyze the packets exchanged between several software components of the 5G network on the packet forwarding path, and gain real-time insights to optimize the performance of mobile networks, without depending on any information exposed by the equipment vendors [14].

We begin by analyzing the suitability of existing P4 and eBPF parsing techniques used in the context of network layer headers for parsing application layer payloads. We establish the conditions under which such parsing is feasible in P4 and eBPF for different application layer header structures. Next, using optimized implementations of application layer packet parsing algorithms in P4 and eBPF, we quantify the overhead of on-path application layer message parsing on the performance of endhost applications. We also evaluate the use case of on-path application layer telemetry performed at P4 switches or eBPF programs and compare it with state-of-the-art off-path telemetry techniques. Our results show that, when operating within feasible limits, on-path application telemetry can achieve upto 36% higher throughput than state-of-the-art off-path telemetry techniques while adding negligible delays to the packet forwarding path. The delay in obtaining the performance metrics is also several orders of magnitude lower with on-path telemetry, making it suitable for real-time applications. Our results help to clearly understand the costs and benefits of parsing application layer headers within P4 switches or eBPF programs on the packet forwarding path. The insights from our work can help in making informed decisions about offloading various types of application layer processing to P4 switches and eBPF programs, and also in realizing interesting use cases like on-path application layer telemetry.

II. Related work

While parsing application layer payload on the packet forwarding path in P4 or eBPF is challenging, some prior work has tackled this problem in the context of parsing some simple application layer requests within P4 switches or kernel eBPF code, in order to offload some userspace application processing to network switches or the kernel on the endhost, thereby accelerating the userspace application itself. P4DNS [10] and P4DDPI [15] offload parts of or complete DNS query processing onto a programmable switch using P4 [1]. AccelUpf [9] offloads parts of UPF onto programmable hardware. DeeP4R [11] offloads a firewall onto a switch. PPS [16] performs string matching on application payloads using a simplified Aho-Corasick algorithm [17] for Deep Packet Inspection applications. At the end host, BMC [12] offloads parts of memcached [18], Katran [5] offloads a load balancer, and Electrode [13] offloads parts of a distributed protocol to an eBPF program running in the kernel. All of this prior work either parses structured and well defined application layer headers like DNS [19], or they perform restrictive application payload parsing, e.g., DeeP4R only looks at the URL of a certain size. Offloading more complex parts of the application requires parsing more complex application layer messages, and our work helps quantify the limits and feasibility of such processing.

A compelling use case for parsing application layer headers on the packet forwarding path is on-path application layer telemetry. Recent research in *network telemetry* deals with several problems around collecting and aggregating network/-transport layer performance metrics in P4 switches [20], [4], [3], [21]. Apart from this, there are a plethora of eBPF [2] based tools like coroot [22], cilium [7] and deepflow [8], which perform on-path network monitoring and debugging. Conflou [23] and Langlet et al. [24] provide a framework to collect telemetry data from the network in a scalable fashion. All such prior work has usually worked with network layer or transport layer headers to extract metrics, while our work aims to quantify the feasibility of parsing application layer messages, so that future research can extend the scope of such ideas to application layer metrics as well. In contrast to on-path telemetry, off-path *network telemetry* systems use several techniques to mirror packets from switches [25], [26], [27], [28] to analysis servers, and frameworks at such servers (e.g., [29]) can currently analyse over 100 Gbps of traffic. Parsing complex application layer headers to extract application metrics is much easier in such off-path userspace software, but the overhead and delay due to mirrored packets make such systems unsuitable for real-time use cases of application telemetry.

III. Application message parsing algorithms

An application layer payload in a network packet can contain several pieces of structured or unstructured information, e.g., HTTP URL, embedded JSON or XML payload, or even well defined L7 protocol headers like the DNS protocol. In most cases, the application payload can be thought of as a list of *attributes*, e.g., each key-value pair in a JSON or XML message, each query parameter inside an HTTP URL, or each field inside an L7 protocol header could be an attribute. In order to offload application processing or to compute any application layer metrics for telemetry, one must be able to identify and extract one or more attributes of interest in the payload and take suitable action based on the value of the attribute. Without loss of generality, we will refer to the attribute of interest in the application payload as the *query attribute*. Extracting the query attribute will involve parsing the application payload of the incoming packet first to determine the *starting offset* of the query attribute in the payload and then using the *length* of that attribute to extract the value present in the attribute.

Network layer or transport layer information is present in well defined, structured headers (e.g., TCP/UDP/IP headers), wherein most required information is present at fixed offsets from the start of the corresponding header. This makes packet parsing on the critical forwarding path in hardware packet processing pipelines tractable at linerate. In contrast, for most

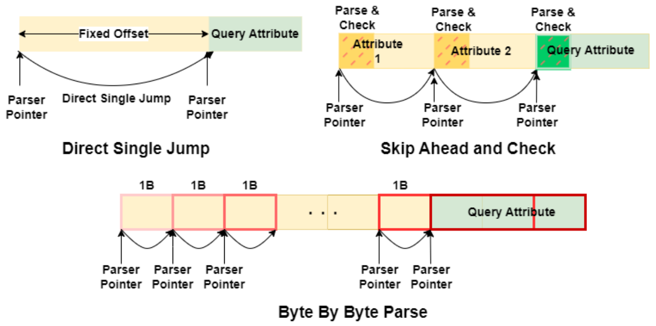


Fig. 1: Payload Parsing Techniques

application layer protocols, the query attribute may be present at variable offsets inside the application payload.

We now consider well established parsing algorithms and identify the suitability of each algorithm to parse application payloads under different constraints on the size and order of the attributes in the application payload (see Figure 1).

Direct single jump for fixed starting offset: If both the size and order of all attributes are fixed in the application payload, then the starting offset of the attribute being queried will always be a fixed value, and we can easily extract the attribute by jumping to the desired offset during packet processing. Most network and transport layer headers are parsed in this manner in hardware/software packet processing pipelines, but this simple algorithm may not be applicable for most usecases of application layer telemetry and application offloading.

Skip ahead and check with partial information: If the size of all attributes in the payload is fixed, but the attributes can appear in any order in the payload, identifying the starting offset of the query attribute will involve parsing the minimum number of bytes needed to identify every attribute, skipping ahead if it is not our required attribute, and repeating this process until we arrive on the starting offset of the query attribute. While not as efficient as the direct single jump, this parsing algorithm lets us skip parsing unnecessary bytes by leveraging information about the known set of attributes in the application payload. This technique is used to parse network/transport layer headers when there is some uncertainty in the header structure, e.g., with TCP options. This technique has also been used to parse certain application layer headers in hardware in limited settings [9].

Byte-by-byte parsing for fully unknown starting offset: If the size of attributes is not fixed, and we do not know the order in which attributes appear in the payload, or even which attributes may be present or absent, then we essentially have no information about the starting offset of the query attribute. In such cases, we have to examine each byte to determine the start location of the query attribute by having a match window slide over the entire packet one byte at a time. Such parsing is not considered suitable for linerate packet processing when parsing network and transport layer headers, but has been employed with some modifications for parsing a small number of bytes in application payloads for specific applications [11].

Note that the parsing algorithms required to parse application payload get progressively complex from a single direct jump to byte-by-byte parsing as we go from highly structured to fully unstructured application payload. While the algorithms described above are fairly intuitive and commonly employed to parse network and transport layer headers in software and hardware packet processing pipelines (albeit with slightly different names or implementations), it is not clear how well these algorithms will perform when parsing complex application layer payloads in real time on the forwarding path. To answer this question, we will first implement the above algorithms within the P4 and eBPF frameworks, solving challenges as they arise (§IV), and then use our implementations to evaluate the performance overhead and feasibility (§V).

IV. Implementation

We now describe the implementations of the various parsing techniques discussed in §III, and challenges in realising them within eBPF programs in the endhost network stack, and in P4 programmable switches. While the implementation can perform any operation of its choosing (e.g., count the number of instances when a specific value is present in the attribute, or send a response packet based on the value of the attribute) after extracting the attribute from the packet, without loss of generality, we assume that both eBPF and P4 implementations described in this section count the number of instances the query attribute matches a specific pre-defined value in the application layer payload.

A. *eBPF-based implementation*

eBPF is a framework that lets users safely inject packet processing code at various “hooks” in the Linux packet processing pipeline [2]. Before loading an eBPF program into the kernel, it is checked against a verifier to ensure that the program cannot crash the kernel. The verifier poses several restrictions on the eBPF program, e.g., loops must be bounded by a constant value, and programs must contain no more than 8192 jumps. Further, the verifier traces all possible execution paths in the program and allows only programs with under 1 million verified instructions to be loaded into the kernel. We now describe how we implement complex application layer payload parsing in eBPF within this restricted framework.

Direct single jump. We know that the query attribute is located at an offset of, say *att* bytes, so we add *att* to the pointer pointing to the start of the packet to obtain a pointer to the starting offset of the query attribute. Now we need to match the attribute with a pre-defined value and on a successful match, increment a counter. In an eBPF program, we can only match a maximum of 8 bytes at a time, as *long* data type is the longest primitive data type available in an eBPF program. For this reason, we loop for *req_match_value_length*/8 iterations matching 8 bytes from the packet at offset *att* with the required match value. It's important to note that the *req_match_value_length* must be a constant as the eBPF verifier only supports constant bounded loops [30]. This is not an unrealistic requirement as the user writing an eBPF program to do application level telemetry will be aware of the query and, in turn, the match value size, and hence can make it a constant in the program.

Byte-by-byte parsing. In the case of byte-by-byte parsing, we touch each byte in the application payload in an outer loop running for application payload length iterations. Because the loop bound should be a constant value, we loop over the largest possible application payload and break out of the loop when we try to access an offset inside the packet beyond the *data_end* pointer, which is a pointer pointing to the end of the packet. Starting at every byte offset, we try to match the application payload with the pre-defined value of the query attribute. The limitation on the number of jumps allowed in an eBPF program limits the size of the application payload and matching attribute length we can parse.

Assuming that the value for maximum application payload is 1000 bytes, our logic can support a maximum match value length of 48 bytes, as the program will contain 1000* (48/8 + 2) jumps, which is just under the maximum 8192 jumps allowed in an eBPF program.

Skip ahead and check. In this case of the skip-ahead and check parsing technique, we parse and match the minimum number of bytes needed to identify an attribute uniquely from the list of attributes present in the packet payload. We have an outer loop running for iterations equalling number of attributes present, and we break out of this loop as soon as we reach the starting offset of the required attribute. Inside this loop, we have a list of *if* statements which match the attribute at the current offset with the query attribute. On a mismatch, we increment the pointer by the size of the matched attribute (assuming it is known) such that the pointer points at the starting offset of the next attribute during the next iteration of the *for* loop. Once we identify the starting offset of the query attribute, we match the value in the attribute with a pre-defined value.

Assuming the number of attributes in the payload is N, the skip ahead and check technique has lesser number of jumps (N^2 to be exact) as compared to the case of byte-by-byte parsing, which has jumps proportional to the number of bytes being parsed. Therefore, it is possible to parse longer payloads by skipping ahead over unnecessary bytes. However, the logic of skipping ahead and checking has N^N execution paths which it may take during execution. The eBPF verifier will completely go down each individual path and verify instructions encountered along each path. The total verifiable instructions limit being 1 million, we can only support a handful of attributes before the total number of paths an eBPF program can take, multiplied by the number of instructions along each path, crosses that limit, and the verifier fails. Therefore, both parsing techniques becomes unfeasible for longer application layer headers, but for different reasons, as we show in §V.

B. P4-based implementation

P4 is a high-level language that allows us to configure a programmable *parser* in hardware to extract custom fields from network packets and *match-action* tables across multiple stages of the pipeline to match extracted fields and take suitable actions. To ensure packet processing at near line-rate, the P4 programming language and the underlying devices impose certain constraints similar to eBPF. For example, a P4 parser can process headers only in a directed acyclic graph (DAG) and cannot go back to a previously parsed header. Unlike eBPF, P4 completely lacks the ability of any loop execution. Besides the language constraints of P4, the limited on-board memory on programmable hardware imposes further restrictions on packet processing, the number of headers that can be extracted and so on.

Direct single jump. Assuming that the query attribute is at an offset of *att* bytes from the start of the application payload, the P4 parser extracts headers worth *att* bytes to reach the required attribute and then extracts the next match value length bytes into a P4 header, after which we try to match that specific P4 header with the required value. On a successful match, we increment the counter or take other suitable action.

Note that because hardware platforms impose restrictions on the number of bytes that can be extracted into a header, we can try to use multiple P4 headers to parse the packet upto *att* bytes if required. However, other hardware limitations on the total bytes available to store headers may limit the size of application payload we can parse even with this workaround. We evaluate all such limitations in §V.

Skip ahead and check. The P4 parser first extracts a small part of each attribute into a P4 header and then tries to match the extracted bytes with attributes from the attribute list of the application payload. On a successful match, we further extract the remaining bytes of the attribute until its end (assuming we know the length of each attribute in the application payload), then perform a match of the completely extracted query attribute with a pre-defined specific value and increment a counter on a successful match. If the first few extracted bytes match some other attribute which is not the query attribute, we extract the exact amount of bytes into another P4 header in order to reach the start of the next attribute in the payload, after which we repeat the same procedure until we hit the required attribute. This P4 implementation is limited by the number of parser states, which grows as N^2, where N is the number of attributes in the payload, due to the need to check for a match with all N attributes when parsing each attribute. As we end up creating more headers in the implementation of this parsing logic (albeit probably smaller in size compared to the direct single jump design), we also might end up hitting the limit on space available to store headers inside the hardware.

Byte-by-byte parsing. Because we have no information about the attributes in the payload, we extract the application payload into 1-byte sized headers until a delimiter or something signifying the end of an attribute is found. The P4 program collects these 1-byte headers into groups, transitioning from one group of headers to the other when a delimiter is found.

The packet processing pipeline then matches these groups of 1-byte headers, say H_{11},.., H_{1n} for the first attribute, H_{21}, ..., H_{2n} for the second attribute, and so on, against the required query attribute value using match-action tables, and updates suitable counters accordingly. Once again, the number of bytes of the application payload we can parse using this technique depends on the hardware limitations on parser states and header storage available.

Byte-by-byte parsing with recirculation. We also implement another variant of the byte-by-byte parsing algorithm, adapted from Deep4R [11]. In this variant, the incoming packet travels the programmable switch multiple times using recirculation, and 1 byte of the application payload is extracted during each traversal. The incoming packet is first cloned, and the cloned packet is processed byte-by-byte while the original packet passes through the switch untouched. The extracted byte is discarded at the end of the P4 pipeline to allow the parser to extract a new byte in each recirculation, thus essentially implementing a byte-by-byte parsing approach, but by using only one parser state to store the extracted byte in each round. Information about previously extracted bytes is stored as metadata in the packet, and this metadata, along with the current byte is matched using match-action tables to decide the action for the next recirculation. The key benefit of this design over the original byte-by-byte parsing is that it consumes fewer parser states and header space. However, the tradeoff is that recirculated packets consume switch bandwidth and can hurt performance at high load.

V. Evaluation

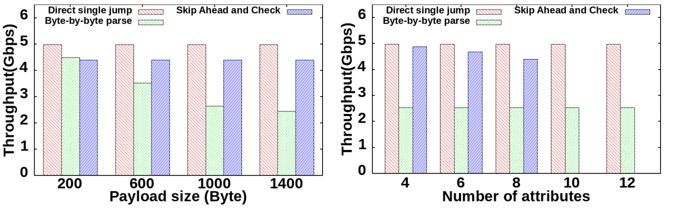
We now evaluate our implementations of the various parsing algorithms over varying sizes and structures of application payloads to analyze the feasibility and overheads of parsing application layer headers in P4 switches and eBPF programs.

Setup. We use two servers, each having 24 Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz CPUs on board, which can run 48 threads with hyperthreading enabled, on Linux kernel version 5.15. Both machines have Intel 40G QSFP+ NICs on board. We generate network traffic from one of the servers using MoonGen [31]. We run a sink application on the other server, which receives the network traffic from the load generator and selectively echoes back some of the packets for round trip latency (RTT) measurement at the load generator. To parse application payloads at the end host, we connect both servers directly without any switches in between and run our eBPF application-payload parsing code in the kernel of the destination server. To parse application payloads in P4 switches, we connect the load generator and sink server via an Intel Tofino switch (Netberg Aurora 610), which runs our application payload parsing P4 code. In addition, we also evaluate P4-based application payload parsing on a second hardware platform by running it on a Netronome Agilio 2x10 GbE SmartNIC connected to the sink server. We maintain a tolerable 0.1% packet loss across all experiments presented in this section.

Prototypes. In addition to the on-path application parsing prototypes described in §IV, we also implement an off-path monitoring application using the state-of-the-art AF_PACKET [32] packet capture library to serve as a baseline for evaluating the on-path application telemetry usecase. This monitoring application registers a packet socket, which sets up an alternate packet buffer inside the kernel, into which incoming packets are copied by the device driver. These copied packet buffers are memory mapped into the user application to save an additional packet copy and processed suitably by the monitoring application to extract application layer metrics.

Num bytes parsed		0	200	400	800	1200	1400
Throughput (Gbps)	eBPF	4.9	4.4	3.9	3.1	2.5	2.3
	in-application	4.9	4.4	3.9	3.0	2.5	2.3

TABLE I: Overhead of on-path parsing in eBPF



(a) Increasing Payload

(b) Increasing attributes

Fig. 2: Throughput for different parsing techniques

A. Overhead of on-path parsing

Parsing application payloads on the packet forwarding path increases the length of the packet processing pipeline and can potentially impact the application performance. We quantify the impact on the single-core saturation throughput and end-to-end application latency of the sink application when an eBPF program parsing application payload is running on the same CPU core. Table I shows how the saturation throughput of the application falls by around 10% as we increase the number of bytes parsed in the eBPF program from 0 to 200 bytes. We also see similar overheads of performing this parsing inside the application itself. We see a decrease in throughput by 50% in the rare case of having to parse all 1400 bytes (complete MTU payload). We found the impact to be smaller in the case of end-to-end application latency, with the additional latency added by the eBPF program being only 4us when parsing 1400 bytes compared with incurred latency in the case of no application layer parsing. We believe that a drop in throughput by 10% is acceptable when parsing application headers in eBPF because such parsing can enable application offload to eBPF and lead to further performance acceleration for the application. We experience similar overheads at higher linerate throughputs when we scale our application to multiple cores, and we skip those results in the interest of space.

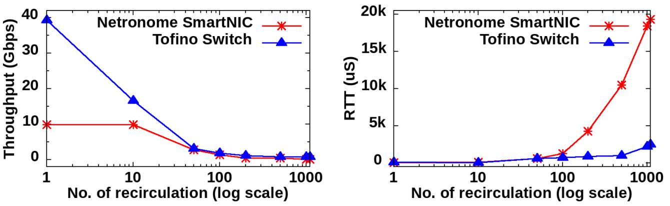
In contrast, parsing application payloads at the P4 programmable switch or smartNIC had no impact on application throughput or latency, irrespective of the number of bytes parsed, as expected with the more efficient hardware linerate processing. The only exception was that application layer P4 processing in the smartNIC, while having no impact on throughput, added a maximum of an additional 150us to the application RTT. We did not observe this impact with the more powerful Intel Tofino programmable switch platform and attribute this additional delay to the differences in the hardware architecture of the two platforms.

B. Comparison of parsing techniques

We now compare the various parsing algorithms discussed in §III with respect to their suitability to parse different types of application layer headers. Figure 2a plots the single-core saturation throughput of the sink application as a function of application payload length when an eBPF program implementing the specified parsing algorithm is running on the same core. The query attribute is present in the last 25 bytes of the application payload in all cases. As expected, the direct single jump and the skip ahead and check techniques have the least impact on the application throughput because the processing involved is lower, while the byte-by-byte parsing has the most impact. Next, Figure 2b plots the saturation throughput against increasing number of attributes in the payload while keeping the payload size fixed at 1400 bytes. Once again, the direct single jump technique performs the best, and the byte-by-byte parse technique performs the worst. The increasing number of attributes has no effect on the throughput of both techniques, as the number of bytes to be parsed remains constant. However, for the skip ahead and check technique, the number of jumps in the eBPF program increases with the number of attributes, and the eBPF verifier does not even allow the program to load beyond 8 attributes. These results demonstrate the tradeoff between the skip ahead and check and byte-by-byte parsing techniques when parsing complex application payloads. *While the skip ahead and check technique is more efficient as it skips parsing unnecessary bytes, it also requires more jumps and may become infeasible due to eBPF verifier constraints when the payload contains a large number of attributes.*

Next, we compare the various parsing algorithms on our P4-based programmable switch and Netronome setups. As long as the P4 programs compile and load on the switch or smartNIC, we found that all parsing techniques performed comparably and imposed no overhead on application performance, as expected with hardware linerate processing. However, different techniques hit different hardware limitations as application payload size and complexity increase. The direct single jump technique is limited by the maximum application layer payload parsing capability of the programmable hardware, which is 580 bytes for the Tofino-based switch and 508 bytes for the Netronome smartNIC, because of the limitations on the space available to store packet header vectors (i.e., PHV bits) on the hardware. Similarly, the byte-by-byte parsing technique is also limited by the PHV bits in the hardware, and because it has to parse more number of 1-byte headers, we found that we could parse an application payload of only 50 bytes for the switch and 85 bytes for the smartNIC. The skip ahead and check parsing technique is limited by the number of parser states (which scales as N^2 where N is the number of attributes) in addition to the PHV bits, and we are able to support an application payload containing a maximum of 9 attributes of size 12 bytes each for the switch. These results indicate that *while hardware-based on-path application telemetry in P4 switches has a lower impact on end-to-end application performance, it can parse smaller number of bytes as compared to the software-based on-path telemetry in eBPF, due to stricter hardware constraints.*

We now evaluate the variant of byte-by-byte parsing based on recirculation. Figure 3a and 3b plot the application throughput and RTT as a function of the number of application payload bytes parsed when performing recirculation based byte-by-byte parsing of the application payload. We see from the figure that the recirculation based design can parse the entire application payload, in theory, without running into limitations on parser states or PHV bits because it parses only one byte at a time. However, in practice, parsing beyond 100 bytes in this manner with recirculation leads to a steep and unacceptable drop in the application throughput on both the switch and smartNIC platforms. We also notice a corresponding steep increase in latency. These overheads are because the recirculated packets choke even the high switch bandwidth (2Tbps for our Tofino switch) when the number of recirculations is large. So, *while the recirculation based design can parse longer application payloads at low load, it also imposes an unacceptably high burden on end-to-end application performance at high load levels.*



(a) Telemetry throughput (b) Telemetry RTT

Fig. 3: Byte-by-byte parsing with number of recirculation

Num of bytes parsed		200	400	800	1200	1400
Throughput (Gbps)	eBPF	4.4	3.9	3.1	2.5	2.3
	AF_PACKET	3.2	2.6	2.4	2.2	2.0

TABLE II: Single Core Throughput: on-path vs off-path

C. On-path vs. off-path telemetry

We now compare the performance of on-path application layer telemetry, a compelling use case for parsing application headers on the packet forwarding path, to the state-of-the-art off-path telemetry implementation. We use a simple eBPF based on-path application telemetry prototype built on top of our parsing technique implementation and compare it with the off-path telemetry prototype built using the state-of-the-art AF_PACKET framework. Both telemetry systems run the simple query of counting the number of packets where the query attribute matches a specific value. Table II shows the application throughput when concurrently running one of the two telemetry methods that parse application payload (on-path eBPF and off-path AF_PACKET) as a function of the number of bytes parsed. In all cases, the metric computation logic shares CPU resources with the application itself. We see from the figure that eBPF-based on-path telemetry is significantly better, leading to 36% higher throughput when parsing 200 bytes, than off-path telemetry. However, the gap between the on-path and off-path techniques narrows as the number of bytes parsed increases. This is because AF_PACKET needs to perform a packet copy into the monitoring application and hence always touches each byte of the packet, whereas eBPF only touches the required number of bytes. Therefore, eBPF telemetry incurs lower overhead when the query attribute is present earlier on in the packet. Next, Table III plots throughput when performing on-path vs off-path telemetry as a function of the number of cores running the telemetry and

Num of cores		1	2	4	8	12
Throughput (Gbps)	eBPF	3.9	7.8	12.6	24.4	33.2
	AF_PACKET	3.2	5.3	8.9	17.5	24.4

TABLE III: Multicore scalability: on-path vs off-path

Bytes Parsed	Telemetry availability delay (in us)	
	eBPF	AF_PACKET
200	9.77	4266
1400	11.69	5767

TABLE IV: Telemetry data availability

application processing. We see from the table that eBPF-based on-path telemetry scales better than AF_PACKET based off-path telemetry, achieving close to linerate (40 Gbps) and 36% higher throughput than off-path telemetry at 12 cores, primarily due to avoiding the packet copying overheads associated with off-path telemetry.

Finally, we measure the delay, after which the extracted application layer metric is made available with both on-path and off-path telemetry techniques. Recall that our telemetry systems run a simple query counting the number of packets where a specific value is present in the query attribute in the application payload. We update this count in an eBPF map from the on-path eBPF telemetry program, monitor this map from a userspace program, and measure the delay from the time the packet arrives in the host and the metric is available in userspace. We compare this delay in obtaining the metric from on-path telemetry with the delay in mirroring the packets and computing metrics in the off-path monitoring application. Table IV shows the average delay in the availability of telemetry data of both approaches at two different numbers of bytes parsed. We see from the figure that the on-path telemetry system makes metrics available within microseconds, while off-path telemetry needs several milliseconds to copy packets to userspace and parse them. In summary, *on-path application layer telemetry is more efficient than off-path telemetry and delivers performance metrics quicker for analysis, as long as the application payload parsing lies within the feasible limits of packet processing frameworks like eBPF and P4.*

VI. Conclusion

In this paper, we study the problem of parsing complex application layer headers in the packet forwarding path, inside P4-programmable switches and kernel eBPF programs, and quantify the overheads of existing parsing techniques when applied to parsing application layer headers. We also identify several challenges when parsing complex application layer headers within the restricted programming framework of eBPF and P4, and propose solutions for the same. Our work shows that parsing application layer headers on the packet forwarding path is feasible and efficient for upto a certain size of application payload, beyond which such parsing can become infeasible due to the hardware and software constraints imposed by the P4 switch or eBPF framework. We also analyze the tradeoffs between on-path and off-path telemetry techniques and show that when performed within the realms of the feasibility of the P4 switch or eBPF framework, on-path application telemetry is more efficient than off-path telemetry, and makes performance metrics available with a much lower delay.

References

[1] M. Budiu and C. Dodd, "The p416 programming language," *ACM SIGOPS Operating Systems Review*, 2017.

[2] M. Fleming, "A thorough introduction to ebpf." https://lwn.net/Articles/740157/, 2017.

[3] 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 *ACM SIGCOMM*, 2017.

[4] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher, "Pint: Probabilistic in-band network telemetry," in *ACM SIGCOMM*, 2020.

[5] Meta, "katran." https://github.com/facebookincubator/katran, 2023.

[6] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. K. Ramakrishnan, "Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing," 2022.

[7] Cilium, "Cilium: ebpf-based networking, security, and observability." https://github.com/cilium/cilium, 2023.

[8] Deepflowio, "Application observability using ebpf." https://github.com/deepflowio/deepflow, 2023.

[9] A. Bose, S. Kirtikar, S. Chirumamilla, R. Shah, and M. Vutukuru, "Accelupf: accelerating the 5g user plane using programmable hardware," in *SOSR*, 2022.

[10] J. Woodruff, M. Ramanujam, and N. Zilberman, "P4dns: In-network dns," in *ANCS*, 2019.

[11] S. Gupta, D. Gosain, M. Kwon, and H. B. Acharya, "Deep4r: Deep packet inspection in P4 using packet recirculation," in *INFOCOM*, 2023.

[12] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller, "{BMC}: Accelerating memcached using safe in-kernel caching and pre-stack processing," in *NSDI*, 2021.

[13] Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu, "Electrode: Accelerating distributed protocols with eBPF," in *NSDI*, 2023.

[14] E. Pateromichelakis, F. Moggio, C. Mannweiler, P. Arnold, M. Shariat, M. Einhaus, Q. Wei, O. Bulakci, and A. De Domenico, "End-to-end data analytics framework for 5g architecture," *IEEE Access*, 2019.

[15] A. AlSabeh, E. Kfoury, J. Crichigno, and E. Bou-Harb, "P4ddpi: Securing p4-programmable data plane networks via dns deep packet inspection," in *NDSS*, 2022.

[16] T. Jepsen, D. Alvarez, N. Foster, C. Kim, J. Lee, M. Moshref, and R. Soulé, "Fast string searching on pisa," in *SOSR*, 2019.

[17] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, 1975.

[18] Memcached, "Memcached." https://memcached.org/, 2023.

[19] N. W. Group, "Domain names - implementation and specification." https://www.ietf.org/rfc/rfc1035.txt, 1987.

[20] p4.org, "In-band network telemetry (int) dataplane specification." https://p4.org/p4-spec/docs/INT_v2_1.pdf, 2020.

[21] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *ACM SIGCOMM*, 2018.

[22] Coroot, "Open-source observability augmented with actionable insights." https://github.com/coroot/coroot, 2023.

[23] A. Khandelwal, R. Agarwal, and I. Stoica, "Confluo: Distributed monitoring and diagnosis stack for high-speed networks," in *NSDI*, 2019.

[24] J. Langlet, R. Ben Basat, G. Oliaro, M. Mitzenmacher, M. Yu, and G. Antichi, "Direct telemetry access," in *ACM SIGCOMM*, 2023.

[25] P. Phaal, S. Panchen, and N. McKee, "Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks," tech. rep., 2001.

[26] B. Claise, "Rfc 3954: Cisco systems netflow services export version 9," 2004.

[27] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, *et al.*, "Packet-level telemetry in large datacenter networks," in *ACM SIGCOMM*, 2015.

[28] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, "Planck: Millisecond-scale monitoring and control for commodity networks," *ACM SIGCOMM CCR*, 2014.

[29] G. Wan, F. Gong, T. Barbette, and Z. Durumeric, "Retina: analyzing 100gbe traffic on commodity hardware," in *ACM SIGCOMM*, 2022.

[30] J. Corbet, "A different approach to bpf loops." https://lwn.net/Articles/877062/, 2021.

[31] P. Emmerich, "Moongen." http://scholzd.github.io/MoonGen/, 2022.

[32] M. Kerrisk, "packet(7) — linux manual page." https://man7.org/linux/man-pages/man7/packet.7.html, 2023.