dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces

Da Yu[†], Yibo Zhu[§], Behnaz Arzani[§], Rodrigo Fonseca[†], Tianrong Zhang[§], Karl Deng[§], Lihua Yuan[§]
[†]*Brown University* [§]*Microsoft*

Abstract

Distributed, in-network packet capture is still the last resort for diagnosing network problems. Despite recent advances in collecting packet traces scalably, effectively utilizing pervasive packet captures still poses important challenges. Arbitrary combinations of middleboxes which transform packet headers make it challenging to even identify the same packet across multiple hops; packet drops in the collection system create ambiguities that must be handled; the large volume of captures, and their distributed nature, make it hard to do even simple processing; and the one-off and urgent nature of problems tends to generate ad-hoc solutions that are not reusable and do not scale. In this paper we propose dShark to address these challenges. dShark allows intuitive groupings of packets across multiple traces that are robust to header transformations and capture noise, offering simple streaming data abstractions for network operators. Using dShark on production packet captures from a major cloud provider, we show that dShark makes it easy to write concise and reusable queries against distributed packet traces that solve many common problems in diagnosing complex networks. Our evaluation shows that dShark can analyze production traces with more than 10 Mpps throughput on a commodity server, and has near-linear speedup when scaling out on multiple servers.

1 Introduction

Network reliability is critical for modern distributed systems and applications. For example, an ISP outage can cause millions of users to disconnect from the Internet [45], and a small downtime in the cloud network can lead to millions of lost revenue. Despite the advances in network verification and testing schemes [18, 26, 27, 34, 44], unfortunately, network failures are still common and are unlikely to be eliminated given the scale and complexity of today's networks.

As such, diagnosis is an integral part of a network operator's job to ensure high service availability. Once a fault that cannot be automatically mitigated happens, operators must quickly analyze the root cause so that they can correct the fault. Many tools have been developed to ease this process.

We can group existing solutions into host-based [40,56,57], and in-network tools [44, 68]. While able to diagnose several problems, host-based systems are fundamentally limited in visibility, especially in cases where the problem causes packets not to arrive at the edge. On the other hand, most in-network systems are based on aggregates [32], or on strong assumptions about the topology [56]. Switch hardware improvements have also been proposed [21,28,42,56]. However, it is unlikely the commodity switches will quickly adopt these features and replace all the existing switches soon.

Because of these limitations, in today's production networks, operators have *in-network packet captures* as the last resort [50,68]. They provide a capture of a packet at each hop, allowing for gathering a full view of packets' paths through the network. Analyzing such "distributed" traces allows one to understand how a packet, a flow, or even a group of flows were affected as they traversed each switch along their path. More importantly, most, if not all, commodity switches support various packet mirroring functionalities.

In this paper, we focus on making the analysis of in-network packet captures practical. Despite the diagnosing potential, this presents many unsolved challenges. As a major cloud provider, although our developers have implemented a basic analysis pipeline similar to [68], which generates some statistics, it falls short as our networks and fault scenarios get more complicated. Multi-hop captures, middleboxes, the (sometimes) daunting volume of captures, and the inevitable loss in the capture pipeline itself make it hard for operators to identify the root problem.

The packets usually go through a combination of header transformations (VXLAN, VLAN, GRE, and others) applied repeatedly and in different orders, making it hard to even parse and count packets correctly. In addition, the packet captures, which are usually generated via switch mirroring and collector capturing, are noisy in practice. This is because the mirrored packets are normally put in the lowest priority to avoid competing with actual traffic and do not have any retransmission mechanisms. It is pretty common for the mirrored packet drop rate to be close to the real traffic drop rate being diagnosed. This calls for some customized logic that can filter out false drops due to noise.

These challenges often force our operators to abandon the statistics generated by the basic pipeline and develop ad-hoc programs to handle specific faults. This is done in haste, with little consideration for correctness guarantees, performance, or reusability, and increasing the mean time to resolution.

To address these challenges, we design dShark, a scalable packet analyzer that allows for the analysis of in-network packet traces in near real-time and at scale. dShark provides a streaming abstraction with flexible and robust grouping of packets: all instances of a single packet at one or multiple hops, and all packets of an aggregate (*e.g.*, flow) at one or multiple hops. dShark is robust to, and hides the details of, compositions of packet transformations (encapsulation, tunneling, or NAT), and noise in the capture pipeline. dShark offers flexible and programmable parsing of packets to define packets and aggregates. Finally, a query (*e.g.,* is the last hop of a packet the same as expected?) can be made against these groups of packets in a completely parallel manner.

The design of dShark is inspired by an observation that a general programming model can describe all the typical types of analysis performed by our operators or summarized in prior work [56]. Programming dShark has two parts: a declarative part, in JSON, that specifies how packets are parsed, summarized, and grouped, and an imperative part in C++ to process groups of packets. dShark programs are concise, expressive, and in languages operators are familiar with. While the execution model is essentially a windowed streaming map-reduce computation, the specification of programs is at a higher level, with the 'map' phase being highly specialized to this context: dShark's parsing is designed to make it easy to handle multiple levels of header transformations, and the grouping is flexible to enable many different types of queries. As shown in §4, a typical analysis can be described in only tens of lines of code. dShark compiles this code, links it to dShark's scalable and high-performance engine and handles the execution. With dShark, the time it takes for operators to start a specific analysis can be shortened from hours to minutes.

dShark's programming model also enables us to heavily optimize the engine performance and ensures that the optimization benefits all analyses. Not using a standard runtime, such as Spark, allows dShark to integrate closely with the trace collection infrastructure, pushing filters and parsers very close to the trace source. We evaluate dShark on packet captures of production traffic, and show that on a set of commodity servers, with four cores per server, dShark can execute typical analyses in real time, even if all servers are capturing 1500B packets at 40Gbps line rate. When digesting faster capturing or offline trace files, the throughput can be further scaled up nearly linearly with more computing resources.

We summarize our contributions as follows: 1) dShark is the first general and scalable software framework for analyzing distributed packet captures. Operators can quickly express their intended analysis logic without worrying about the details of implementation and scaling. 2) We show that dShark can handle header transformations, different aggregations, and capture noise through a concise, yet expressive declarative interface for parsing, filtering, and grouping packets. 3) We show how dShark can express 18 diverse monitoring tasks, both novel and from previous work. We implement and demonstrate dShark at scale with real traces, achieving real-time analysis throughput.

2 Motivation

dShark provides a scalable analyzer of distributed packet traces. In this section, we describe why such a system is needed to aid operators of today's networks.

2.1 Analysis of In-network Packet Traces

Prior work has shown the value of in-network packet traces for diagnosis [50, 68]. In-network packet captures are widely supported, even in production environments which contain heterogeneous and legacy switches. These traces can be described as the most detailed "logs" of a packet's journey through the network as they contain per-packet/per-switch information of what happened.

It is true that such traces can be heavyweight in practice. For this reason, researchers and practitioners have continuously searched for replacements to packet captures diagnosis, like flow records [13, 14], or tools that allow switches to "digest" traces earlier [21, 42, 56]. However, the former necessarily lose precision, for being aggregates, while the latter requires special hardware support which in many networks is not yet available. Alternatively, a number of tools [5, 20, 53] have tackled diagnosis of specific problems, such as packet drops. However, these also fail at diagnosing the more general cases that occur in practice (§3), which means that the need for traces has yet to be eliminated.

Consequently, many production networks continue to employ in-network packet capturing systems [59, 68] and enable them on-demand for diagnosis. In theory, the operators, using packet traces, can reconstruct what happened in the network. However, we found that this is not simple in practice. Next, we illustrate this using a real example.

2.2 A Motivating Example

In 2017, a customer on our cloud platform reported an unexpected TCP performance degradation on transfers to/from another cloud provider. The customer is in the business of providing real-time video surveillance and analytics service, which relies on stable network throughput. However, every few hours, the measured throughput would drop from a few Gbps to a few Kbps, which would last for several minutes, and recover by itself. The interval of the throughput drops was non-deterministic. The customer did a basic diagnosis on their end hosts (VMs) and identified that the throughput drops were caused by packet drops.

This example is representative – it is very common for network traffic to go through multiple different components beyond a single data center, and for packets to be transformed multiple times on the way. Often times our operators do not control both ends of the connections.

In this specific case (Figure 1), the customer traffic leaves the other cloud provider, X's network, goes through the ISP and reaches one of our switches that peers with the ISP (①). To provide a private network with the customer, the traffic is first tagged with a customer-specific 802.1Q label (②). Then, it is forwarded in our backbone/WAN in a VXLAN tunnel (③). Once the traffic arrives at the destination datacenter border (④), it goes through a load balancer (SLB), which uses IP-in-IP encapsulation (⑤,⑥), and is redirected to a VPN gateway, which uses GRE encapsulation (⑦, ⑧), before

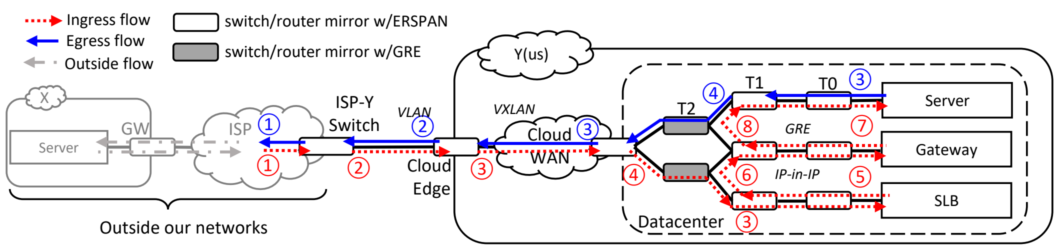

Figure 1: The example scenario. We collect per-hop traces in our network (Y and ISP-Y-switch) and do not have the traces outside our network except the ingress and egress of ISP-Y-switch. The packet format of each numbered network segment is listed in Table 1.

Number	Header Format										
	Headers Added after Mirroring			Mirrored Headers							
①	ETHERNET	IPV4	ERSPAN	ETHERNET						IPV4	TCP
②	ETHERNET	IPV4	ERSPAN	ETHERNET					802.1Q	IPV4	TCP
③	ETHERNET	IPV4	ERSPAN	ETHERNET		IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
④	ETHERNET	IPV4	GRE			IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
⑤	ETHERNET	IPV4	ERSPAN	ETHERNET	IPV4	IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
⑥	ETHERNET	IPV4	GRE		IPV4	IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
⑦	ETHERNET	IPV4	ERSPAN	ETHERNET		IPV4	GRE		ETHERNET	IPV4	TCP
⑧	ETHERNET	IPV4	GRE			IPV4	GRE		ETHERNET	IPV4	TCP

Table 1: The packet formats in the example scenario. Different switch models may add different headers before sending out the mirrored packets, which further complicates the captured packet formats.

reaching the destination server. Table 1 lists the corresponding captured packet formats. Note that beyond the differences in the encapsulation formats, different switches add different headers when mirroring packets (*e.g.*, ERSPAN vs GRE). On the return path, the traffic from the VMs on our servers is encapsulated with VXLAN, forwarded to the datacenter border, and routed back to X.

When our network operators are called up for help, they must answer two questions in a timely manner: 1) are the packets dropped in our network? If not, can they provide any pieces of evidence? 2) if yes, where do they drop? While packet drops seem to be an issue with many proposed solutions, the operators still find the diagnosis surprisingly hard in practice.

Problem 1: many existing tools fail because of their specific assumptions and limitations. As explained in §2.1, existing tools usually require 1) full access to the network including end hosts [5, 20]; 2) specific topology, like the Clos [53], or 3) special hardware features [21, 32, 42, 56]. In addition, operators often need evidence for "the problem is not because of" a certain part of the network (in this example, our network but not ISP or the other cloud network), for pruning the potential root causes. However, most of those tools are not designed to solve this challenge.

Since all these tools offer little help in this scenario, network operators have no choice but to enable in-network capturing and analyze the packet traces. Fortunately, we already deployed Everflow [68], and are able to capture per-hop traces of a portion of flows.

Problem 2: the basic trace analysis tools fall short for the complicated problems in practice. Even if network operators have complete per-hop traces, recovering what happened in the network is still a challenge. Records for the same packets spread across multiple distributed captures, and none of the well-known trace analyzers such as Wireshark [2] has the ability to join traces from multiple vantage points. Grouping them, even for the instances of a single packet across multiple hops, is surprisingly difficult, because each packet may be modified or encapsulated by middleboxes multiple times, in arbitrary combinations.

Packet capturing noise further complicates analysis. Mirrored packets can get dropped on their way to collectors or dropped by the collectors. If one just counts the packet occurrence on each hop, the real packet drops may be buried in mirrored packet drops and remain unidentified. Again, it is unclear how to address this with existing packet analyzers.

Because of these reasons, network operators resort to developing ad-hoc tools to handle specific cases, while still suffering from the capturing noise.

Problem 3: the ad-hoc solutions are inefficient and usually cannot be reused. It is clear that the above ad-hoc tools have limitations. First, because they are designed for specific cases, the header parsing and analysis logic will likely be specific. Second, since the design and implementation have to be swift (cloud customers are anxiously waiting for mitigation!), reusability, performance, and scalability will likely not be priorities. In this example, the tool developed was single threaded and thus had low throughput. Consequently, operators would capture several minutes worth of traffic and have to spend multiples of that to analyze it.

After observing these problems in a debugging session in production environment, we believe that a general, easy-to-program, scalable and high-performance in-network packet trace analyzer can bring significant benefits to network operators. It can help them understand, analyze and diagnose their network more efficiently.

3 Design Goals

Motivated by many real-life examples like the one in §2.2, we derive three design goals that we must address in order to facilitate in-network trace analysis.

3.1 Broadly Applicable for Trace Analysis

In-network packet traces are often used by operators to identify where network properties and invariants have been violated. To do so, operators typically search for abnormal behavior in the large volume of traces. For different diagnosis tasks, the logic is different.

Unfortunately, operators today rely on manual processing or ad-hoc scripts for most of the tasks. Operators must first parse the packet headers, *e.g.,* using Wireshark. After parsing, operators usually look for a few key fields, *e.g.,* 5-tuples, depending on the specific diagnosis tasks. Then they apply filters and aggregations on the key fields for deeper analysis. For example, if they want to check all the hops of a certain packet, they may filter based on the 5-tuple plus the IP id field. To check more instances and identify a consistent behavior, operators may apply similar filters many times with slightly different values, looking for abnormal behavior in each case. It is also hard to join instances of the same packet captured in different points of the network.

Except for the initial parsing, all the remaining steps vary from case to case. We find that there are four types of aggregations used by the operators. Depending on the scenario, operators may want to analyze 1) each single packet on a specific hop; 2) analyze the multi-hop trajectory of each single packet; 3) verify some packet distributions on a single switch or middlebox; or 4) analyze complicated tasks by correlating multiple packets on multiple hops. Table 2 lists diagnosis applications that are commonly used and supported by existing tools. We classify them into above four categories.

dShark must be broadly applicable for all these tasks – not only these four aggregation modes, but also support different analysis logic after grouping, *e.g.,* verifying routing properties or localizing packet drops.

3.2 Robust in the Wild

dShark must be robust to practical artifacts in the wild, especially header transformations and packet capturing noise.

Packet header transformations. As shown in §2.2, these are very common in networks, due to the deployment of various middleboxes [49]. They become one of the main obstacles for existing tools [43, 56, 69] to perform all of the diagnosis logic (listed in Table 2) in one shot. As we can see from the table, some applications need to be robust to header transformations. Therefore, dShark must correctly group the packets as if there is no header transformation. While parsing the packet is not hard (indeed, tools like Wireshark can already do that), it is unclear how operators may specify the grouping logic across different header formats. In particular, today's filtering languages are often ambiguous. For example, the "ip.src == X" statement in Wireshark display filter may match different IP layers in a VXLAN-in-IP-in-IP packet and leads to incorrect grouping results. dShark addresses this by explicitly indexing multiple occurrences of the same header type (*e.g.*, IP-in-IP), and by adding support to address the innermost ([-1]), outermost ([0]), and all ([:]) occurrences of a header type.

Packet capturing noise. We find that it is challenging to localize packet drops when there is significant packet capturing noise. We define noise here as drops of mirrored packets in the network or in the collection pipeline. Naïvely, one may just look at all copies of a packet captured on all hops, check whether the packet appears on each hop as expected. However, 1% or even higher loss in the packet captures is quite common in reality, as explained in §2.2 as well as in [61]. With the naïve approach, every hop in the network will have 1% false positive drop rate in the trace. This makes localizing any real drop rate that is comparable or less than 1% challenging because of the high false positive rate.

Therefore, for dShark, we must design a programming interface that is flexible for handling arbitrary header transformations, yet can be made robust to packet capturing noise.

3.3 Fast and Scalable

The volume of in-network trace is usually very large. dShark must be fast and scalable to analyze the trace. Below we list two performance goals for dShark.

Support real-time analysis when collocating on collectors. Recent efforts such as [68] and [50] have demonstrated that packets can be mirrored from the switches and forwarded to trace collectors. These collectors are usually commodity servers, connected via 10Gbps or 40Gbps links. Assuming each mirrored packet is 1500 bytes large, this means up to 3.33M packets per second (PPS). With high-performance network stacks [1,52,61], one CPU core is sufficient to capture at this rate. Ideally, dShark should co-locate with the collecting process, reuse the remaining CPU cores and be able to keep up with packet captures in real-time. Thus, we set this as the first performance goal – with a common CPU on a commodity server, dShark must be able to analyze *at least* 3.33 Mpps.

Be scalable. There are multiple scenarios that require higher performance from dShark: 1) there are smaller packets even though 1500 bytes is the most typical packet size in our production network. Given 40Gbps capturing rate, this means higher PPS; 2) there can be multiple trace collectors [68] and

Group pattern	Application	Analysis logic	In-nw ck. only	Header transf.	Query LOC
One packet on one hop	Loop-free detection [21] *Detect forwarding loop*	*Group:* same packet(ipv4[0].ipid, tcp[0].seq) on one hop *Query:* does the same packet appear multiple times on the same hop	No	No	8
	Overloop-free detection [69] *Detect forwarding loop involving tunnels*	*Group:* same packet(ipv4[0].ipid, tcp[0].seq) on tunnel endpoints *Query:* does the same packet appear multiple times on the same endpoint	Yes	Yes	8
One packet on multiple hops	Route detour checker *Check packet's route in failure case*	*Group:* same packet(ipv4[-1].ipid, tcp[-1].seq) on all switches *Query:* is valid detour in the recovered path(ipv4[:].ttl)	No	Yes*	49
	Route error *Detect wrong packet forwarding*	*Group:* same packet(ipv4[-1].ipid, tcp[-1].seq) on all switches *Query:* get last correct hop in the recovered path(ipv4[:].ttl)	No*	Yes*	49
	Netsight [21] *Log packet's in-network lifecycle*	*Group:* same packet(ipv4[-1].ipid, tcp[-1].seq) on all switches *Query:* recover path(ipv4[:].ttl)	No*	Yes*	47
	Hop counter [21] *Count packet's hop*	*Group:* same packet(ipv4[-1].ipid, tcp[-1].seq) on all switches *Query:* count record	No*	Yes*	6
Multiple packets on one hop	Traffic isolation checker [21] *Check whether hosts are allowed to talk*	*Group:* all packets at dst ToR(SWITCH=dst_ToR) *Query:* have prohibited host(ipv4[0].src)	No	No	11
	Middlebox(SLB, GW, etc) profiler *Check correctness/performance of middleboxes*	*Group:* same packet(ipv4[-1].ipid, tcp[-1].seq) pre/post middlebox *Query:* is middlebox correct(related fields)	Yes	Yes	18†
	Packet drops on middleboxes *Check packet drops in middleboxes*	*Group:* same packet(ipv4[-1].ipid, tcp[-1].seq) pre/post middlebox *Query:* exist ingress and egress trace	Yes	Yes	8
	Protocol bugs checker(BGP, RDMA, etc) [69] *Identify wrong implementation of protocols*	*Group:* all BGP packets at target switch(SWITCH=tar_SW) *Query:* correctness(related fields) of BGP(FLTR: tcp[-1].src|dst=179)	Yes	Yes*	23‡
	Incorrect packet modification [21] *Check packets' header modification*	*Group:* same packet(ipv4[-1].ipid, tcp[-1].seq) pre/post the modifier *Query:* is modification correct (related fields)	Yes	Yes*	4◇
	Waypoint routing checker [21, 43] *Make sure packets (not) pass a waypoint*	*Group:* all packets at waypoint switch(SWITCH=waypoint) *Query:* contain flow(ipv4[-1].src+dst, tcp[-1].src+dst) should (not) pass	Yes	No	11
	DDoS diagnosis [43] *Localize DDoS attack based on statistics*	*Group:* all packets at victim's ToR(SWITCH=vic_ToR) *Query:* statistics of flow(ipv4[-1].src+dst, tcp[-1].src+dst)	No	Yes*	18
Multiple packets on multiple hops	Congested link diagestion [43] *Find flows using congested links*	*Group:* all packets(ipv4[-1].ipid, tcp[-1].seq) pass congested link *Query:* list of flows(ipv4[-1].src+dst, tcp[-1].src+dst)	No*	Yes*	14
	Silent black hole localizer [43, 69] *Localize switches that drop all packets*	*Group:* packets with duplicate TCP(ipv4[-1].ipid, tcp[-1].seq) *Query:* get dropped hop in the recovered path(ipv4[:].ttl)	No*	Yes*	52
	Silent packet drop localizer [69] *Localize random packet drops*	*Group:* packets with duplicate TCP(ipv4[-1].ipid, tcp[-1].seq) *Query:* get dropped hop in the recovered path(ipv4[:].ttl)	No*	Yes*	52
	ECMP profiler [69] *Profile flow distribution on ECMP paths*	*Group:* all packets at ECMP ingress switches(SWITCH in ECMP) *Query:* statistics of flow(ipv4[-1].src+dst, tcp[-1].src+dst)	No*	No	18
	Traffic matrix [43] *Traffic volume between given switch pairs*	*Group:* all packets at given two switches(SWITCH in tar_SW) *Query:* total volume of overlapped flow(ipv4[-1].src+dst, tcp[-1].src+dst)	No*	Yes*	21

Table 2: We implemented 18 typical diagnosis applications in dShark. "No*" in column "in-network checking only" means this application can also be done with end-host checking with some assumptions. "Yes*" in column "header transformation" needs to be robust to header transformation in our network, but, in other environments, it might not. "ipv4[:].ttl" in the analysis logic means dShark concatenates all ivp4's TTLs in the header. It preserves order information even with header transformation. Sorting it makes path recovery possible. †We profiled SLB. ‡We focused on BGP route filter. ◇We focused on packet encapsulation.

3) for offline analysis, we hope that dShark can run faster than the packet timestamps. Therefore, dShark must horizontally scale up within one server, or scale out across multiple servers. Ideally, dShark should have near-linear speed up with more computing resources.

4 dShark Design

dShark is designed to allow for the analysis of distributed packet traces in near real time. Our goal in its design has been to allow for scalability, ease of use, generality, and robustness. In this section, we outline dShark's design and how it allows us to achieve these goals. At a high level, dShark provides a domain-specific language for expressing distributed network monitoring tasks, which runs atop a map-reduce-like infrastructure that is tightly coupled, for efficiency, with a packet capture infrastructure. The DSL primitives are designed to enable flexible filtering and grouping of packets across the network, while being robust to header transformations and capture noise that we observe in practice.

4.1 A Concrete Example

To diagnose a problem with dShark, an operator has to write two related pieces: a declarative set of trace specifications indicating relevant fields for grouping and summarizing packets; and an imperative callback function to process groups of packet summaries.

Here we show a basic example – detecting forwarding loops in the network with dShark. This means dShark must check whether or not any packets appear more than once at any switch. First, network operators can write the trace specifications as follows, in JSON:

```
{
  Summary: {
    Key: [SWITCH, ipId, seqNum],
    Additional: []
  },
  Name: {
    ipId:   ipv4[0].id,
    seqNum: tcp[0].seq
  },
  Filter: [
    [eth, ipv4, ipv4, tcp]: {    // IP-in-IP
      ipv4[0].srcIp: 10.0.0.1
    }
  ]
}
```

The first part, "Summary", specifies that the query will use three fields, *SWITCH*, *ipId* and *seqNum*. dShark builds a *packet summary* for each packet, using the variables specified in "Summary". dShark groups packets that have the same "key" fields, and shuffles them such that each group is processed by the same processor.

SWITCH, one of the only two predefined variables in dShark,[1] is the switch where the packet is captured. Transparent to operators, dShark extracts this information from the additional header/metadata (as shown in Table 1) added by packet capturing pipelines [59, 68].

Any other variable must be specified in the "Name" part, so that dShark knows how to extract the values. Note the explicit index "[0]" – this is the key for making dShark robust to header transformations. We will elaborate this in §4.3.

In addition, operators can constrain certain fields to a given value/range. In this example, we specify that if the packet is an IP-in-IP packet, we will ignore it unless its outermost source IP address is 10.0.0.1.

In our network, we assume that *ipId* and *seqNum* can identify a unique TCP packet without specifying any of the 5-tuple fields.[2] Operators can choose to specify additional fields. However, we recommend using only necessary fields for better system efficiency and being more robust to middleboxes. For example, by avoiding using 5-tuple fields, the query is robust to any NAT that does not alter *ipId*.

The other piece is a query function, in C++:

```
map<string, int> query(const vector<Group>& groups) {
  map<string, int> r = {{"loop", 0}, {"normal", 0}};
  for (const Group& group : groups) {
    group.size() > 1 ?
      (r["loop"]++) : (r["normal"]++);
  }
  return r;
}
```

The query function is written as a callback function, taking an array of groups and returning an arbitrary type: in this case, a map of string keys to integer values. This is flexible for operators – they can define custom counters like in this example, get probability distribution by counting in predefined bins, or pick out abnormal packets by adding entries into the dictionary. In the end, dShark will merge these key-value pairs from all query processor instances by unionizing all

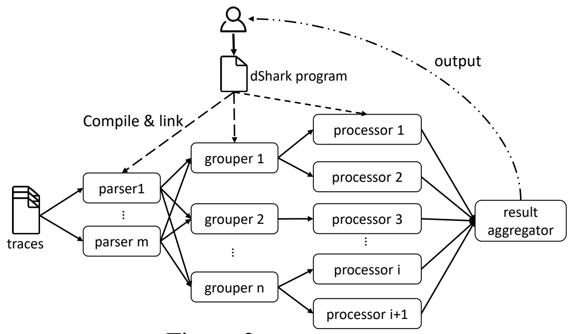

Figure 2: dShark architecture.

keys and summing the values of the same keys. Operators will get a human-readable output of the final key-value pairs.

In this example, the query logic is simple. Since each packet group contains all copies of a packet captured/mirrored by the same switch, if there exist two packet summaries in one group, a loop exists in the network. The query can optionally refer to any field defined in the summary format. We also implemented 18 typical queries from the literature and based on our experience in production networks. As shown in Table 2, even the most complicated one is only 52 lines long. For similar diagnosis tasks, operators can directly reuse or extend these query functions.

4.2 Architecture

The architecture of dShark is inspired by both how operators manually process the traces as explained in 3.1, and distributed computing engines like MapReduce [15]. Under that light, dShark can be seen as a streaming data flow system specialized for processing distributed network traces. We provide a general and easy-to-use programming model so that operators only need to focus on analysis logic without worrying about implementation or scaling.

dShark's runtime consists of three main steps: *parse*, *group* and *query* (Figure 2). Three system components handle each of the three steps above, respectively. Namely,

- *Parser:* dShark consumes network packet traces and extracts user-defined key header fields based on different user-defined header formats. Parsers send these key fields as packet summaries to groupers. The dShark parsers include recursive parsers for common network protocols, and custom ones can be easily defined.
- *Grouper:* dShark groups packet summaries that have the same values in user-defined fields. Groupers receive summaries from all parsers and create batches per group based on time windows. The resulting packet groups are then passed to the query processors.
- *Query processor:* dShark executes the query provided by users and outputs the result for final aggregation.

dShark pipeline works with two cascading MapReduce-like stages: 1) first, packet traces are (mapped to be) parsed in parallel and shuffled (or reduced) into groups; 2) query processors run analysis logic for each group (map) and finally aggregate the results (reduce). In particular, the *parser* must

[1] The other predefined variable is *TIME*, the timestamp of packet capture.

[2] In our network and common implementation, IP ID is chosen independently from TCP sequence number. This may not always be true [58].

handle header transformations as described in §3.2, and the *grouper* must support all possible packet groupings (§3.1). All three components are optimized for high performance and can run in a highly parallel manner.

Input and output to the dShark pipeline. dShark ingests packet traces and outputs aggregated analysis results to operators. dShark assumes that there is a system in place to collect traces from the network, similar to [68]. It can work with live traces when collocating with trace collectors, or run anywhere with pre-recorded traces. When trace files are used, a simple coordinator (§5.4) monitors the progress and feeds the traces to the parser in chunks based on packet timestamps. The final aggregator generates human-readable outputs as the query processors work. It creates a union of the key-value pairs and sums up values output by the processors (§5).

Programming with dShark. Operators describe their analysis logic with the programming interface provided by dShark, as explained below (§4.3). dShark compiles operators' programs into a dynamic-linked library. All parsers, groupers and query processors load it when they start, though they link to different symbols in the library. dShark chooses this architecture over script-based implementation (*e.g.,* Python or Perl) for better CPU efficiency.

4.3 dShark Programming Model

As shown in the above example, the dShark programming interface consists of two parts: 1) declarative packet trace specifications in JSON, and 2) imperative query functions (in C++). We design the specifications to be declarative to make common operations like *select*, *filter* and *group* fields in the packet headers straightforward to the operators. On the other hand, we make the query functions imperative to offer enough degrees of freedom for the operators to *define* different diagnosis logic. This approach is similar to the traditional approach in databases of embedding imperative user-defined functions in declarative SQL queries. Below we elaborate on our design rationale and on details not shown in the example above.

"Summary" in specifications. A packet summary is a byte array containing only a few key fields of a packet. We introduce packet summary for two main goals: 1) to let dShark compress the packets right after parsing while retaining the necessary information for query functions. This greatly benefits dShark's efficiency by reducing the shuffling overhead and memory usage; 2) to let groupers know which fields should be used for grouping. Thus, the description of a packet summary format consists of two lists. The first contains the fields that will be used for grouping and the second of header fields that are not used as grouping keys but are required by the query functions. The variables in both lists must be defined in the "Name" section, specifying where they are in the headers.

"Name" in specifications. Different from existing languages like Wireshark filter or BPF, dShark requires an explicit index when referencing a header, *e.g.,* "ipv4[0]" instead of simply "ipv4". This means the first IPv4 header in the packet. This is for avoiding ambiguity, since in practice a packet can have multiple layers of the same header type due to tunneling. We also adopt the Python syntax, *i.e.,* "ipv4[-1]" to mean the last (or innermost) IPv4 header, "ipv4[-2]" to mean the last but one IPv4 header, *etc*.

With such header indexes, the specifications are both robust to header transformations and explicit enough. Since the headers are essentially a stack (LIFO), using negative indexes would allow operators to focus on the end-to-end path of a packet or a specific tunnel regardless of any additional header transformation. Since network switches operate based on outer headers, using 0 or positive indexes (especially 0) allows operators to analyze switch behaviors, like routing.

"Filter" in specifications. Filters allow operators to prune the traces. This can largely improves the system efficiency if used properly. We design dShark language to support adding constraints for different types of packets. This is inspired by our observation in real life cases that operators often want to diagnose packets that are towards/from a specific middlebox. For instance, when diagnosing a specific IP-in-IP tunnel endpoint, *e.g.*, 10.0.0.1, we only care IP-in-IP packets whose source IP is 10.0.0.1 (packets after encapsulation), and common IP packets whose destination IP is 10.0.0.1 (packets before encapsulation). For convenience, dShark supports "*" as a wildcard to match any headers.

Query functions. An operator can write the query functions as a callback function that defines the analysis logic to be performed against a batch of groups. To be generally applicable for various analysis tasks, we choose to prefer language flexibility over high-level descriptive languages. Therefore, we allow operators to program any logic using the native C++ language, having as input an array of packet groups, and as output an arbitrary type. The query function is invoked at the end of time windows, with the guarantee that all packets with the same key will be processed by the same processor (the same semantics of a shuffle in MapReduce).

In the query functions, each *Group* is a vector containing a number of summaries. Within each summary, operators can directly refer the values of fields in the packet summary, *e.g., summary.ipId* is *ipId* specified in JSON. In addition, since it is in C++, operators can easily query our internal service REST APIs and get control plane metadata to help analysis, *e.g.,* getting the topology of a certain network. Of course, this should only be done per a large batch of batches to avoid a performance hit. This is a reason why we design query functions to take *a batch of* groups as input.

4.4 Support For Various Groupings

To show that our programming model is general and easy to use, we demonstrate how operators can easily specify the four different aggregation types, which we extend to *grouping* in dShark, listed in §3.1.

Single-packet single-hop grouping. This is the most basic

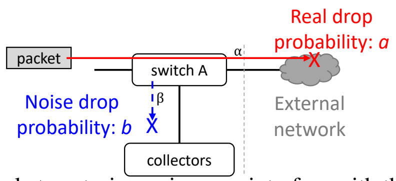

Figure 3: Packet capturing noise may interfere with the drop localization analysis.

Case	Probability	w/o E2E info	w/ E2E info
No drop	$(1-a)(1-b)$	Correct	Correct
Real drop	$a(1-b)$	Correct	Correct
Noise drop	$(1-a)b$	Incorrect	Correct
Real + Noise drop	ab	Incorrect	Incorrect

Table 3: The correctness of localizing packet drops. The two types of drops are independent because the paths are disjoint after *A*.

grouping, which is used in the example (§4.1). In packet summary format, operators simply specify the "key" as a set of fields that can uniquely identify a packet, and from which switch (*SWITCH*) the packet is collected.

Multi-packet single-hop grouping. This grouping is helpful for diagnosing middlebox behaviors. For example, in our data center, most software-based middleboxes are running on a server under a ToR switch. All packets which go into and out of the middleboxes must pass through that ToR. In this case, operators can specify the "key" as *SWITCH* and some middlebox/flow identifying fields (instead of identifying each packet in the single-packet grouping) like 5-tuple. We give more details in §6.1.

Single-packet multi-hop grouping. This can show the full path of each packet in the network. This is particularly useful for misrouting analysis, *e.g.,* does the traffic with a private destination IP range that is supposed to stay within data centers leak to WAN? For this, operators can just set packet identifying fields as the key, without *SWITCH*, and use the [-1] indexing for the innermost IP/TCP header fields. dShark will group all hops of each packet so that the query function checks whether each packet violates routing policies. The query function may have access to extra information, such as the topology, to properly verify path invariants.

Multi-packet multi-hop grouping. As explained in §3.2, loss of capture packets may impact the results of localizing packet drops, by introducing false positives. In such scenarios dShark *should* be used with multi-packet multi-hop groupings, which uses the 5-tuple and the sequence numbers as the grouping keys, without *ipId*. This has the effect of grouping together transport-level retransmissions. We next explain the rationale for this requirement.

4.5 Addressing Packet Capture Noise

To localize where packets are dropped, in theory, one could just group all hops of each packet, and then check where in the network the packet disappears from the packet captures on the way to its destination. In practice, however, we find that the noise caused by data loss in the captures themselves, *e.g.,* drops on the collectors and/or drops in the network on the way to the collector, will impact the validity of such analysis.

We elaborate this problem using the example in Figure 3 and Table 3. For ease of explanation we will refer the to paths of the mirrored packets from each switch to the collector as β type paths and the normal path of the packet as α type paths. Assume switch *A* is at the border of our network and the ground truth is that drop happens after *A*. As operators, we want to identify whether the drop happens within our network. Unfortunately, due to the noise drop, we will find *A* is dropping packets with probability *b* in the trace. If the real drop probability *a* is less than *b*, we will misblame *A*. This problem, however, can be avoided if we correlate individual packets across different hops in the network as opposed to relying on simple packet counts.

Specifically, we propose two mechanisms to help dShark avoid miss-detecting where the packet was dropped:

Verifying using the next hop(s). If the β type path dropping packets is that from a switch in the middle of the α path, assuming that the probability that *the same* packet's mirror is dropped on two β paths is small, one can find the packet traces from the next hop(s) to verify whether *A* is really the point of packet drop or not. However, this mechanism would fail in the "last hop" case, where there is no next hop in the trace. The "last hop" case is either 1) the specific switch is indeed the last on the α path, however, the packets may be dropped by the receiver host, or 2) the specific switch is the last hop before the packet goes to external networks that do not capture packet traces. Figure 3 is such a case.

Leveraging information in end-to-end transport. To address the "last hop" issue, we leverage the information provided by end-to-end transport protocols. For example, for TCP flows, we can verify a packet was dropped by counting the number of retransmissions seen for each TCP sequence number. In dShark, we can just group all packets with the same TCP sequence number across all hops together. If there is indeed a drop after *A*, the original packet and retransmitted TCP packets (captured at all hops in the internal network) will show up in the group as packets with different IP IDs, which eliminates the possibility that the duplicate sequence number is due to a routing loop. Otherwise, it is a noise drop on the β path.

This process could have false positives as the packet could be dropped both on the β and α path. This occurs with probability of only $a \times b$ – in the "last hop" cases like Figure 3, the drops on β and α path are likely to be independent since the two paths are disjoint after *A*. In practice, the capture noise *b* is $\ll 100\%$. Thus any *a* can be detected robustly.

Above, we focused on describing the process for TCP traffic as TCP is the most prominent protocol used in data center networks [6]. However, the same approach can be applied to any other reliable protocols as well. For example, QUIC [31] also adds its own sequence number in the packet header. For general UDP traffic, dShark's language also allows the operators to specify similar identifiers (if exist) based on byte offset from the start of the payload.

5 dShark Components and Implementation

We implemented dShark, including parsers, groupers and query processors, in >4K lines of C++ code. We have designed each instance of them to run in a single thread, and can easily scale out by adding more instances.

5.1 Parser

Parsers recursively identify the header stack and, if the header stack matches any in the Filter section, check the constraints on header fields. If there is no constraint found or all constraints are met, the fields in the Summary and Name sections are extracted and serialized in the form of a byte array. To reduce I/O overhead, the packet summaries are sent to the groupers in batches.

Shuffling between multiple parsers and groupers: When working with multiple groupers, to ensure grouping correctness, all parsers will have to send packet summaries that belong to the same groups to the same grouper. Therefore, parsers and groupers *shuffle* packet summaries using a consistent hashing of the "key" fields. This may result in increased network usage when the parsers and groupers are deployed across different machines. Fortunately, the amount of bandwidth required is typically very small – as shown in Table 2, common summaries are only around 10B, more than 100× smaller than an original 1500B packet.

For analyzing live captures, we closely integrate parsers with trace collectors. The raw packets are handed over to parsers via memory pointers without additional copying.

5.2 Grouper

dShark then groups summaries that have the same keys. Since the grouper does not know in advance whether or not it is safe to close its current group (groupings might be very long-lived or even perpetual), we adopt a tumbling window approach. Sizing the window presents trade-offs. For query correctness, we would like to have all the relevant summaries in the same window. However, too large of a window increases the memory requirements.

dShark uses a 3-second window – once three seconds (in packet timestamps) passed since the creation of a group, this group can be wrapped up. This is because, in our network, packets that may be grouped are typically captured within three seconds.[3] In practice, to be robust to the noise in packet capture timestamps, we use the number of packets arriving thereafter as the window size. Within three seconds, a parser with 40Gbps connection receives no more than 240M packets even if all packets are as small as 64B. Assuming that the number of groupers is the same as or more than parsers, we can use a window of 240M (or slightly more) packet summaries. This only requires several GB of memory given that most packet summaries are around 10B large (Table 2).

[3]The time for finishing TCP retransmission plus the propagation delay should still fall in three seconds.

5.3 Query Processor

The summary groups are then sent to the query processors in large batches.

Collocating groupers and query processors: To minimize the communication overhead between groupers and query processors, in our implementation processors and groupers are threads in the same process, and the summary groups are passed via memory pointers.

This is feasible because the programming model of dShark guarantees that each summary group can be processed independently, *i.e.,* the query functions can be executed completely in parallel. In our implementation, query processors are child threads spawned by groupers whenever groupers have a large enough batch of summary groups. This mitigates thread spawning overhead, compared with processing one group at one time. The analysis results of this batch of packet groups are in the form of a key-value dictionary and are sent to the result aggregator via a TCP socket. Finally, the query process thread terminates itself.

5.4 Supporting Components in Practice

Below, we elaborate some implementation details that are important for running dShark in practice.

dShark compiler. Before initiating its runtime, dShark compiles the user program. dShark generates C++ meta code from the JSON specification. Specifically, a definition of *struct Summary* will be generated based on the fields in the summary format, so that the query function has access to the value of a field by referring to *Summary.variable_name*. The template of a callback function that extracts fields will be populated using the Name section. The function will be called after the parsers identify the header stack and the pointers to the beginning of each header. The Filter section is compiled similarly. Finally, this piece of C++ code and the query function code will compile together by a standard C++ compiler and generate a dynamic link library. dShark pushes this library to all parsers, groupers and query processors.

Result aggregator. A result aggregator gathers the output from the query processors. It receives the key-value dictionaries sent by query processors and combines them by unionizing the keys and summing the values of the same keys. It then generates human-readable output for operators.

Coordinate parsers. dShark parsers consume partitioned network packet traces in parallel. In practice, this brings a synchronization problem when they process *offline* traces. If a fast parser processes packets of a few seconds ahead of a slower parser (in terms of when the packets are captured), the packets from the slower parser may fall out of grouper moving window (§5.2), leading to incorrect grouping.

To address this, we implemented a coordinator to simulate live capturing. The coordinator periodically tells all parsers

until which timestamp they should continue processing packets. The parsers will report their progress once they reach the target timestamp and wait for the next instruction. Once all parsers report completion, the coordinator sends out the next target timestamp. This guarantees that the progress of different parsers will never differ too much. To avoid stragglers, the coordinator may drop parsers that are consistently slower.
Over-provision the number of instances. Although it may be hard to accurately estimate the minimum number of instances needed (see §6) due to the different CPU overhead of various packet headers and queries, we use conservative estimation and over-provision instances. It only wastes negligible CPU cycles because we implement all components to spend CPU cycles only on demand.

6 dShark Evaluation

We used dShark for analyzing the in-network traces collected from our production networks[4]. In this section, we first present a few examples where we use dShark to check some typical network properties and invariants. Then, we evaluate the performance of dShark.

6.1 Case Study

We implement 18 typical analysis tasks using dShark (Table 2). We explain three of them in detail below.
Loop detection. To show the correctness of dShark, we perform a controlled experiment using loop detection analysis as an example. We first collected in-network packet traces (more than 10M packets) from one of our networks and verified that there is no looping packet in the trace. Then, we developed a script to inject looping packets by repeating some of the original packets with different TTLs. The script can inject with different probabilities.

We use the same code as in §4.1. Figure 4 illustrates the number of looping packets that are injected and the number of packets caught by dShark. dShark has zero false negative or false positive in this controlled experiment.
Profiling load balancers. In our data center, layer-4 software load balancers (SLB) are widely deployed under ToR switches. They receive packets with a virtual IP (VIP) as the destination and forward them to different servers (called DIP) using IP-in-IP encapsulation, based on flow-level hashing. Traffic distribution analysis of SLBs is handy for network operators to check whether the traffic is indeed balanced.

To demonstrate that dShark can easily provide this, we randomly picked a ToR switch that has an SLB under it. We deployed a rule on that switch that mirrors *all* packets that go towards a specific VIP and come out. In one hour, our collectors captured more than 30M packets in total.[5]

Our query function generates both flow counters and packet

[4]All the traces we use in evaluation are from clusters running internal services. We do not analyze our cloud customers traffic without permission.

[5]An SLB is responsible for multiple VIPs. The traffic volume can vary a lot across different VIPs.

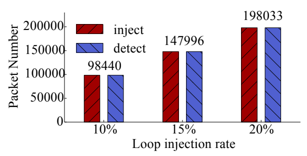

Figure 4: Injected loops are all detected.

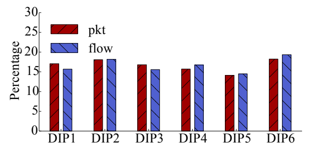

Figure 5: Traffic to an SLB VIP has been distributed to destination IPs.

counters of each DIP. Figure 5 shows the result – among the total six DIPs, DIP5 receives the least packets whereas DIP6 gets the most. Flow-level counters show a similar distribution. After discussing with operators, we conclude that for this VIP, load imbalance does exist due to imbalanced hashing, while it is still in an acceptable range.
Packet drop localizer. Noise can affect the packet drop localizer. Here we briefly evaluate the effectiveness of using transport-level retransmission information to reduce false positives (§4.5). We implemented the packet drop localizer as shown in Table 2, and used the noise mitigation mechanism described in §4.5. In a production data center, we deployed a mirroring rule on *all* switches to mirror *all* packets that originate from or go towards all servers, and fed the captured packets to dShark. We first compare our approach, which takes into account gaps in the sequence of switches, and uses retransmissions as evidence of actual drops, with a naïve approach, that just looks at the whether the last captured hop is the expected hop. Since the naïve approach does not work for drops at the last switch (including ToR and the data center boundary Tier-2 spine switches), for this comparison we only considered packets whose last recorded switch were leaf (Tier-1) switches. The naïve approach reports 5,599 suspected drops while dShark detects 7. The reason for the difference is drops of mirrored packets, which we estimated in our log to be approximately 2.2%. The drops detected by dShark are real, because they generated retransmissions with the same TCP sequence number.

Looking at all packets (and not only the ones whose traces terminate at the Tier-1 switches), we replayed the trace while randomly dropping capture packets with increasing probabilities. dShark reported 5,802, 5,801, 5,801 and 5,784 packet drops under 0%, 1%, 2% and 5% probabilities respectively. There is still a possibility that we miss the retransmitted packet, but, from the result, it is very low (0.3%).

6.2 dShark Component Performance

Next, we evaluate the performance of dShark components individually. For stress tests, we feed offline traces to dShark as fast as possible. To represent commodity servers, we use eight VMs from our public cloud platform, each has a Xeon 16-core 2.4GHz vCPU, 56GB memory and 10Gbps virtual network. Each experiment is repeated for at least five times and we report the average. We verify the speed difference between the fastest run and slowest run is within 5%.
Parser. The overhead of the parser varies based on the layers

of headers in the packets: the more layers, the longer it takes to identify the whole header stack. The number of fields being extracted and filter constraints do not matter as much.

To get the throughput of a parser, we designed a controlled evaluation. Based on the packet formats in Table 1, we generated random packet traces and fed them to parsers. Each trace has 80M packets of a given number of header layers. Common TCP packets have the fewest header layers (three – Ethernet, IPv4, and TCP). The most complicated one has eight headers, *i.e.,* ⑤ in Table 1.

Figure 6 shows that in the best case (parsing a common TCP packet), the parser can reach nearly 3.5 Mpps. The throughput decreases when the packets have more header layers. However, even in the most complicated case, a single-thread parser still achieves 2.6 Mpps throughput.

Grouper. For groupers, we find that the average number of summaries in each group is the most impacting factor to grouper performance. To show this, we test different traces in which each group will have one, two, four, or eight packets, respectively. Each trace has 80M packets.

Figure 7 shows that the grouper throughput increases when each group has more packets. This is because the grouper uses a hash table to store the groups in the moving window (§5.2). The more packets in each group, the less group entry inserts and hash collisions. In the worst case (each packet is a group by itself), the throughput of one grouper thread can still reach more than 1.2 Mpps.

Query processor. The query processor performs the query function written by network operators against each summary group. Of course, the query overhead can vary significantly depending on the operators' needs. We evaluate four typical queries that represent two main types of analysis: 1) loop detection and SLB profiler only check the size of each group (§4.1); 2) the misrouting analysis and drop localization must examine every packet in a group.

Figure 8 shows that the query throughput of the first type can reach 17 or 23 Mpps. The second type is significantly slower – the processor runs at 1.5 Mpps per thread.

6.3 End-to-End Performance

We evaluate the end-to-end performance of dShark by using a real trace with more than 640M packets collected from production networks. Unless otherwise specified, we run the loop detection example shown in §4.1.

Our first target is the throughput requirement in §3: 3.33 Mpps per server. Based on the component throughput, we start two parser instances and three grouper instances on one VM. Groupers spawn query processor threads on demand. Figure 9 shows dShark achieves 3.5 Mpps throughput. This is around three times a grouper performance (Figure 7), which means groupers run in parallel nicely. The CPU overhead is merely four CPU cores. Among them, three cores are used by groupers and query processors, while the remaining core is used by parsers. The total memory usage is around 15 GB.

On the same setup, the drop localizer query gets 3.6 Mpps with similar CPU overhead. This is because, though the query function for drop localizer is heavier, its grouping has more packets per group, leading to lighter overhead (Figure 7).

We further push the limit of dShark on a single 16-core server. We start 6 parsers and 9 groupers, and achieve 10.6 Mpps throughput with 12 out of 16 CPU cores fully occupied. This means that even if the captured traffic is comprised of 70% 64B small packets and 30% 1500B packets, dShark can still keep up with 40Gbps live capturing.

Finally, dShark must scale out across different servers. Compared to running on a single server, the additional overhead is that the shuffling phase between parsers and groupers will involve networking I/O. We find that this overhead has little impact on the performance – Figure 9 shows that when running two parsers and three groupers on each server, dShark achieves 13.2 Mpps on four servers and 26.4 Mpps on eight servers. This is close to the numbers of perfectly linear speedup 14 Mpps and 28 Mpps, respectively. On a network with full bisection bandwidth, where traffic is limited by the host access links, this is explained because we add parsers and groupers in the same proportion, and the hashing in the shuffle achieves an even distribution of traffic among them.

7 Discussion and Limitations

Complicated mappings in multi-hop packet traces. In multi-hop analysis, dShark assumes that at any switch or middlebox, there exist 1:1 mappings between input and output packets, if the packet is not dropped. This is true in most parts of our networks. However, some layer 7 middleboxes may violate this assumption. Also, IP fragmentation can also make troubles – some fragments may not carry the TCP header and break analysis that relies on TCP sequence number. Fortunately, IP fragmentation is not common in our networks because most servers use standard 1500B MTU while our switches are configured with larger MTU.

We would like to point out that it is not a unique problem of dShark. Most, if not all, state-of-art packet-based diagnosis tools are impacted by the same problem. Addressing this challenge is an interesting future direction.

Alternative implementation choices. We recognize that there are existing distributed frameworks [12,15,64] designed for big data processing and may be used for analyzing packet traces. However, we decided to implement a clean-slate design that is specifically optimized for packet trace analysis. Examples include the zero-copy data passing via pointers between parsers and trace collectors, and between groupers and query processors. Also, existing frameworks are in general heavyweight since they have unnecessary functionalities for us. That said, others may implement dShark language and programming model with less lines of code using existing frameworks, if performance is not the top priority.

Offloading to programmable hardware. Programmable hardware like P4 switches and smart NICs may offload dShark

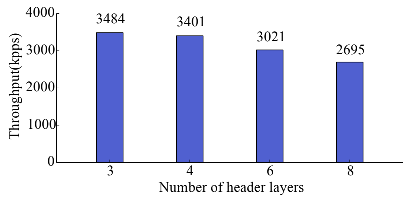

Figure 6: Single parser performance with different packet headers.

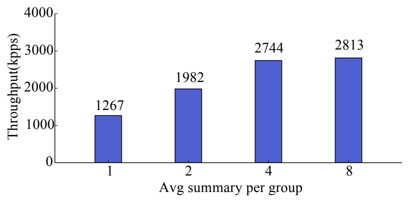

Figure 7: Single grouper performance with different average group sizes.

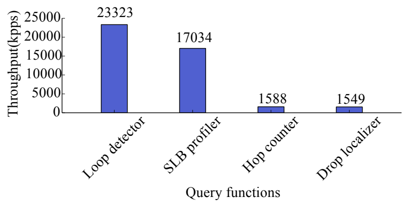

Figure 8: Single query processor performance with different query functions.

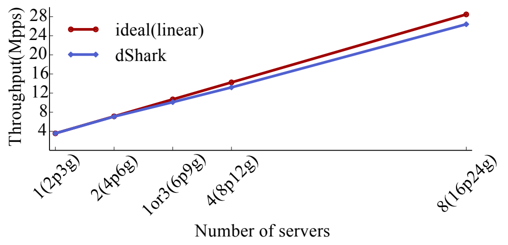

Figure 9: dShark performance scales near linearly.

from CPU for better performance. However, dShark already delivers sufficient throughput for analyzing 40Gbps online packet captures per server (§6) in a practical setting. Meanwhile, dShark, as a pure software solution, is more flexible, has lower hardware cost, and provides operators a programming interface they are familiar with. Thus, we believe that dShark satisfies the current demand of our operators. That said, in an environment that is fully deployed with highly programmable switches,[6] it is promising to explore hardware-based trace analysis like Marple [42].

8 Related Work

dShark, to the best of our knowledge, is the first framework that allows for the analysis of distributed packet traces in the face of noise, complex packet transformations, and large network traces. Perhaps the closest to dShark are PathDump [56] and SwitchPointer [57]. They diagnose problems by adding metadata to packets at each switch and analyzing them at the destination. However, this requires switch hardware modification that is not widely available in today's networks. Also, in-band data shares fate with the packets, making it hard to diagnose problems where packets do not reach the destination.

Other related work that has been devoted to detection and diagnosis of network failures includes:

Switch hardware design for telemetry [21, 28, 32, 36, 42]. While effective, these work require infrastructure changes that are challenging or even not possible due to various practical reasons. Therefore, until these capabilities are mainstream, the need to for distributed packet traces remains. Our summaries may resemble NetSight's postcards [21], but postcards are fixed, while our summaries are flexible, can handle transformations, and are tailored to the queries they serve.

Algorithms based on inference [3, 8, 19, 20, 22, 38, 40, 53, 54, 68]. A number of works use anomaly detection to find the source of failures within networks. Some attempt to cover the full topology using periodic probes [20]. However, such probing results in loss of information that often complicates detecting certain types of problems which could be easily detected using packet traces from the network itself. Other such approaches, *e.g.,* [38,40,53,54], either rely on the packet arriving endpoints and thus cannot localize packet drops, or assume specific topology. Work such as EverFlow [68] is complementary to dShark. Specifically, dShark's goal is to analyze distributed packet captures fed by Everflow. Finally, [7] can only identify the general type of a problem (network, client, server) rather than the responsible device.

Work on detecting packet drops. [11, 16, 17, 23–25, 29, 33, 37, 39, 41, 46, 60, 63, 65–67] While these work are often effective at identifying the cause of packet drops, they cannot identify other types of problems that often arise in practice *e.g.,* load imbalance. Moreover, as they lack full visibility into the network (and the application) they often are unable to identify the cause of problems for specific applications [6].

Failure resilience and prevention [4, 9, 10, 18, 27, 30, 34, 35, 47, 48, 51, 55, 62] target resilience or prevention to failures via new network architectures, protocols, and network verification. dShark is complementary to these works. While they help avoid problematic areas in the network, dShark identifies where these problems occur and their speedy resolution.

9 Conclusion

We present dShark, a general and scalable framework for analyzing in-network packet traces collected from distributed devices. dShark provides a programming model for operators to specify trace analysis logic. With this programming model, dShark can easily address complicated artifacts in real world traces, including header transformations and packet capturing noise. Our experience in implementing 18 typical diagnosis tasks shows that dShark is general and easy to use. dShark can analyze line rate packet captures and scale out to multiple servers with near-linear speedup.

Acknowledgments

We thank our shepherd, Anja Feldmann, and the anonymous reviewers for their insightful comments. Da Yu was partly funded by NSF grant CNS-1320397.

References

[1] Data plane development kit (DPDK). `http://dpdk.org/`, 2018. Accessed on 2018-01-25.

[6] Unfortunately, this can take some time before happening. In some environments, it may never happen.

[2] Wireshark. http://www.wireshark.org/, 2018. Accessed on 2018-01-25.

[3] ADAIR, K. L., LEVIS, A. P., AND HRUSKA, S. I. Expert network development environment for automating machine fault diagnosis. In *SPIE Applications and Science of Artificial Neural Networks* (1996), pp. 506–515.

[4] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., MATUS, F., PAN, R., YADAV, N., VARGHESE, G., ET AL. CONGA: Distributed congestion-aware load balancing for datacenters. *ACM SIGCOMM Computer Communication Review 44*, 4 (2014), 503–514.

[5] ARZANI, B., CIRACI, S., CHAMON, L., ZHU, Y., LIU, H., PADHYE, J., OUTHRED, G., AND LOO, B. T. Closing the network diagnostics gap with vigil. In *Proceedings of the SIGCOMM Posters and Demos* (New York, NY, USA, 2017), SIGCOMM Posters and Demos '17, ACM, pp. 40–42.

[6] ARZANI, B., CIRACI, S., CHAMON, L., ZHU, Y., LIU, H., PADHYE, J., OUTHRED, G., AND LOO, B. T. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), USENIX Association.

[7] ARZANI, B., CIRACI, S., LOO, B. T., SCHUSTER, A., AND OUTHRED, G. Taking the blame game out of data centers operations with netpoirot. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 440–453.

[8] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. *ACM SIGCOMM Computer Communication Review 37*, 4 (2007), 13–24.

[9] BODÍK, P., MENACHE, I., CHOWDHURY, M., MANI, P., MALTZ, D. A., AND STOICA, I. Surviving failures in bandwidth-constrained datacenters. In *ACM SIGCOMM* (2012), pp. 431–442.

[10] CHEN, G., LU, Y., MENG, Y., LI, B., TAN, K., PEI, D., CHENG, P., LUO, L. L., XIONG, Y., WANG, X., ET AL. Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers. In *USENIX ATC* (2016).

[11] CHEN, Y., BINDEL, D., SONG, H., AND KATZ, R. H. An algebraic approach to practical and scalable overlay network monitoring. *ACM SIGCOMM Computer Communication Review 34*, 4 (2004), 55–66.

[12] CHOTHIA, Z., LIAGOURIS, J., DIMITROVA, D., AND ROSCOE, T. Online reconstruction of structural information from datacenter logs. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 344–358.

[13] CLAISE, B., TRAMMELL, B., AND AITKEN, P. RFC7011: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. https://tools.ietf.org/html/rfc7011, Sept. 2013.

[14] CLAISE, B., E. RFC3954: Cisco Systems NetFlow Services Export Version 9. https://tools.ietf.org/html/rfc3954, Oct. 2004.

[15] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[16] DUFFIELD, N. Network tomography of binary network performance characteristics. *IEEE Transactions on Information Theory 52*, 12 (2006), 5373–5388.

[17] DUFFIELD, N. G., ARYA, V., BELLINO, R., FRIEDMAN, T., HOROWITZ, J., TOWSLEY, D., AND TURLETTI, T. Network tomography from aggregate loss reports. *Performance Evaluation 62*, 1 (2005), 147–163.

[18] FOGEL, A., FUNG, S., PEDROSA, L., WALRAED-SULLIVAN, M., GOVINDAN, R., MAHAJAN, R., AND MILLSTEIN, T. A general approach to network configuration analysis. In *NSDI, 12th USENIX Symposium on Networked Systems Design and Implementation* (2015), USENIX.

[19] GHASEMI, M., BENSON, T., AND REXFORD, J. RINC: Real-time Inference-based Network diagnosis in the Cloud. Tech. rep., Princeton University, 2015. https://www.cs.princeton.edu/research/techreps/TR-975-14.

[20] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., ET AL. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM* (2015), pp. 139–152.

[21] HANDIGOL, N., HELLER, B., JEYAKUMAR, V., MAZIÈRES, D., AND MCKEOWN, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 71–85.

[22] HELLER, B., SCOTT, C., MCKEOWN, N., SHENKER, S., WUNDSAM, A., ZENG, H., WHITLOCK, S., JEYAKUMAR, V., HANDIGOL, N., MCCAULEY, J., ET AL. Leveraging SDN layering to systematically troubleshoot networks. In *ACM SIGCOMM HotSDN* (2013), pp. 37–42.

[23] HERODOTOU, H., DING, B., BALAKRISHNAN, S., OUTHRED, G., AND FITTER, P. Scalable near real-time failure localization of data center networks. In *ACM KDD* (2014), pp. 1689–1698.

[24] HUANG, Y., FEAMSTER, N., AND TEIXEIRA, R. Practical issues with using network tomography for fault diagnosis. *ACM SIGCOMM Computer Communication Review 38*, 5 (2008), 53–58.

[25] KANDULA, S., KATABI, D., AND VASSEUR, J.-P. Shrink: A tool for failure diagnosis in IP networks. In *ACM SIGCOMM MineNet* (2005), pp. 173–178.

[26] KAZEMIAN, P., CHAN, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013), pp. 99–111.

[27] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *NSDI* (2012), vol. 12, pp. 113–126.

[28] KIM, C., PARAG BHIDE, E. D., HOLBROOK, H., GHANWANI, A., DALY, D., HIRA, M., AND DAVIE, B. In-band Network Telemetry (INT). https://p4.org/assets/INT-current-spec.pdf, June 2016.

[29] KOMPELLA, R. R., YATES, J., GREENBERG, A., AND SNOEREN, A. C. IP fault localization via risk modeling. In *USENIX NSDI* (2005), pp. 57–70.

[30] KUŹNIAR, M., PEREŠÍNI, P., VASIĆ, N., CANINI, M., AND KOSTIĆ, D. Automatic failure recovery for software-defined networks. In *ACM SIGCOMM HotSDN* (2013), pp. 159–160.

[31] LANGLEY, A., RIDDOCH, A., WILK, A., VICENTE, A., KRASIC, C., ZHANG, D., YANG, F., KOURANOV, F., SWETT, I., IYENGAR, J., BAILEY, J., DORFMAN, J., ROSKIND, J., KULIK, J., WESTIN, P., TENNETI, R., SHADE, R., HAMILTON, R., VASILIEV, V., CHANG, W.-T., AND SHI, Z. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 183–196.

[32] LI, Y., MIAO, R., KIM, C., AND YU, M. FlowRadar: A better NetFlow for data centers. In *USENIX NSDI* (2016), pp. 311–324.

[33] LIU, C., HE, T., SWAMI, A., TOWSLEY, D., SALONIDIS, T., AND LEUNG, K. K. Measurement design framework for network tomography using fisher information. *ITA AFM* (2013).

[34] LIU, H. H., ZHU, Y., PADHYE, J., CAO, J., TALLAPRAGADA, S., LOPES, N. P., RYBALCHENKO, A., LU, G., AND YUAN, L. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 599–613.

[35] Liu, J., Panda, A., Singla, A., Godfrey, B., Schapira, M., and Shenker, S. Ensuring connectivity via data plane mechanisms. In *USENIX NSDI* (2013), pp. 113–126.

[36] Liú, Y., Miao, R., Kim, C., and Yuú, M. LossRadar: Fast detection of lost packets in data center networks. In *ACM CoNEXT* (2016), pp. 481–495.

[37] Ma, L., He, T., Swami, A., Towsley, D., Leung, K. K., and Lowe, J. Node failure localization via network tomography. In *ACM SIGCOMM IMC* (2014), pp. 195–208.

[38] Mahajan, R., Spring, N., Wetherall, D., and Anderson, T. User-level internet path diagnosis. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 106–119.

[39] Mathis, M., Heffner, J., O'Neil, P., and Siemsen, P. Pathdiag: Automated TCP diagnosis. In *PAM* (2008), pp. 152–161.

[40] Moshref, M., Yu, M., Govindan, R., and Vahdat, A. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 129–143.

[41] Mysore, R. N., Mahajan, R., Vahdat, A., and Varghese, G. Gestalt: Fast, unified fault localization for networked systems. In *USENIX ATC* (2014), pp. 255–267.

[42] Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., and Kim, C. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 85–98.

[43] Narayana, S., Tahmasbi, M., Rexford, J., and Walker, D. Compiling path queries. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 207–222.

[44] Nelson, T., Yu, D., Li, Y., Fonseca, R., and Krishnamurthi, S. Simon: Scriptable interactive monitoring for sdns. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (New York, NY, USA, 2015), SOSR '15, ACM, pp. 19:1–19:7.

[45] Newman, L. H. How a tiny error shut off the internet for parts of the us. *Wired* (Nov 2017). Accessed Jan 1st, 2018.

[46] Ogino, N., Kitahara, T., Arakawa, S., Hasegawa, G., and Murata, M. Decentralized boolean network tomography based on network partitioning. In *IEEE/IFIP NOMS* (2016), pp. 162–170.

[47] Paasch, C., and Bonaventure, O. Multipath TCP. *Communications of the ACM 57*, 4 (2014), 51–57.

[48] Ports, D. R. K., Li, J., Liu, V., Sharma, N. K., and Krishnamurthy, A. Designing distributed systems using approximate synchrony in data center networks. In *USENIX NSDI* (2015), pp. 43–57.

[49] Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchene, F., Bonaventure, O., and Handley, M. How hard can it be? designing and implementing a deployable multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX Association, pp. 399–412.

[50] Rasley, J., Stephens, B., Dixon, C., Rozner, E., Felter, W., Agarwal, K., Carter, J., and Fonseca, R. Planck: Millisecond-scale monitoring and control for commodity networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 407–418.

[51] Reitblatt, M., Canini, M., Guha, A., and Foster, N. Fattire: Declarative fault tolerance for software-defined networks. In *ACM SIGCOMM HotSDN* (2013), pp. 109–114.

[52] Rizzo, L. Netmap: a novel framework for fast packet i/o. In *USENIX ATC* (2012).

[53] Roy, A., Bagga, J., Zeng, H., and Sneoren, A. Passive realtime datacenter fault detection. *ACM NSDI* (2017).

[54] Roy, A., Bagga, J., Zeng, H., and Sneoren, A. Passive realtime datacenter fault detection. In *ACM NSDI* (2017).

[55] Schiff, L., Schmid, S., and Canini, M. Ground control to major faults: Towards a fault tolerant and adaptive SDN control network. In *IEEE/IFIP DSN* (2016), pp. 90–96.

[56] Tammana, P., Agarwal, R., and Lee, M. Simplifying datacenter network debugging with pathdump. In *OSDI* (2016), pp. 233–248.

[57] Tammana, P., Agarwal, R., and Lee, M. Distributed network monitoring and debugging with switchpointer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), USENIX Association, pp. 453–456.

[58] Touch, J. RFC6864: Updated Specification of the IPv4 ID Field. `https://tools.ietf.org/html/rfc6864`, Feb. 2013.

[59] Wang, M., Li, B. L., and Li, Z. sFlow: Towards resource-efficient and agile service federation in service overlay networks. In *IEEE ICDCS* (2004), pp. 628–635.

[60] Widanapathirana, C., Li, J., Sekercioglu, Y. A., Ivanovich, M., and Fitzpatrick, P. Intelligent automated diagnosis of client device bottlenecks in private clouds. In *IEEE UCC* (2011), pp. 261–266.

[61] Wu, W., and DeMar, P. Wirecap: A novel packet capture engine for commodity nics in high-speed networks. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (New York, NY, USA, 2014), IMC '14, ACM, pp. 395–406.

[62] Wundsam, A., Mehmood, A., Feldmann, A., and Maennel, O. Network troubleshooting with mirror VNets. In *IEEE GLOBECOM* (2010), pp. 283–287.

[63] Yu, M., Greenberg, A. G., Maltz, D. A., Rexford, J., Yuan, L., Kandula, S., and Kim, C. Profiling network performance for multi-tier data center applications. In *USENIX NSDI* (2011).

[64] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. Apache spark: A unified engine for big data processing. *Commun. ACM 59*, 11 (Oct. 2016), 56–65.

[65] Zhang, Y., Breslau, L., Paxson, V., and Shenker, S. On the characteristics and origins of internet flow rates. *ACM SIGCOMM Computer Communication Review 32*, 4 (2002), 309–322.

[66] Zhang, Y., Roughan, M., Willinger, W., and Qiu, L. Spatio-temporal compressive sensing and internet traffic matrices. *ACM SIGCOMM Computer Communication Review 39*, 4 (2009), 267–278.

[67] Zhao, Y., Chen, Y., and Bindel, D. Towards unbiased end-to-end network diagnosis. *ACM SIGCOMM Computer Communication Review 36*, 4 (2006), 219–230.

[68] Zhu, Y., Kang, N., Cao, J., Greenberg, A., Lu, G., Mahajan, R., Maltz, D., Yuan, L., Zhang, M., Zhao, B. Y., et al. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM* (2015), pp. 479–491.

[69] Zhu, Y., Kang, N., Cao, J., Greenberg, A., Lu, G., Mahajan, R., Maltz, D., Yuan, L., Zhang, M., Zhao, B. Y., and Zheng, H. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 479–491.