SwitchScope: A View from the Inside

Anonymous Author(s)

ABSTRACT

As modern switches become increasingly powerful, flexible, and even programmable, network operators have an ever greater need to monitor the behavior of their switches. Many existing systems provide the ability to observe and analyze traffic by treating switches as a black box, but do not provide visibility into the experience of packets within the switch. To fill this gap, we present SwitchScope, a telemetry system that illuminates this black box and enables network operators to ask a suite of useful queries about the switch’s behavior of modifying, dropping, and forwarding packets in an intuitive and powerful Spark-like dataflow language. To minimize the overhead of SwitchScope on switch metadata, the SwitchScope compiler uses a “tag little, compute early” approach that tags packets with metadata as they move through the switch pipeline, and computes queries as early as possible to free up metadata for later processing. SwitchScope also combines information from the ingress and egress pipelines to answer aggregate queries about packets dropped due to a full queue.

1 INTRODUCTION

Network monitoring is crucial to ensuring high availability and performance in modern networks. At the core of these networks lies a set of switches, which are responsible for delivering data packets and enforcing network policies such as load balancing, access control, and attack detection. Switches themselves are intricate hardware with multiple pipelines for processing packets, memory for storing state, and complex logic for implementing network policies. However, today’s network operators have limited visibility into the data planes of these switches.

When issues occur with routing or application performance, switches are often to blame [5, 6]. For example, a switch might have incorrect forwarding rules that cause packets to never reach their intended destination. If a switch’s buffers become congested, flows will start to experience latency, which can severely impact latency-sensitive applications such as video streaming or gaming. Finally, attacks on network devices can get past switches that are not filtering packets properly, and network attacks like a DDoS can completely fill switch buffers and significantly disrupt the network.

A new generation of switch hardware [3] allows network operators to write custom packet-processing code in languages like P4 [2]. These programmable switches give network operators much greater flexibility over packet processing, which can lead to more efficient network design and greater insight into network performance. However, this advancement in switch design comes with the risk of introducing bugs in a switch’s processing, either due to programmer error or compilers that contain bugs. Thus, as programmable switch adoption rises, it will become increasingly important to monitor the processing that occurs within switches to ensure they are behaving as expected.

Each of these examples introduce questions a network operator might want to ask of the switch. For example, to detect incorrect forwarding behavior such as a black hole, one can query for the counts of packets being forwarded out each output port to observe whether any ports are not sending any packets out. To determine when a switch is experiencing congestion, one can ask about the size of the queue when a packet enters or leaves the queue. To detect attacks that target an internal host’s software, such as an SSH exploit, operators can ask whether packets which should have been dropped by an access control list (ACL) rule (such as inbound SSH connections) were instead forwarded out.

Recent telemetry systems for programmable switches support queries written in a dataflow programming model [4], a powerful and familiar language for expressing computation on a stream of incoming packets [7, 12]. However, these systems only operate on the packets as they enter the switch, and they ignore packet processing done by the switch itself. They are useful for detecting network-level attacks, but their limited expressivity does not allow operators to analyze a switch’s internal processing.

In this paper, we present SwitchScope, a network telemetry system capable of answering queries about internal switch processing using recent advances in programmable switches. SwitchScope expands on the dataflow programming model for packet streams, and allows network operators to reason about how a switch modifies, drops, delays, and loses packets. In particular, we:

- Create dataflow constructs for monitoring packets at the ingress and egress pipelines, and for collecting aggregate statistics about packets lost due to a full queue;
- Compile queries by tagging each packet with the relevant metadata about its journey through the switch, while computing statistics as early in packet processing as possible, to minimize overhead; and
• Overcome limitations in switch programmability to monitor queuing loss by introducing a hybrid switch-controller solution, by joining and synchronizing traffic counts from the ingress and egress pipelines.

2 UNDER THE SWITCH’S HOOD:
Packets’ Life Cycle in PISA

To motivate the design of SwitchScope, we first describe the life of packets as they traverse a programmable switch that SwitchScope runs on, and introduce the relevant portions of a switch’s data plane that we want to monitor.

Many modern programmable switches follow the Protocol Independent Switch Architecture (PISA) model, which consists of a series of processing stages in one or more pipelines [3]. At each stage, a series of match-action units (MAUs) trigger to modify fields in a packet header, store data in memory or to forward or drop the packet, subject to constraints on processing and memory access in each stage. We assume the FIFO switch model in Figure 1, with 3 components of switch processing: an ingress pipeline, a queuing system, and an egress pipeline.

Ingress Parsing. When a packet first enters the switch, the switch parses the packet and extracts its header values based on its configured parser. These header values, along with metadata such as the timestamp and port the packet arrived at, become part of a packet header vector (PHV) that represents the packet as it moves through the switch.

Ingress Processing. The packet then passes through the ingress pipeline. The most important task of the switch during ingress processing is forwarding, or to decide which port to send the packet out in order for the packet to reach its intended destination. In addition, the switch may modify fields in the packet header, or decide to drop the packet if it matches an access control list (ACL) rule that prevents the packet from being forwarded. If the packet is meant to be forwarded, the switch sets the output port that it intends to send the packet out on. During processing, the switch may also add custom metadata to the PHV that can be used in later stages.

Queuing. The packet then reaches the end of ingress processing, and it is placed into a queue for the appropriate output port. If the packet was instead marked for drop, the packet is now dropped. If the queue does not have space, the packet is also dropped.

Egress Processing. During egress processing, the switch may modify more header fields or decide to drop the packet. Upon reaching the end of egress processing without being marked for drop, a packet is forwarded onward toward its destination.

3 Enabling Packet Life Cycle Queries

In this section, we first explain the declarative, dataflow programming model for network telemetry queries. We then describe SwitchScope’s query fields that represent the packet’s experience within the switch and the tuple types that queries operate on, and demonstrate example queries for monitoring modifications, ACL drops and queuing loss. Finally, we introduce a loss operator for reasoning about queuing loss.

3.1 Dataflow Programming Model

Several PISA-based telemetry systems let network operators express declarative queries treating packets as tuples. Two such systems, Sonata [7] and Marple [12], provide an abstraction similar to the dataflow programming paradigm used by Apache Spark [17]. These languages operate on streams of incoming tuple data, where each tuple represents a packet header vector. Queries can then apply map, filter, groupby, and reduce operations to evaluate expressions, filter, and aggregate data, respectively, on the set of incoming tuples.

Dataflow programming is a generally expressive model for common network telemetry queries, but existing systems have limitations that prevent them from analyzing a packet’s experience within the switch. Sonata’s tuples represent packets as they appear on arrival at the switch, and thus queries cannot reason about header modifications, queuing delay, or packet loss. Marple tuples include information about queuing, but cannot track modifications or dropped/lost packets without monitoring at multiple switches or sending copies of all packets to a central query processor. With SwitchScope, we avoid costly network overhead by analyzing a packet’s experience directly on the switch being monitored.
3.2 Packet Life Cycle: Fields in Tuples

To support queries about the packet life cycle, we must expand the tuple abstraction to include fields that capture the packet’s experience at various stages of processing. Figure 1 shows the full set of fields and their locations.

When the packet first enters the switch and is parsed, the switch gets information about headers_in (the packet’s initial header fields), port_in (the port the packet arrived at), and time_in (the timestamp when the packet arrived). The packet then moves through ingress processing, where its headers may be modified, and it will either (i) have its intended output port set, or (ii) be marked for drop because it matched an ACL rule. After ingress processing finishes, headers_mid defines the packet’s headers after any ingress modifications, while port_intent refers to the packet’s intended output port, if it was not marked for drop.

What if the packet was marked for drop during ingress processing? First, we define port_intent = -1 to indicate an ACL drop on ingress. Second, we define that a packet has “finished” processing in a pipeline when (i) there is no more processing to do, or (ii) the packet is marked for drop. Thus, for dropped packets, headers_mid represents the packet’s headers at the time it was marked for drop.

Otherwise, the packet attempts to enter the queue. For now we assume the queue has space, so the packet enters the queue, and eventually the packet is dequeued and enters egress processing. At this point, the switch knows about the packet’s experience in the queue: queuing.time_in/out and queuing.len_in/out, the times the packet entered and exited the queue and the size of the queue at those times. The packet then enters egress processing, where it may undergo more modifications or be marked for drop. After the packet has finished being processed by the egress pipeline, its header_out values are known as well as port_out, the port the packet is sent out. Similar to ingress processing, if the packet is marked for drop during egress processing, port_out = -1 and headers_out represents the packet’s headers at the time it was marked.

3.3 Queries Over Ingress and Egress Tuples

Each query in SwitchScope must begin by defining a stream of tuples for that query to operate on. To motivate the choice of tuple streams we provide, we first consider two alternate approaches. We could provide a single tuple that contains all fields, similar to prior work, but a single tuple does not allow queries to specify whether they observe packets at the ingress or egress pipeline, which is important when dealing with queuing loss. We could provide four tuple types, start/end of ingress and start/end of egress, but the start and end of each pipeline are redundant because packets that start a pipeline always reach the end.

<table>
<thead>
<tr>
<th>Type</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mods</td>
<td>.egress()</td>
</tr>
<tr>
<td></td>
<td>.filter(ipv4.srcIP_in != ipv4.srcIP_out)</td>
</tr>
<tr>
<td></td>
<td>.map((ipv4.srcIP_in) =&gt; 1)</td>
</tr>
<tr>
<td></td>
<td>.reduce(keys=(ipv4.srcIP_in), func=sum)</td>
</tr>
<tr>
<td>ACL drop</td>
<td>.ingress()</td>
</tr>
<tr>
<td></td>
<td>.filter(port_intent == -1)</td>
</tr>
<tr>
<td></td>
<td>.map((ipv4.srcIP_in) =&gt; 1)</td>
</tr>
<tr>
<td></td>
<td>.reduce(keys=(ipv4.srcIP_in), func=sum)</td>
</tr>
<tr>
<td>Delay</td>
<td>.egress()</td>
</tr>
<tr>
<td></td>
<td>.filter(queue.len_out &gt; Th)</td>
</tr>
<tr>
<td></td>
<td>.map((ipv4.srcIP_in) =&gt; 1)</td>
</tr>
<tr>
<td></td>
<td>.reduce(keys=(ipv4.srcIP_in), func=sum)</td>
</tr>
<tr>
<td>Loss</td>
<td>.ingress()</td>
</tr>
<tr>
<td></td>
<td>.filter(tcp.dstPort_in == 80)</td>
</tr>
<tr>
<td></td>
<td>.lost([ipv4.srcIP_in], 20ms)</td>
</tr>
</tbody>
</table>

Table 1: Example SwitchScope queries.

Thus, SwitchScope provides a favorable trade-off between these two extremes by providing two tuple streams that queries can operate on: ingress() and egress(). Ingress() queries operate on all packets that are seen by the switch, and their tuples contain:

(headers_in, headers_mid, port_in, port_intent, time_in)

Egress() queries operate on all packets that reach egress processing, and their tuples contain:

(headers_in, headers_mid, headers_out, port_in, port_intent, port_out, time_in, queuing.time_in/out, queuing.len_in/out)

Note that while each tuple type defines a single pipeline, the queries might be compiled to either the start or end of that pipeline, depending on the fields used in each query and the resource constraints of the switch, as discussed in §4.1.

SwitchScope’s language targets four types of queries about the life of packets in the switch: (i) packet modifications, (ii) access control list drops, (iii) queuing delay, and (iv) queuing loss. Table 1 showcases example queries targeting each.

3.4 Aggregate Queries over Queuing Loss

We now handle the special case of queuing loss. Consider the strawman approach of introducing a third tuple type for “lost” packets, that produces a tuple for each packet lost due to a full queue. Unlike ACL drops, the switch does not provide a programmable hook for analyzing the packet when it attempts to enter a full queue, as this occurs outside the
programmable pipelines. This means that to detect queuing loss, we must somehow observe each packet that appears at ingress processing, but never reaches egress processing even after accounting for possible queuing delay.

An expensive option would be to forward each packet at ingress and egress to a central query processor for analysis. Alternatively, the switch could keep state about packets seen at ingress and at egress, and later a central query processor could compare the per-packet state. It would be too expensive to store state for every packet the switch sees, but it is possible to keep aggregate counts—for instance, packet counts grouped by /8 IP subnets—in both pipelines.

SwitchScope provides a special operator for tracking packets lost to the queue:

```
.lost(groupby_fields, epoch_ms)
```

which computes counts of lost packets grouped by the specified fields. We place two special restrictions on using .lost() in a query:

- Queries with .lost() can only operate on ingress tuples, as by definition, lost packets never enter egress processing.
- The aggregate operator .reduce is not allowed before .lost(), but simple operators (.map, .filter) are allowed.

Next, we define the time windows that .lost() counts are aggregated over. A simple strawman would be to read the counts at ingress+egress in absolute time increments, but due to queuing delay, the counts at ingress and egress at any instant will be displaced by the number of packets currently in the queue. Instead, we use the *arrival* time of packets as the epoch boundary. For example,

```
.lost([ipv4.srcIP_in], 20ms)
```

would report how many packets experience queuing loss that arrive in 20ms windows. Tuples returned by .lost() contain (epoch#, count, groupby_fields). Figure 2 shows an example of epoch timings. Note this means that the switch needs to store counts for the previous and next epoch at any time; we discuss this further in §4.3.

## 4 COMPILING PACKET LIFE CYCLE QUERIES

In this section, we explain the functions of the SwitchScope compiler, and how we overcome two challenges of compiling packet life cycle queries to a switch: (i) where to place state and computation and (ii) how to handle queuing loss.

### 4.1 The SwitchScope Compiler

SwitchScope uses a hybrid switch-controller design based on prior work [7, 12]. The SwitchScope compiler takes as input (i) a set of queries that the network operator writes and (ii) P4 code for the switch forwarding logic. The compiler divides each of the queries between a portion that can run in the data plane, and a portion that needs to be offloaded to a central query processor, due to switch constraints (discussed in §5). It further distributes the query operators to be executed at different locations in the pipeline, depending on (i) the tuple type used, (ii) the order of operators, and (iii) available switch resources. For example, a .filter(port_in == 2) operator could be applied at the start of ingress processing, even if applied to egress tuples.

The compiler then integrates the portion of the queries that can be executed at the switch with the forwarding logic to produce a single P4 program, which is loaded onto the switch. SwitchScope uses the switch’s MAUs to execute query operators [3], and aggregate operators like .reduce using the switch’s register memory as a multi-stage hash table. Hash collisions are dealt with by sending packets to the query processor, with an expiration policy based on epochs that we discuss in §4.3.

### 4.2 Tag Little, Compute Early

The first challenge in compiling SwitchScope queries is deciding where to place the query handling in the pipeline. For example, take the “Modification” query in Table 1, which operates on a stream of egress() tuples and filters for packets whose source IP was modified during switch processing. The query must be processed at the egress pipeline by definition, but it also needs access to the packet’s ipv4.srcIP_in when the packet arrived at the switch, before any modifications. To solve this, we tag the packet with PHV metadata containing its initial source IP when it arrives at the switch. When the packet reaches the end of egress processing, the switch
SwitchScope: A View from the Inside

compares its current source IP to the tagged metadata (both contained in the PHV) to complete the filter operation. However, excessive packet tagging uses up valuable state in the PHV that could be used by other queries, as well as existing switch processing. Thus, we want to minimize adding state to the PHV, and "free" that state when it is no longer needed. Since PHV state is pre-allocated by the P4 compiler, for our purposes, "freeing" state in the PHV means allowing future operators or switch processing to reuse the space allocated to prior operators/processing when that state is no longer needed. For example, consider the following query fragment:

```plaintext
egress()
.filter(ipv4.srcIP_in != ipv4.srcIP_mid)
```

The filter operation requires tagging the packet with its source IP when it arrives at the switch. But once the packet reaches the end of ingress processing, the filter can be applied, and the metadata is no longer needed. By executing the filter as early as possible, we can free any state that was used by the filter for future operators or other switch processing, reducing the overhead of queries on PHV state.

Together, these two compilation techniques form our "tag little, compute early" strategy for reducing the overhead of queries on switch state. The SwitchScope compiler tags packets with relevant query fields in their PHV when they become available, and executes operators as early as possible so their PHV space can be reused by future operators and switch processing.

### 4.3 Monitoring Queuing Loss in Epochs

Our second major challenge is monitoring queuing loss. As discussed in §3.4, the switch does not provide a programmable hook into packets that are dropped due to a full queue. This makes it difficult for SwitchScope to track individual packets which experience queuing loss, but it is feasible to track packet counts for queuing loss. Our solution is to store these counts in registers on both the ingress and egress pipelines, and later the query processor will retrieve these counts and compare them.

To compile lost queries, we take inspiration from Sonata’s ability to join the results of two queries together. The query on queuing loss in Table 1 can be expressed as:

```plaintext
ingress()
.map((ipv4.srcIP_in) => count=1)
.reduce(sum)
.join((egress())
 .map((ipv4.srcIP_in) => count=1)
 .reduce(sum)),
 func='diff',
 window='arrival', epoch_ms=5ms)
```

The queries track packet counts per source IP at the ingress and egress pipelines, respectively, and the .join computes their difference.

In order to handle epochs, SwitchScope tags each packet with an epoch number when it arrives at the switch, computed by \( \lfloor \text{time\_in\_epoch\_ms} \rfloor \), as this is easy to compute in the data plane using a bitshift. At each pipeline, the switch then stores \( (\text{epoch\#}, \text{groupby\_fields}) \rightarrow (\text{epoch\#}, \text{count in a d-stage hash table for each IP it observes, and updates the count if it has already been initialized. Our model assumes a FIFO processing ordering, so that when the first packet from epoch x is dequeued, no other packets from epoch x-1 will be seen by egress processing. Thus, when the egress pipeline first sees a packet from a new epoch, it can alert the central query processor to pull results from the switch. In case no packets arrive during an epoch, the query processor can pull results from the switch after no more packets from the previous could be waiting in the queue: the start time of the current epoch + the max ingress processing + queuing delay, which we can know in advance.

Finally, to expire old data, we use the fact that we store the epoch in the key of the hash table. If the insertion for a new packet collides with an entry with an epoch less than the current minus 1, we assume that the query processor had sufficient time to query the switch for counts from all previous epochs, and thus the data can be overridden.

### 5 DISCUSSION

**Integration of queries with user P4 program:** SwitchScope monitors a switch’s packet processing, so naturally it must integrate its own queries with the user’s existing P4 code. There are three key ways in which this occurs. First, the execution flow of a P4 program is defined in a control code block, and SwitchScope must augment this block to insert its query processing before/after existing processing. Second, the existing P4 code contains a custom parser that defines the packet’s headers; by extracting this parser, SwitchScope can allow queries to use any custom headers defined by the user. Finally, we have described the switch as “marking” a packet for drop, but in reality, when a packet matches an ACL rule,
a drop() action is called that may immediately terminate processing and drop the packet. To prevent this, SwitchScope modifies the user program to override the drop() action to set a “mark” bit in the PHV, which is then read implicitly when an query checks if port_intent/out == -1.

**Query compilation with switch resource constraints:**
PISA switches are limited in the number of stages per pipeline, the number of instructions that can be executed per stage, the size of the PHV, and the amount of register memory available in each stage [3, 7]. Each of these constraints reduces the number of query operators that can be executed on the switch, and fitting these constraints becomes increasingly difficult when running multiple queries simultaneously and integrating with existing switch processing. Sonata uses these constraints as input to an integer linear program (ILP) [7], and solves it to find an optimal partitioning of queries into the data plane of the switch that minimizes communication with the central query processor. SwitchScope can also use this ILP formulation, except the ILP now includes constraints on the ingress and egress pipelines separately, and we solve for the optimal partitioning of queries based on their respective pipelines, metadata tagging required, and the division of the existing switch processing among the ingress and egress pipelines.

**Queries with multiple input and output ports:**
Our current switch model assumes FIFO processing with a single ingress and egress pipeline shared among all ports. However, switches often contain multiple distinct pipelines that each process a fraction of ports on the switch. For example, consider a query fragment that generates a traffic matrix of in-out port pairs:

```hs
.egress()
.map((port_in, port_out) => 1)
.reduce(keys=(port_in, port_out), func=sum)
.filter(count > T)
```

In a single-pipeline switch, each in-out port pair has a single count at the egress pipeline, and it is easy to detect when this count exceeds a threshold. But with multiple pipelines, detecting when the count exceeds the threshold becomes a distributed heavy-hitter problem, in which multiple counts may individually fall below the threshold, but whose sum exceeds it. This problem has attracted research in a network-wide setting [9], and applying these techniques *within* a switch is an interesting future direction.

**Network-wide queries:** Much of SwitchScope’s query language can be extended to support network-wide queries that abstract the network as “one big switch.” In this abstraction, the in-out ports of the abstract switch are the border routers of the network, and the switch’s processing accounts for all switches within the network. “Tagging” a packet means adding headers to the packet as it traverses the network, rather than stripping the tags before the packet leaves the switch. This design could be combined with other network telemetry systems like Path Queries [13] to reason about the paths packets take through the network.

6 RELATED WORK

**Active measurement:** Some tools use active probing to monitor and detect issues in the network [6, 15]. However, active measurement systems only track artificial probes, whereas we want to monitor the modifications and drop/loss behavior that real network traffic experiences.

**Passive measurement:** Some passive systems work by forwarding copies of all packets to a central server (or set of servers), which is infeasible for large networks [8]. Everyflow [18] and dShark [16] require operators to filter a limited set of IPs to monitor, whereas SwitchScope can support all traffic at line rate. Other systems require control of end hosts [1, 10, 14], while SwitchScope does not.

**Dataflow telemetry:** Several recent systems use a dataflow language model for network telemetry. Sonata and Marple partition queries between the switch data plane and a central processor, but only observe packets when they *arrive* at the switch[7, 12], rather than monitoring the experience of packets within the switch. dShark is another dataflow telemetry system, but it analyzes packet traces rather than running in the data plane and requires switches to mirror traffic to collector boxes.

**Handling loss:** A couple of systems deal specifically with detecting lost packets. LossRadar [11] requires monitoring at multiple locations, while we monitor within a single switch. 007 requires end hosts to participate [1].

7 CONCLUSION

SwitchScope fills an important gap in network telemetry systems by peering inside the internals of modern programmable switches. SwitchScope offers rich telemetry insight into the life cycle of packets inside a switch where they could experience packet modifications, ACL drops, queuing delay and queuing loss. SwitchScope compiles and integrates dataflow telemetry queries with existing switch processing and employs a “tag little, compute early” compilation strategy to minimize query overhead. We also solve the problem of a missing programmable hook for packets lost to the queue, allowing a switch to track properties of its own packets lost to full queues by monitoring at both ingress and egress pipelines. In the future, any telemetry system for programmable switches should provide the same power and ease of monitoring the data plane as SwitchScope.
REFERENCES


In USENIX Networked Systems Design and Implementation.


