Compiling Path Queries in Software-Defined Networks

Srinivas Narayana, Jennifer Rexford and David Walker
Princeton University
{narayana, jrex, dpw}@cs.princeton.edu

ABSTRACT

Monitoring the flow of traffic along network paths is essential for SDN programming and troubleshooting. For example, traffic engineering requires measuring the ingress-egress traffic matrix; debugging a congested link requires determining the set of sources sending traffic through that link; and locating a faulty device might involve detecting how far along a path the traffic makes progress. Past path-based monitoring systems operate by diverting packets to collectors that perform "after-the-fact" analysis, at the expense of large data-collection overhead. In this paper, we show how to do more efficient "during-the-fact" analysis. We introduce a query language that allows each SDN application to specify queries independently of the forwarding state or the queries of other applications. The queries use a regular-expression-based path language that includes SQL-like "groupby" constructs for count aggregation. We track the packet trajectory *directly on the data plane* by converting the regular expressions into an automaton, and tagging the automaton state (*i.e.*, the path prefix) in each packet as it progresses through the network. The SDN policies that implement the path queries can be combined with arbitrary packet-forwarding policies supplied by other elements of the SDN platform. A preliminary evaluation of our prototype shows that our "during-the-fact" strategy reduces data-collection overhead over "after-the-fact" strategies.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations

Keywords

Network monitoring; software-defined network; network query

1. INTRODUCTION

Networks are notoriously difficult to measure well. Tools such as NetFlow, sFlow, and SNMP are effective at monitoring flows, packets, or aggregate statistics at individual links. However, operators often need to reason about the flow of traffic along *paths* through the network. Examples include measuring an ingress-egress traffic matrix [1], detecting the traffic sources and sinks affected by a congested link [2], catching access-control violations as they occur [3], or localizing faulty devices on the network using path-level information (§2).

Answers to such path-based queries can be achieved using one of two broad approaches. *Hoarders* record as much information as possible, so users can specify a wide range of more refined queries after the fact. *Neat Freaks* specify queries ahead of time, allowing them to collect and store much less data, but support a narrower range of pre-determined queries. Of course, almost any practical system contains elements of both the Hoarder and the Neat Freak. In the context of SDN, NetSight [3] is an example towards the Hoarder end of the spectrum: It creates a *postcard* for each packet at each hop in its journey. These postcards, which contain the packet header, the matching flow entry, and the switch and output port, are sent to servers that store them for later analyses, such as backtrace construction.

In contrast, in this paper, we illustrate how to develop a "tunable" Neat Freak for path queries in SDNs. Our query language allows users to ask questions about packets traversing paths specified using *regular expressions* of *boolean packet predicates*. The language also includes an SQL-like *groupby* construct for aggregating traffic statistics by different packet headers, such as source IP address, at the beginning, middle, or end of paths. Importantly, the specification is completely modular: It is possible to specify many different queries, each independent of one another, and independent of the underlying packet-forwarding policy. To illustrate the flexibility of our query language, we present a number of practical example queries (see Table 1).

Our run-time system implements these queries by generating OpenFlow rules that analyze packets *as they flow through the network's data plane*, to avoid directing every packet (or postcard) to collectors for analysis. To achieve this, we record packets' past trajectories onto bits on the packets themselves (*i.e.*, *tags*). The main insight is that the necessary information is just the packet's current state on a *Deterministic Finite Automaton (DFA)* that represents the path queries. Our run-time system first converts the path queries to this abstract DFA, and then generates *tag and capture* rules that (in effect) run the *transitioning* and *accepting* rules of the DFA—in a distributed fashion on the switches. Each packet traverses the DFA with its own in-

```
pred ::= true | false | match(header=value)
         | pred & pred | (pred | pred) | ~pred
         | ingress() | egress()

atom ::= in_atom(pred)
         | out_atom(pred)
         | in_group(pred, [header])
         | out_group(pred, [header])

path ::= atom | path ^ path | (path | path) | path*
```

Figure 1: Syntax of path queries.

dependent state (encoded as the tag), which is manipulated by OpenFlow rules to "move" the packet on its (own) DFA. Finally, the run-time system puts together these *tag and capture* rules with the forwarding policy specified by other components of the SDN platform.

Our run-time system is *proactive*, generating rules ahead of time to process traffic within the network, rather than at the controller. As such, we incur data-collection overheads only for packets that satisfy the queried paths. In addition, the system is lightweight, running on top of OpenFlow and leveraging counters on the data plane to collect statistics, and standard forwarding actions to redirect packets to a collector. Most importantly, the *user* gets to decide where to sit along the "Hoarders vs. Neat Freaks" spectrum, by specifying queries that collect the measurements that might be of interest—now and later. Preliminary evaluation of our prototype (built on top of Pyretic [4]) shows that we can deliver accurate results with monitoring and collection overheads proportional to the traffic volume matching a payload query. In summary, our contributions are:

- a query interface that allows compositional specification of path queries, through a regular expression-based language with grouping constructs (§2),
- a runtime system that directly observes packet paths on the data plane, and works with unmodified end hosts and OpenFlow 1.0 switches (§3), and
- preliminary results showing that our in-network "Neat Freak" strategy reduces data-collection overhead over typical "Hoarder" strategies (§4).

2. PATH-QUERY LANGUAGE

A path query is an expression that recognizes sets of packets that traverse particular paths through a network. Fig. 1 presents the basic syntax of path expressions, and Table 1 shows example applications that can be concisely expressed using the query language.

Basic Atoms. The simplest path query is one consisting of just one point in the network; we call such simple paths *atoms*. Moreover, every atom contains a *predicate*, which helps identify the specific network location of interest and the set of packets flowing through that location. For example, the atom

```
in_atom(match(srcip=H1) & match(switch=1))
```

identifies the set of packets with source IP `H1` that flow *in* to switch 1. In contrast,

```
out_atom(match(srcip=H1) & match(switch=1))
```

identifies the set of packets with source IP `H1` that flow *out* of switch 1. Those two sets of packets might be quite different, especially if switch 1 drops or rewrites some packets with source IP `H1`. Note that we usually abbreviate expressions such as `in_atom(match(f1=v1) & match(f2=v2))` with simply `in_atom(f1=v1, f2=v2)`. For example, we abbreviate the `in_atom` above with `in_atom(srcip=H1, switch=1)`. Two other useful atoms are `in_atom(ingress())`, which matches all packets entering the network, and `out_atom(egress())`, which matches all packets leaving the network.

Grouping Atoms. A basic atom identifies just *one* set of packets. A *grouping atom*, on the other hand, identifies *many* closely-related sets of packets. For example, the following atom matches all packets entering the network (via the `ingress` predicate) and then divides that set in to subsets based on the switch at which they arrive.

```
in_group(ingress(), ['switch'])
```

More generally, `in_group(pred, [h1, h2, ..., hn])` selects packets coming into a switch that satisfy the predicate `pred` and then divides that set into subsets where each packet in the subset shares common values for the headers `h1`, `h2`, ..., `hn`. Hence the grouping atom

```
in_group(ingress(), ['srcip','dstip'])
```

would divide the incoming packets into groups based on source IP-destination IP pairs. The `out_group` atom is similar to the `in_group` atom, except it identifies packets on their way out of a switch (like `out_atom`).

Regular Path Combinators. Given path queries p_1, p_2, we can build more complex paths using the usual concatenation (^), alternation/choice (|) and Kleene star/iteration ($*$). For instance, the path p_1 ^ p_2 requires that a packet first traverse the path p_1 and then traverse the path p_2. If p_1 and p_2 are simple atoms, this means that the packet must satisfy the predicate designated by p_1, incur exactly one hop "across the wire", and then satisfy the predicate designated by p_2. Table 1 presents several additional example queries.

Using Path Queries to Inspect or Count Packets. A programmer uses a path query by directing the packets matching the query to a *bucket*. Intuitively, a bucket is an abstract "location" in the network that collects packets. There are two kinds of buckets: *packet buckets* and *counting buckets*.

Packet buckets are currently implemented as queues on the controller that store a set of packets[1]. Programmers access the packets that arrive on these queues by associating a callback with the bucket. Packet buckets are quite useful for debugging. For instance, suppose an operator wants to localize a faulty switch that drops traffic somewhere along the path `S1, S2, ..., S10` in the network. She can designate a small subset of this traffic as "probe traffic", and specify a predicate `probe_pred` that matches against these probes. Now, she can easily set up a path query and an associated packet bucket that determines how far along this path the probe packets got, before they were dropped. She installs the query

```
p = (in_atom(switch=S1 & probe_pred) ^
        in_atom(probe_pred)*)
```

and a packet bucket to go with it:

[1]These packets can in principle be sent to designated "packet collector" nodes directly from the network switches.

Example	Query code	Description
A simple path	`in_atom(switch=S1) ^ in_atom(switch=S4)`	All packets going from switch `S1` to `S4` in the network.
Traffic matrix	`in_group(ingress(), ['switch']) ^ (true)* ^ out_group(egress(), ['switch'])`	All packets from any ingress point to any egress point, with counts grouped by (ingress, egress) switch pair.
Congested link diagnosis	`in_group(ingress(), ['switch']) ^ (true)* ^ in_atom(switch=sc,inport=pc) ^ (true)* ^ out_group(egress(), ['dstip'])`	Determine flows (switch sources → IP sinks) utilizing a congested link (on switch `sc` and port `pc`), to help reroute traffic around the congested link.
Faulty switch localization	`in_atom(switch=S1 & probe_pred) ^ in_atom(probe_pred)*`	Localize a faulty switch by observing how far designated probe packets move along a problematic path (see §2).
Middlebox traversal	`(in_atom(switch=TC) ^ in_atom(switch=IDS)) | (in_atom(switch=IDS) ^ in_atom(switch=TC))`	Count packets traversing adjacent middleboxes (*e.g.,* transcoder (`TC`), intrusion detection system (`IDS`)) in either order.
Waypoint invariant violation	`in_atom(ingress()) ^ in_atom(∼switch=FW)* ^ out_atom(egress())`	Detect all packets evading a firewall switch `FW` on the data plane (see §4).

Table 1: Example applications of the path-query language in Figure 1. We abbreviate `in_atom(true)` to `true`.

```
b = PacketBucket() // create bucket b
b.register_callback(last_hop) // record progress
p.set_bucket(b)    // associate b with path p
```

The callback `last_hop()` records the (topologically) last switch ID (say `k`) along the path that reported the probe packet. Armed with this information, the operator can investigate further—say, by running tests on switch `k+1` that did not report the probe packets.

Collecting aggregate traffic statistics requires a different sort of bucket—the counting bucket. Rather than directing individual packets to the controller, a counting bucket simply returns the total byte and packet counts across all packets matching the query. The specification of a counting bucket includes a measurement interval for returning the count values (e.g., every 30 seconds).

3. PATH QUERY COMPILATION

The main objective of the path query implementation is to find a way to use existing switch-level primitives (*e.g.,* specified by the OpenFlow API) to recognize packets *directly on the data plane* as they move through trajectories satisfying the path queries. At the same time, we must avoid corrupting the underlying packet-forwarding policy. If we can achieve these two goals, we can collect payloads or count the packets satisfying the queried paths *only* at the point where these 'interesting' packets complete their trajectory.

Overall, we achieve this goal by compiling the queries into OpenFlow rules that preserve the underlying forwarding policy, but add a few bits of state (*i.e.*, a *tag*) to packets as they traverse the network. These tags encode the *state* of the packets on a *Deterministic Finite Automaton* (DFA) that represents the set of input path queries (which are essentially regular expressions of predicates, hence this intermediate representation). All data plane packets move through this "abstract" DFA—which is actually distributed throughout the network—with *state transitioning* and *accepting* actions at every hop, that determine whether a packet is eventually counted against a query. We split the query compilation into the following key steps:

1. conversion of the set of path queries into a Deterministic Finite Automaton (DFA) (§3.1),
2. construction of packet-tagging and capturing/counting policies from the DFA (§3.2), and
3. merging tagging and counting policies with the packet forwarding policy to form a full policy for the network (§3.3).

We use `Pyretic` as an intermediate language to translate the DFA (from step (1)) into independent policies (in step (2)) that are finally compiled (in step (3)) into OpenFlow rules, so we briefly review the key primitives that we use from this language. Our compilation works in principle with any platform that exposes parallel and sequential composition primitives for network policies.

Background on Pyretic Primitives (`+` and `>>`). Table 2 summarizes some key elements of Pyretic's policy language. Each Pyretic policy should be thought of as a function from a packet to a set of packets. The network switches will implement these functions, taking one packet in a port, and producing 0 (dropped), 1 (unicast forwarding), or more (multicast forwarding) packets out other ports. For example, the policy `modify(port=2)` is a function that moves a packet from its current port to port 2. Predicates act as policies that filter packets: they drop any packet that does not match the predicate and leave any packet that does match it unchanged.

Programmers create larger policies by combining existing policies together using parallel (`+`) and sequential (`>>`) composition operators. The sequential composition `p >> q` executes the policy `p` on a packet and then `q` on the results of `p`. For instance,

```
match(srcip=H1,port=1) >> modify(port=2)
```

forwards any packet from `H1`, entering the switch at port 1, out port 2 and drops all other packets. The parallel composition `p+q` combines the outputs of `p` and `q`. For example,

```
match(srcip=H1) + match(srcip=H2)
```

selects packets from `H1` or `H2`'s source IP address, dropping all other packets.

Running example. For the remainder of this section, to illustrate our compilation algorithms, we show how to compile the following example into OpenFlow.

```
p1 = (in_atom(srcip=H1, switch=1) ^
      in_atom(dstip=H3, switch=3))
p2 = in_atom(switch=1) ^ in_atom(switch=3)
```

Query `p1` specifies packets that go from switch 1 to 3, with `H1`'s source IP address at switch 1 and `H3`'s destination IP at switch 3. The path `p2` specifies packets that go from switch 1 to 3.

Concept	Example	Description
Modification	`modify(port=2)`	Rewrites a packet field
Predicate	`match(switch=2)`	Filters packets
Parallel composition	`monitor + route`	The union of results from 2 policies.
Sequential composition	`balance >> route`	Pipe the output from the first in to the second

Table 2: Syntax of Key Pyretic Policies.

Predicate	String	Predicate	String
`switch=1 & srcip=H1`	`a`	`switch=1`	`a|e`
`switch=1 & ∼srcip=H1`	`e`	`switch=3`	`b|f`
`switch=3 & dstip=H3`	`b`	`p1`	`ab`
`switch=3 & ∼dstip=H3`	`f`	`p2`	`(a|e)(b|f)`

Figure 2: Strings emitted for the running example (§3.1). We avoid writing `match(...)` for brevity.

3.1 From Path Queries to DFA

We first convert the path queries into a Deterministic Finite Automaton (DFA). To construct the automaton, we first convert our path queries into standard regular expressions over strings. We use off-the-shelf libraries to convert regular expressions into DFAs.

Converting path queries to regular expressions. Path queries are regular expressions over packet predicates. To compile them, we convert them into regular expressions over strings of characters. The key step here is assigning specific characters to predicates. For example, consider the first query `p1` in our running example. We assign characters to predicates as follows.

```
match(srcip=H1, switch=1): 'a'
match(dstip=H3, switch=3): 'b'
```

This leads to the regular expression `ab` for path `p1`.

Next, we turn to converting the query `p2` in to a regular expression. Here, consider a packet with `srcip=H1` and `dst=10.0.0.3` entering switch 1. This packet satisfies two predicates: `match(switch=1, srcip='H1')` from `p1` and `match(switch=1)` from `p2`.

If we assign these two predicates different characters, our packet will wind up being in two different automaton states at the same time. This is undesirable, because we would rather only have *one DFA state per packet*, to make it feasible to encode and match using a limited number of scratch bits on the packet. Consequently, we partition the complex predicates we find in path queries into non-overlapping subparts (*i.e.*, an *orthogonal basis* for the complete set of predicates) and assign one character to each such non-overlapping subpart. For example, we construct the representations for the predicates and the paths `p1` and `p2` shown in Fig. 2. The DFA for `p1` and `p2` together is shown in Fig. 3(b).

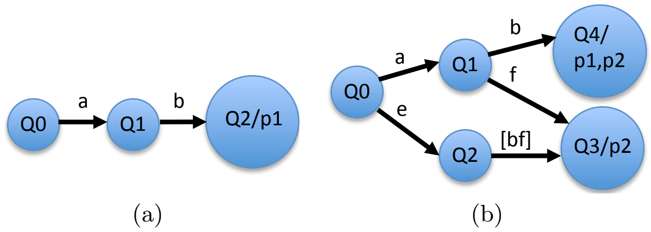

Figure 3: Deterministic finite automaton for the regular expressions from the running example (§3). (a) DFA only for path `p1`. (b) DFA for both `p1` and `p2`.

3.2 From a DFA to Tagging/Counting Policies

The next step is to translate the DFA into primitive *state transitioning* and *accepting* policies in `Pyretic`. These policies will read and write the relevant DFA state into each packet as it moves from switch to switch.

State Transition Policies. At any given time, each packet processed by the network is associated with some DFA state. We implement these states using Pyretic's virtual header field mechanism.[2] Packets just entering the network are stamped with the "starting" state of the DFA, and transitions thereafter are implemented on each switch as policies that check the current state, and write in a new state according to the transition rules. For example, consider a transition from DFA state `s` to DFA state `t` on seeing character `c`. Assume also that a function `pred_of` converts `c` to the associated predicate `p`. In such a case we generate the following clause.

```
match(state=s & pred_of(c)) >> modify(state=t)
```

The *tagging policy* for an automaton is the parallel composition of all such transition policies:

```
tagging = clause_1 + ... + clause_n
```

For example, the tagging policy for the DFA in Fig. 3(a) is implemented as follows:

```
(match(state=Q0) & match(srcip=H1,switch=1)) >>
    modify(state=Q1)) +
(match(state=Q1) & match(dstip=H3,switch=3)) >>
    modify(state=Q2))
```

Accepting policies. In addition to encoding the DFA transitions, we must encode when a packet is *accepted* by the automaton. The accepting actions are encoded as Pyretic policies that recognize the transition into an accepting DFA state, and direct those packets to the bucket associated with the accepting query. In general, when the automaton enters an accepting state `t` on reading character `c` from state `s`, the following *counting policy* captures the packets that satisfy the path query.

```
match(state=s) & pred_of(c) >> p.bucket()
```

In our running example (Fig. 3(a)), we generate the following counting policy for the accepting state.

```
match(state=Q1) & match(dstip=H3,switch=3) >>
    p1.bucket()
```

3.3 From Tagging/Counting to Pyretic Policy

The final step is merging the policies generated from the DFA with the global packet forwarding policy generated by other SDN applications. Our compiler assembles the following components: (1) the `tagging` policy, which implements DFA transitions on packets that undergo a state change, (2)

[2]Pyretic's virtual headers are currently encoded in the packet's VLAN header. Other mechanisms are possible.

Priority	Match	Action
3	`srcip=H1, dstip=H1, state=Q0`	→ `fwd(2), state=Q1`
3	`srcip=H1, dstip=H3, state=Q0`	→ `fwd(1), state=Q1`
2	`dstip=H1, state=Q0`	→ `fwd(2), state=Q2`
2	`dstip=H3, state=Q0`	→ `fwd(1), state=Q2`
1	`dstip=H1`	→ `fwd(2)`
1	`dstip=H3`	→ `fwd(1)`
0	`*`	→ `drop`

Table 3: Openflow rules on `S1` after installing the path queries in running example (§3). The original forwarding policy only has the last 3 rules shown.

an `unaffected` policy, which implements the identity function on those packets that do *not* undergo a state change, (3) The `fwding` policy, which implements the forwarding policies of other applications, and (4) the `counting` policy, which identifies the accepted packets. We put them together as follows:

```
((tagging + unaffected) >> fwding) + counting
```

Intuitively, this policy states that for each packet, we either tag (initiating a state change) or leave the packet unchanged, and then forward it. In parallel, we count the packet (or send it to the controller).

The `unaffected` policy captures the negation of all matches in the `tagging` policy. For our running example DFA in Fig. 3(a), `unaffected` is defined as follows.

```
~(match(state=Q0) & match(srcip=H1,switch=1)) &
~(match(state=Q1) & match(dstip=H3,switch=3))
```

We show a sample of the final OpenFlow 1.0 rules generated as a result of compiling the path queries for switch 1, in Table 3. The original forwarding policy only had the last 3 rules shown there, corresponding to destination-based forwarding for hosts `H1` and `H3`. However, the compilation has carefully teased apart overlapping predicates and actions among the queries and the forwarding rules, to generate data plane rules that work for both forwarding as well as queries.

Summary. We have shown how to take regular path queries, represent them as regular expressions, convert them to DFAs and then compile those DFAs to Pyretic policies. Notably, the translation of multiple simultaneous actions—tagging, capturing, and forwarding—into OpenFlow rules would have been rather difficult, if it were not for the ability to put together independently-constructed policies using parallel and sequential composition (which are supported by the Frenetic languages).

Implementation status. Our current prototype fully supports `in_atom` with all predicate combinations, and the basic path operators: concatenation (`^`), alternation (`|`), and repetition (`*`). We have also implemented a special case of the `out_atom` to support the network `egress()` predicate. The compilation algorithm above has been fully implemented, but it only supports queries with `in_atoms` (no `group` or `out` atoms). Currently, `Pyretic` already supports reactive compilation for groups for *single-point* queries; we are extending compilation support for the *path-level* group atoms. We are also working on fully supporting `out_atoms` and additional path combinators (negation (∼) and intersection (∧)).

4. EVALUATION

We show a preliminary demonstration of the reduction in data collection overheads while using a query-based system.

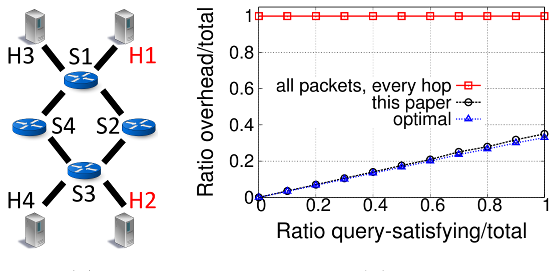

Figure 4: (a) Topology (b) Collection overhead (§4)

We run our experiments in `mininet` [5], a platform which uses linux container-based virtualization to emulate a bunch of network hosts and switches.

Reduction in payload collection overheads. We use the network topology in Fig. 4(a), where `S4` is a designated firewall switch. A network operator wants all traffic (in any direction) to pass through it. In our experiment, we setup `iperf` data transfers between hosts `H1` ↔ `H2`, and `H3` ↔ `H4`. We set up the routing so that `H3` ↔ `H4` traffic goes through the firewall `S4`, but `H1` ↔ `H2` traffic goes through `S2`—violating the operator's expectation.

Fortunately, the operator can preemptively install a query to collect the payloads of all packets whose trajectories imply just such a violation, by writing the path query that captures this condition:

```
in_atom(ingress()) ^ in_atom(~switch=S4)*
    ^ out_atom(egress())
```

This catches all packets entering the network (`ingress()` `in_atom`), following a trajectory that does not pass through the firewall (*i.e.*, every hop is such that `switch` ≠ `S4`), and exiting the network[3] (`egress() out_atom`).

We measure the collection overhead, which we define as the bytes received at the collector as a fraction of total data plane traffic, by collecting byte count statistics on the network using `tshark`. As an independent variable, we vary the volume of traffic that satisfies (or will eventually satisfy) the query, as a fraction of total data plane traffic. We achieve this by adjusting the ratio of the `iperf` transfer bandwidths we set for the `H1` ↔ `H2` flow (which satisfies the query) and `H3` ↔ `H4` flow (which does not satisfy the query).

Fig. 4(b) shows the resulting collection overhead measured against query-satisfying traffic. We compare three strategies to collect the same set of packets: (1) a naïve strategy that collects all packets at every hop, (2) our solution which uses compiled data plane rules to directly track data plane trajectories, and (3) the "optimal" single-point measurement which collects query-satisfying packets at network egress using `pcap` filters[4].

We make two observations. First, the overhead of our system grows linearly on the fraction of query-satisfying

[3]Note that in this topology, the firewall `S4` is never the first or last hop of a packet's intended ingress → egress trajectory.

[4]It is not, in general, possible to reduce a path measurement into one or more "single point" measurements without additional data plane state to record trajectory information. However, it is possible in this simple experiment.

traffic—the naïve strategy always collects all packets at every hop, and is always at fraction 1 in this curve. Further, our system's data collection overhead (byte volume) is very close to "optimal", *i.e.*, only $\approx 1/3$rd the volume of the query-satisfying traffic, as seen from the slope of the graph. This is because the compiled data plane rules can identify precisely those packets that *complete* the queried trajectories[5] using the packet state, without having to collect the packets at every hop. In particular, even if 100% of data plane packets (distinguished by location or payload) eventually satisfy the query, only $\approx 33\%$ of them will be sent to the collector, when they egress at `S3` after traversing 3 network hops.

For simplicity, each strategy in Fig. 4(b) sends the entire payload of the packets it collects from switches to the collector. Optimizing the number of bytes collected per packet, and building a distributed system to spray packets over multiple archiving collectors, are orthogonal to optimizing the *set of packets* collected in the first place. Our query system targets the latter; techniques to tackle the former problems [2,3] apply equally well to all strategies compared here.

Statistics collection overheads. We set up a network with five switches (each connected to one host) arranged in a cycle, and measure the switch ingress-to-egress traffic matrix. Since counting happens efficiently on the OpenFlow rule counters, applications can just poll switches for volume statistics instead of estimating them offline from samples. We vary the polling period from (every) 10s to 30s, and find that the collection overhead is in the range 5.4–7.2 KB/switch/polling period in all our runs. The exact collection overhead depends on the forwarding policy used, which determines the number of rules queried. The numbers we present are for a static destination-based forwarding policy.

Limitations. Query-based in-data-plane measurement helps operators "tune" the collection and processing overheads in the network according to their needs. However, as the queries increase in number and complexity, so does the corresponding DFA, whose installation in the network is increasingly constrained by data plane resources—such as switch rule space and bits to encode state on the packet (*e.g.,* VLAN header). For example, with our sample queries, we observed a 2–3x increase in the number of switch rules (installed on OpenFlow 1.0 switches which have a single rule table). In ongoing work, we are exploring techniques to evaluate and reduce these overheads.

5. RELATED WORK

Path-level measurements. Prior works such as trajectory sampling [2] and NetSight [3] directly observe packets from the data plane, diverting all or part of the packet to collectors for "after-the-fact" analysis. In contrast, we allow operators to *control* how much they want to store for later use, by only collecting traffic satisfying pre-specified queries, while flexibly supporting the full generality of queries available in earlier approaches.

Policy-based debugging. Header Space Analysis [6] and VeriFlow [7] detect when the current forwarding policy violates desirable trace properties (e.g., loop freedom). However, they do not offer a way to collect path-level traffic measurements in the data plane.

[5]Other semantics for data collection are possible.

Traffic query languages and systems. Gigascope [8] and Frenetic [9] support SQL-like queries on packet streams at a single point in the network. Even combining such measurements from multiple locations is not sufficient to infer packet trajectories, since packets (i) may enter the network at multiple locations and (ii) be dropped, modified, or rerouted in flight. In contrast, we show how to answer queries on *packet paths*—*i.e.*, packet observations spread across space and time.

Forwarding policies based on paths. Merlin [10], FatTire [11], and FlowTags [12] present contexts where implementing forwarding policies based on network paths is useful. However, these works do not address path-based *measurement,* which needs to preserve the behavior of an *existing* forwarding policy specified separately from the queries themselves.

6. ACKNOWLEDGMENTS

The authors thank Vimal Jeyakumar and Laurent Vanbever for helpful early discussions, Vimal Jeyakumar, Naga Praveen Katta, Joshua Reich, Kelvin Zou, Xin Jin, Mojgan Ghasemi, Divjyot Sethi, Bharath Balasubramanian and Jiasi Chen for helpful comments on earlier drafts of this paper, and Xin Jin for help with writing rule tables in latex. This work was supported by NSF grant TC-1111520 and DARPA grant MRC-007692-001.

7. REFERENCES

[1] Y. Zhang, M. Roughan, N. Duffield, and A. Greenberg, "Fast accurate computation of large-scale IP traffic matrices from link loads," in *Proc. ACM SIGMETRICS*, 2003.

[2] N. G. Duffield and M. Grossglauser, "Trajectory sampling for direct traffic observation," *IEEE/ACM Trans. Networking*, June 2001.

[3] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. USENIX NSDI*, 2014.

[4] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing Software-Defined Networks," in *Proc. USENIX NSDI*, 2013.

[5] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proc. ACM CoNEXT*, 2012.

[6] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. USENIX NSDI*, 2012.

[7] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *Proc. USENIX NSDI*, 2013.

[8] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: A stream database for network applications," in *Proc. ACM SIGMOD*, 2003.

[9] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *Proc. ACM International Conference on Functional Programming*, 2011.

[10] R. Soulé, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster, "Managing the network with Merlin," in *Proc. ACM Workshop on Hot Topics in Networking*, 2013.

[11] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "FatTire: Declarative Fault Tolerance for Software-defined Networks," in *Hot Topics in Software Defined Networks*, 2013.

[12] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags," in *Proc. USENIX NSDI*, 2014.