Compiling Path Queries

Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker
Princeton University
Technical Report

Abstract

Measuring the flow of traffic along network paths is crucial for many management tasks, including traffic engineering, diagnosing congestion, and mitigating DDoS attacks. We introduce a declarative query language for efficient path-based traffic monitoring. Path queries are specified as regular expressions over predicates on packet locations and header values, with SQL-like "groupby" constructs for aggregating results anywhere along a path. A run-time system compiles queries into a deterministic finite automaton. The automaton's transition function is then partitioned, compiled into match-action rules, and distributed over the switches. Switches stamp packets with automaton states to track the progress towards fulfilling a query. Only when packets satisfy a query are they packet counted, sampled, or sent to collectors for further analysis. By processing queries in the data plane, users "pay as they go", as data-collection overhead is limited to exactly those packets that satisfy the query. We implemented our system on top of the Pyretic SDN controller and evaluated its performance on a campus topology. Our experiments indicate that the system can enable "interactive debugging"—compiling multiple queries in a few seconds—while fitting rules comfortably in modern switch TCAMs and the automaton state into two bytes (e.g., a VLAN header).

1 Introduction

Effective traffic-monitoring tools are crucial for running large networks—to track a network's operational health, debug performance problems when they inevitably occur, account and plan for resource use, and ensure that the network is secure. Poor support for network monitoring and debugging can result in costly outages [6].

The network operator's staple measurement toolkit is well-suited to monitoring traffic at a single location (*e.g.,* SNMP/RMON and NetFlow/sFlow), or probing an end-to-end path at a given time (*e.g.,* ping and traceroute). However, operators often need to ask questions involving packets that traverse specific *paths, over time*: for example, to measure the traffic matrix [21], to resolve congestion or a DDoS attack by determining the ingress locations directing traffic over a specific link [20, 56], to localize a faulty device by tracking how far packets get before being dropped, and to take corrective action when packets evade a scrubbing device (even if transiently).

Answering such questions requires measurement tools that can analyze packets based both on their *location* and *headers*, attributes which may change as the packets flow through the network. The key measurement challenge is that, *in general*, it is hard to determine a packet's upstream or downstream path or headers. Current approaches either require inferring flow statistics by "joining" traffic data with snapshots of the forwarding policy, or answer only a small set of predetermined questions, or collect much more data than necessary (§2).

In contrast, when operators want to measure path-level flows in a network, they should be able to:

1. ask network-wide questions using concise, declarative queries,
2. program measurements without worrying about the network's forwarding policy (or its changes),
3. program measurements without worrying about other measurements of the system,
4. get direct and accurate results, without having to "infer" them by joining multiple datasets,
5. have direct control over measurement overhead, and
6. use standard match-action switch hardware [10,36].

A Path Query Language. We have developed a *query language* where users specify regular expressions over boolean conditions on packet location and header contents. To allow concise queries over disjoint subsets of packets, the language includes an SQL-like "groupby" construct that aggregates query results anywhere along a path. Different actions can be taken on a packet when it satisfies a query, such as incrementing counters, direct-

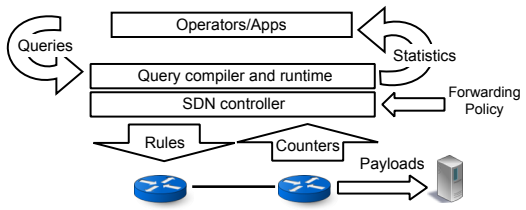

Figure 1: Path Query System.

ing traffic to a mirroring port or controller, or sampling at a given rate. These actions may be applied either before or after the packets traverse the matching trajectory.

The Run-time System. To implement a path query, the run-time system programs the switches to record path information in each packet as it flows through the data plane. While prior approaches have tracked packet paths this way [30, 49, 56], a naive encoding of every detail of the path—location and headers—would incur significant overheads. For example, encoding a packet's source and destination MAC addresses, and connection 5-tuple (24 bytes) at each hop incurs more than a 10% space overhead on a 1500-byte packet, if the packet takes six hops.

Instead, we customize packet path information to the input queries. More specifically, the run-time system compiles queries into a deterministic finite automaton (DFA), whose implementation is then distributed across the switches. The state of the DFA is stored in each packet as updated as it traverses the network. Upon receiving a packet, the switch reads the current DFA state, checks conditions implied by the query, writes a new DFA state on to the packet, executes actions associated with forwarding policy, and sends the packet on its way. Further, if a packet reaches an accepting state of the DFA, the actions associated with the accepting state are triggered. Hence, if the action associated with an accepting state is to send the packet to a collector, only packets actually matching a query are ever sent to a collector.

The mechanism we propose has an attractive "pay for what you query" cost model. Intuitively, our technique acts as an application-specific compression scheme for packet content and paths: rather than coding every detail of the packet trajectory, *only the information necessary to answer queries* is represented in the automaton state. When a packet hits an accepting state, all user-requested information about the packet path can be reconstructed.

Prototype Implementation and Evaluation. We have implemented a prototype of our query system on the Pyretic SDN controller [38] with the NetKat compiler [59]. Our compilation algorithms generate rules both for single and multi-stage match-action tables (*e.g.,* OpenFlow [36], [10]), and we implemented several compiler optimizations that reduce rule-space overhead and query compile time significantly. Our system design satisfies requirements (1)-(6) outlined earlier. On an emulated Stanford network topology, our prototype can compile several queries we tested (together) in under 10 seconds. We believe such compile times can enable "interactive" network debugging by human operators. The amount of packet state is less than two bytes, and fits in standard fields like VLAN or MPLS headers. Further, the emitted data plane rules—numbering a few hundreds—fit comfortably in the TCAM available on modern switches [10, 16, 26].

Contributions. In summary, this paper contributes:

1. the design of a query language that allows users to identify packets traversing a given set of paths (§3),
2. an evaluation of query expressiveness and the debugging model through examples (§4),
3. a run-time system that compiles queries to data-plane rules that emulate a distributed DFA (§6),
4. a set of optimizations that reduce query compile time by several orders of magnitude (§7), and
5. a prototype implementation and evaluation with the Pyretic SDN controller and Open vSwitch (§8).

We have open-sourced our prototype [66] and instructions to reproduce the results are available online [47].

Our preliminary workshop paper [40] on designing a path query system was only partly implemented, and the compilation strategy was prohibitively expensive for even moderately-sized networks. In this paper, we implement and evaluate a full system, and develop optimizations essential to make the system work in practice.

2 Design of Path Measurement

How to know which path a packet took through the network? How to collect or count all packets going through a specific path? These are the questions that a number of prior path measurement approaches [1, 18, 24, 49, 56, 60, 65, 75, 78] aim to answer. We discuss this design space in terms of two broad classes, *out-of-band* and *in-band* measurements. Policy-checking approaches, such as header space analysis [32] and VeriFlow [33] cannot answer these questions, as they only report packets that *could* satisfy certain conditions (*e.g.,* reachability), but do not capture the *actual* packets that do. These two sets of packets may be different because of congestion, faults, and misconfigurations in the data plane.

'Out-of-band' path measurement. These techniques collect observations of packets from network devices, and *infer* path properties of interest—for example, from independent packet samples [1, 53], trajectory labels [18], postcards [24], or matched and mirrored packets [71, 78]. Unfortunately, it is difficult to determine the full path of a single packet through observations spread out in space and time both *correctly* and *efficiently*:

(i) Dynamic forwarding policies: A simple way to get path measurements is to capture traffic entering a network (*e.g.,* packet samples [1]) and use the routing tables to estimate the paths the traffic would take. However, packet forwarding changes often due to topology changes, failover mechanisms (*e.g.,* MPLS fast reroute), and traffic engineering. Further, today's devices do not provide the timestamps at which the forwarding tables were updated, so it is difficult to reconcile packet-forwarding state with collected traffic data.

(ii) Packets dropped in flight: It is tricky to estimate actual packet trajectories even when packet forwarding is static. Packets may be dropped downstream from where they are observed, *e.g.,* due to congestion or faulty equipment, so it is difficult to know if a packet actually *completed* its inferred downstream trajectory.

(iii) Ambiguous upstream path: The alternative of observing traffic deeper in a network, on internal links of interest, cannot always tell where the traffic entered. For example, packets with identical header fields may arrive at multiple ingress points, *e.g.,* when packet headers are spoofed as in a DDoS attack, or when two ISPs peer at multiple points. Such packets would follow different paths eventually merging on the same downstream interface: disambiguating them at that point is impossible.

(iv) Packets modified in flight: Compounding the difficulty, switches may modify the header fields of packets in flight. "Inverting" packet modifications to compute the upstream trajectory is inherently ambiguous, as the upstream packet could have contained arbitrary values on the rewritten fields. Computing all possibilities is computationally difficult [77]. Further, packet modifications thwart schemes like trajectory sampling [18] that hash on header fields to sample a packet at each hop on its path.

(v) Opaque multi-path routing: Switch features like equal cost multi-path (ECMP) routing are currently implemented through hardware hash functions which are closed source and vendor-specific. This confounds techniques that attempt to infer downstream paths for packets. This is not a fundamental limitation (*e.g.,* some vendors may expose hash functions), but a pragmatic one.

(vi) High data collection overhead: Since both upstream and downstream trajectory inference is inaccurate, we are left with the option of collecting packets or digests at every hop [24, 60]. However, collecting all traffic is infeasible due to the bandwidth overheads. Even targeted match-and-mirror solutions [71, 78] cannot sustain the bandwidth overheads to collect all traffic affected by a problem. Sampling the packets at low rates [18] would make such overheads manageable, but at the expense of losing visibility into the (majority) unsampled traffic. This lack of visibility hurts badly when diagnosing problems for specific traffic (*e.g.,* a specific customer's TCP connections) that the sampling missed.

'In-band' path measurement: These approaches tag packets with metadata that *directly* identify paths of interest [30, 34, 40, 56, 65, 75] from the packet. However, current approaches have multiple drawbacks:

(vii) Limited expressiveness: Unfortunately, existing approaches only identify *complete, topological* paths, *i.e.*, the full sequence of switches and ports traversed by a packet [49, 56, 65, 75]. This is limiting for two reasons. First, operators also care about packet *headers*, including modifications to header fields in flight—*e.g.,* to localize the switch that violates a network slice isolation property [32]. Second, often an operator may only care about *waypoint* properties—for example, packets that skip a firewall—but current approaches will still incur the tag space overhead to record the *entire* path.

(viii) Strong assumptions: Current approaches require strong assumptions: *e.g.,* symmetric topology [65], no loops [65, 75], stable paths to a destination [56], or requiring that packets reach the end hosts [30, 34]. Unfortunately, an operator may be debugging the network exactly when such conditions *do not* hold.

Our approach. We design an accurate "in-band" path measurement system without the limitations of the prior solutions. A run-time system compiles *modular*, *declarative path queries* along with the network's forwarding policy (specified and changing independently), generating the switch-level rules that process *exactly* the packets matching the queries, in operator-specified ways—*e.g.,* counting, sampling, and mirroring. Hence, our system satisfies requirements (1)-(6) laid out in §1. Further, since the emitted data-plane rules process packets at every hop, our system overcomes problems (i), (ii), (iii), and (v) above. Identifying packet paths "in-band" with packet state untouched by regular forwarding actions removes ambiguities from packet modification (iv), and avoids unnecessary collection overheads (vi). Finally, our query language and implementation allow waypoint and header-based path specification (vii) and do not require strong operational assumptions to hold (viii).

However, we must overcome some challenges:

(i) Resource constraints: The space to carry packet trajectory metadata is limited, as packets must fit within the network's MTU. Further, switch rule-table space is limited [16], so the system should generate a compact set of packet-processing rules. Finally, to be usable for operator problem diagnosis, the system should compile queries in an acceptable amount of time.

(ii) Interactions between multiple measurement and forwarding rules: Switches must identify packets on all operator-specified paths—with some packets possibly on multiple queried paths simultaneously. The switch rules that match and modify packet trajectory metadata should not affect regular packet forwarding in the network, even

```
field ::= location | header
location ::= switch | inport | outport
header ::= srcmac | dstmac | srcip | dstip | ...

pred ::= true | false | field=value
         | pred & pred | (pred | pred) | ~pred
         | ingress() | egress()

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

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

Figure 2: Syntax of path queries.

when operators specify that packets matching the queries be handled differently than the regular traffic.

Practically, our query system is complementary to other measurement tools which are "always on" at low overheads [1, 53, 78]—as opposed to completely replacing those tools. Instead, our query system enables operators to focus their attention and the network's limited resources on clearly-articulated tasks during-the-fact.

3 Path Query Language

A path query identifies the set of packets with particular header values and that traverse particular locations. Such queries can identify packets with changing headers, as happens during network address translation, for instance. When the system recognizes that a packet has satisfied a query, any user-specified action may be applied to that packet. Fig. 2 shows the syntax of the language. In what follows, we explain the details via examples.

Packet Predicates and Simple Atoms. One of the basic building blocks in a path query is a *boolean predicate* (`pred`) that matches a packet at a single location. Predicates may match on standard header fields, such as:

```
srcip=10.0.0.1 & dstip=10.0.0.2
```

as well as the packet's location (a switch and interface). `True` and `false` are predicates which match all packets, and no packets, respectively. Conjunction (&), disjunction (|), and negation (~) are standard. The language also provides syntactic sugar for predicates that depend on topology, such as `ingress()`, which matches all packets that enter the network at some *ingress* interface, *i.e.*, an interface attached to a host or a device in another administrative domain. Similarly, `egress()` matches all packets that exit the network at some *egress* interface.

Atoms further refine the meaning of predicates, and form the "alphabet" for the language of path queries. The simplest kind of atom is an `in_atom` that tests a packet as it *enters* a switch (*i.e.*, before forwarding actions). Analogously, an `out_atom` tests a packet as it *leaves* the switch (*i.e.*, after forwarding actions). The set of packets matching a given predicate at switch entry and exit may be different from each other, since a switch may rewrite packet headers, multicast through several ports, or drop the packet entirely. For example, to capture all packets that enter a device `S1` with a destination IP address (say `192.168.1.10`), we write:

```
in_atom(switch=S1 & dstip=192.168.1.10)
```

To capture those packets that are sent *out* of `S1` with a destination IP address, say `10.0.1.10`, we write: `out_atom(switch=S1 & dstip=10.0.1.10)`

It also possible to combine those ideas, testing packet properties on both "sides" of a switch. More specifically, the `in_out_atom` tests one predicate as a packet enters a switch, and another as the packet exits it. For example, to capture all packets that enter a NAT switch with the virtual destination IP address `192.168.1.10` and exit with a private IP address `10.0.1.10`, we would write:

```
in_out_atom(switch=NAT & dstip=192.168.1.10,
            dstip=10.0.1.10)
```

Partitioning and Indexing Sets of Packets. It is often useful to specify groups of related packets concisely in one query. We introduce *group atoms*—akin to SQL `groupby` clauses—that aggregate results by packet location or header field. These group atoms provide a concise notation for partitioning a set of packets that match a predicate in to subsets based on the value of a particular packet attribute. More specifically, `in_group(pred, [h1,h2,...,hn])` collects packets that match the predicate `pred` at switch ingress, and then divides those packets into separate sets, one for each combination of the values of the headers `h1, h2, ..., hn`. The `out_group` atom is similar. For example,

```
in_group(switch=10, [inport])
```

captures all packets that enter switch 10, and organizes them into sets according to the value of the `inport` field. Such a groupby query is equivalent to writing a series of queries, one per `inport`. The path query system conveniently expands `groupbys` for the user and manages all the results, returning a table indexed by inport.

The `in_out_group` atom generalizes both the `in_group` and the `out_group`. For example,

```
in_out_group(switch=2, [inport], true, [outport])
```

captures all packets that enter `switch=2`, and exit it (*i.e.*, not dropped), and groups the results by the combination of input and output ports. This single query is shorthand for an `in_out_atom` for each pair of ports `i, j` on switch 2, *e.g.,* to compute a port-level traffic matrix.

Querying Paths. Full paths through a network may be described by combining atoms using the regular path combinators: concatenation (`^`), alternation (`|`), repetition (∗), intersection (&), and negation (∼). The most interesting combinator is concatenation: Given two path queries `p1` and `p2`, the query `p1 ^ p2` specifies a path that satisfies `p1`, takes a hop to the next switch, and then satisfies `p2` from that point on. The interpretation of the other operators is natural: `p1 | p2` specifies paths that satisfy *either* `p1` *or* `p2`; `p1*` specifies paths that are zero or more repetitions of paths satisfying `p1`; `p1 & p2` specifies paths that satisfy *both* `p1` and `p2`, and ∼`p1` specifies paths that do *not* satisfy `p1`. Table 1 presents several useful queries that illustrate the utility of our system.

Query Actions. An application can specify what to do with packets that match a query. For example, packets can be counted using hardware counters, be sent out a specific port (*e.g.,* towards a collector), sent to the SDN controller, or extracted through sampling mechanisms on switches. Below, we show `pyretic` sample code[1] for various use cases. Suppose that `p` is a path query defined according to the language (Fig. 2). Packets can be sent to abstract locations that "store" packets, called *buckets.* There are three types of buckets: *count buckets*, *packet buckets*, and *sampling buckets.* A count bucket is an abstraction that allows the application to count the packets going into it. Packets are not literally forwarded and held in controller data structures. In fact, the information content is stored in counters on switches. Below we illustrate the simplicity of the programming model.

```
cb = count_bucket() // create count bucket
cb.register(f)      // process counts by callback f
p.set_bucket(cb)    // direct packets matching p
                    // into bucket cb
...
cb.pull_stats()     // get counters from switches
```

Packets can be sent to the controller, using the packet buckets and an equally straightforward programming idiom. Similarly, packets can also be *sampled* using technologies like NetFlow [1] or sFlow [3] on switches.

In general, an application can ask packets matching path queries to be processed by an arbitrary *NetKAT* policy, *i.e.*, any forwarding policy that is a mathematical function from a packet to a set of packets [5,38]. The output packet set can be empty (*e.g.,* for dropped packets),

[1] We have taken a few liberties to make the code easier to format.

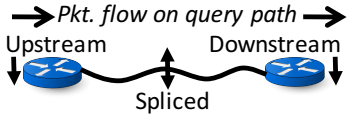

Figure 3: Query Capture Locations.

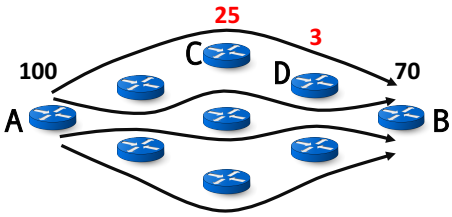

Figure 4: An example debugging scenario (§4).

or contain multiple packets (*e.g.,* for multicasted packets). For instance, packets matching a path query `p` can be forwarded out a specific mirroring port `mp`:

```
p.set_policy(fwd(mp)) // forward out mirror port
```

An arbitrarily complex Pyretic policy `pol` can be used instead of `fwd` above by writing `p.set_policy(pol)`.

Query Capture Locations. The operator can specify where along a path to capture a packet that satisfies a query: either *downstream*—once it has traversed a trajectory satisfying a query `p`, *upstream*—right as it enters the network, or *spliced*—somewhere in the middle. The difference between these three scenarios is illustrated in Fig. 3. The packets captured for the same query may differ at the three locations, because the network's forwarding policy may change as packets are in flight, or packets appearing upstream may be lost downstream due to congestion and faults. For query `p`, the operator writes `p.down()` to ask matching packets to be captured downstream, `p.up()` to be captured upstream, and `p.updown()` to be captured at both locations. He writes `splice(p1,p2)` to ask matching packets to be captured between two sub-paths of `p`, where `p = p1 ^ p2`.

The capabilities described above allow the implementation of a *network-wide packet capture tool*. Drawing on `wireshark` terminology [70], an operator is now able to write global, *path-based capture filters* to collect exactly and only the packets matching a query.

4 Interactive Debugging with Path Queries

Consider a scenario shown in Fig. 4 where an operator is tasked with diagnosing a tenant's performance problem in a large compute cluster, where the connections between two groups of tenant virtual machines A and B suffer from poor performance with low UDP throughput. The $A \rightarrow B$ traffic is routed along the four paths shown.

Debugging this performance problem is challenging. Virtualization obscures visibility into the end-host networking stacks, making network-stack-based analysis

Example	Query code	Description
A simple path	`in_atom(switch=S1) ^ in_atom(switch=S4)`	Packets going from switch `S1` to `S4` in the network.
Slice isolation	`true* ^ (in_out_atom(slice1, slice2) | in_out_atom(slice2, slice1))`	Packets going from network slice `slice 1` to `slice2`, or vice versa, when crossing a switch.
Firewall evasion	`in_atom(ingress()) ^ (in_atom(~switch=FW))* ^ out_atom(egress())`	Catch packets evading a firewall device `FW` when moving from any network ingress to egress interface.
DDoS sources	`in_group(ingress(), [switch]) ^ true* ^ out_atom(egress(), switch=vic)`	Determine traffic contribution by volume from all ingress switches reaching a DDoS victim switch `vic`.
Switch-level traffic matrix	`in_group(ingress(), [switch]) ^ true* ^ out_group(egress(), [switch])`	Count packets from any ingress to any egress switch, with results grouped by (ingress, egress) switch pair.
Congested link diagnosis	`in_group(ingress(), [switch]) ^ true* ^ out_atom(switch=sc) ^ in_atom(switch=dc) ^ true* ^ out_group(egress(), [switch])`	Determine flows (switch sources $\rightarrow$ sinks) utilizing a congested link (from switch `sc` to switch `dc`), to help reroute traffic around the congested link.
Port-to-port traffic matrix	`in_out_group(switch=s, true, [inport], [outport])`	Count traffic flowing between any two ports of switch `s`, grouping the results by the ingress and egress interface.
Packet loss localization	`in_atom(srcip=H1) ^ in_group(true, [switch]) ^ in_group(true, [switch]) ^ out_atom(dstip=H2)`	Localize packet loss by measuring per-path traffic flow along each 4-hop path between hosts `H1` and `H2`.
Loop detection	`port = in_group(true, [switch, inport]); port ^ true* ^ port`	Detect packets that visit any fixed switch and port twice in their trajectory.
Middlebox order	`(true* ^ in_atom(switch=FW) ^ true*) & (true* ^ in_atom(switch=P) ^ true*) & (true* ^ in_atom(switch=IDS) ^ true*) & ~(in_atom(ingress()) ** in_atom(switch=FW) ** in_atom(switch=P) ** in_atom(switch=IDS) ** out_atom(egress()))`	Packets that traverse a firewall `FW`, proxy `P` and intrusion detection device `IDS`, but do so in an undesirable order [51].
NAT debugging	`in_out_atom(switch=NAT & dstip=192.168.1.10, dstip=10.0.1.10)`	Catch packets entering a `NAT` with destination IP `192.168.1.10` and leaving with the (modified) destination IP `10.0.1.10`.
ECMP debugging	`in_out_group(switch=S1 & ecmp_pred, outport=1 | outport=2, [], [outport])`	Measure ECMP traffic splitting on switch `S1` for a small portion of traffic (predicate `ecmp_pred`), over ports 1 and 2.
Hop-by-hop debugging	`in_atom(switch=S1 & probe_pred) ^ true*`	Get notified at each hop as probe packets (satisfying `probe_pred`) starting at `S1` make progress through the network.

Table 1: Some example path query applications. The *middlebox order* query uses the abbreviation `**` to represent a concatenation with `true*`, *i.e.*, : `x ** y = x ^ true* ^ y`.

[63, 72] impossible. Coarse-grained packet sampling [1, 18] may still miss the specific traffic relevant to diagnosis, between VMs *A* and *B*. Even the more targeted "match and mirror" packet-collection techniques [78] would incur large overhead to collect all traffic from *A* to *B*. In general, it is complex for human operators to write packet-mirroring switch rules (*e.g.,* ACLs), since packets can be modified in flight, *e.g.,* to balance load [46, 68], and forwarded through several paths.

In contrast, we show the ease with which a declarative query language and run-time system allow an operator to efficiently pull "the needle from the haystack"—to determine the root cause of the performance problem.

As a first step, the operator determines whether the end host or the network is problematic, by issuing a query counting all traffic that enters the network from *A* destined to VM *B*. She writes the query `p1` below:

```
p1 = in_atom(srcip=vm_a, switch=s_a) ^ true*
     out_atom(dstip=vm_b, switch=s_b)
p1.updown()
```

The run-time then provides statistics for $A \rightarrow B$ traffic, measured at two points in the network: upstream (as traffic enters), and downstream (as traffic leaves). By comparing these two statistics, the operator can determine whether packets never left the host NIC, or were lost in the network. She does not have to correlate sampled packet records, or manually program switch ACLs to collect information from multiple devices.

Suppose the operator discovers a large loss rate in the network, as query `p1` returns values 100 and 70 as shown in Fig. 4. Her next step is to localize the interface where most drops happen, using a downstream query `p2`:

```
probe_pred = switch=s_a & srcip=vm_a & dstip=vm_b
p2 = stitch(in_atom(probe_pred),
            in_group(true, ['switch']),
            range(0,n))
```

The convenience function `stitch(...)` returns a set of queries by concatenating its first argument (an `in_atom`) with k copies of its second argument (an `in_group` atom), returning one query for each `k` in `0..n`. For example, `stitch(A, B, range(0,2)) = { A, A ^ B, A ^ B ^ B}` . This set of queries counts $A \rightarrow B$ traffic on each switch-level path and its prefix, from VMs A to B. The operator does not need to explicitly use the network's forwarding policy, or program specific devices along the paths, to get per-path statistics for this traffic.

Suppose the run-time returns, among statistics for other paths, the packet counts 25 and 3 shown in red in Fig. 4. The operator concludes that link $C \rightarrow D$ along the first path has a high $A \rightarrow B$ packet drop rate (22 out of 25 dropped). Such packet drops may be due to persistent congestion or a faulty interface, affecting all traffic on the interface, or bugs in software or hardware (*e.g.,* a "silent blackhole" [78]) which affect just $A \rightarrow B$ traffic. To tease out the possibilities, the operator writes two queries measured midstream (shown) and downstream (elided):

```
probe_pred = switch=s_a & srcip=vm_a & dstip=vm_b
p3 = splice((in_atom(probe_pred) ^ true*
             ^ in_atom(switch=s_c)),
            in_atom(switch=s_d))
p4 = splice(true* ^ in_atom(switch=s_c),
            in_atom(switch=s_d))
```

These queries determine the traffic loss rate on the $C \rightarrow D$ link, for all traffic traversing the link, as well as specifically the $A \rightarrow B$ traffic. By comparing these two loss rates, the operator can rule out certain root causes in favor of others. For example, if the loss rate for $A \rightarrow B$ traffic is particularly high relative to the overall loss rate, it means that that just the $A \rightarrow B$ traffic is silently dropped.

5 Example applications

We describe some example applications of the query language below.

Traffic matrix. A switch-level ingress-egress traffic matrix is a matrix of $N \times N$ numbers, where N is the number of switches in the network, and the $(i, j)^{\text{th}}$ entry is the volume of traffic entering the network at switch i and leaving the network at switch j. It is a useful input for management tasks like traffic engineering and anomaly detection. Some prior approaches to obtain the traffic matrix use coarse-grained data from SNMP logs or NetFlow, and algorithmically estimate (*e.g.,* tomography, gravity modeling, [76]) the numbers using the routing policy. Other sampling-based approaches (*e.g.,* [18]) send packet samples to collectors, and estimate traffic demand based on the collected samples. With the path query system, it is possible to get an exact answer to the question using data plane counters. An operator can write the query

```
in_group(ingress(), [switch]) ^ true*
  ^ out_group(egress(), [switch])
```

Here the predicate `ingress()` is syntactic sugar in our language for the set of ingress locations of the network, *i.e.*, switch interfaces which are connected to hosts or other administrative domains. The predicate `egress()` is similarly defined for egress locations.

It is also possible to get a *port-to-port* traffic matrix for a single switch (§3): `in_out_group(switch=s, true, [inport], [outport])`.

Counter-based packet loss localization. One way to debug poor performance due to high packet loss rate between two hosts (say `A, B`) is to figure out where packets are dropped in transit. It is possible to write a query that provides—for each switch along each path on which traffic between the hosts is forwarded—the packet counters of that specific traffic *separately for each path* that uses that switch. This is especially useful in highly-symmetric topologies like Fat Trees where there is extensive multipath routing for the traffic. Let's say we have a `k=4` fat tree, where every path has a length of 5 network hops. We initialize the query as follows:

```
def anyhop():
  return in_group(true, [switch])
def anynhops(n):
  q = anyhop()
  for m in range(1, n):
    q = q ^ anyhop()
  return q
pred = ingress() & srcip=ipA & dstip=ipB
ing = in_group(pred,['switch'])
p1 = ing ^ anynhops(1)
p2 = ing ^ anynhops(2)
p3 = ing ^ anynhops(3)
p4 = ing ^ anynhops(4)
```

Each call to the `anyhop()` function initializes a grouping atom for a new switch (which will be determined later on). We use this to generate a path with a flexible number of switches along the way, in the function `anynhops()`, for a given value of `n`. By running this value of `n` from 1 to 4 (which is the length of the tail of the path in a `k=4` fat tree), we can represent all paths and their prefixes between the source and the destination hosts. The result is a counts table with an entry for each path (and its prefix) taken by packets that enter the first edge switch with `srcip ipA` and `dstip ipB`.

Loop detection on data plane. Loop detection is a very useful primitive in networks. To write a query that detects packets returning to the same interface, we can write the query

```
some_port = in_group(true, [switch, inport])
```

```
p = some_port ^ true* ^ some_port
```

Here, `some_port` initializes a group atom that points to a single interface (*i.e.*, combination of switch and inport)—which is to be determined later. Then we can reuse this switch and port match (fixed to a set value) in a different point in the query to detect a packet that visits the *same* switch and inport twice in its trajectory.

Measuring multi-path load splits. Equal cost or weighted multi-path traffic splitting over a number of output ports is a common hardware mechanism for load balancing in several network contexts (*e.g.,* wide-area traffic engineering using OSPF equal-cost splitting or Google's B4 [28] network). Switches implement such functions at line rate through hardware hash functions which are often unspecified and hence vendor-specific (*e.g.,* [25]). However, as a network operator, it is frequently useful to measure how well load is balanced across the alternative outputs for a small subset of traffic (*e.g.,* are elephant flows originating from a given source on different outputs? [4]). However, exact load numbers for a small set of flows can be challenging to estimate either from sampled data (*e.g.,* NetFlow), and existing ACL counters may not have the desired granularity. Path queries enable an operator to measure the splits for very fine-grained traffic aggregates. Suppose the operator is interested in the traffic splits between two alternative ECMP choices on ports `1, 2` on switch `S1`, for a small set of flows (captured by a predicate `ecmp_pred`). She can write the query

```
in_out_group(switch=S1 & ecmp_pred,
             outport=1 | outport=2,
             [], [outport])
```

A similar query can be used to measure multicast load across different tree links at a given switch.

Congested link diagnosis. When a link in a network gets congested, it is useful for operators to know which sources are utilizing the congested link, and which downstream destinations are affected by it. A classic approach for this problem is to join the traffic matrix with the network's forwarding policy and derive those ingress-egress traffic demands that utilize the link [20]. Alternatives include using NetFlow or trajectory samples [18] to estimate the traffic contribution from each source destined to each downstream sink. An exact answer can be obtained from the data plane using path queries. Suppose the operator is interested in the link between switches `Si` and `Sj`. Then he writes:

```
in_group(ingress(), [switch]) ^ true*
  ^ out_atom(switch=Si) ^ in_atom(switch=Sj)
  ^ true* ^ out_group(egress(), [switch])
```

DDoS "sink tree". If a client is under a distributed denial-of-service (DDoS) attack, it is helpful to understand the traffic contribution by volume from all ingress (source) locations to the victim. A query that looks very similar to the one to inspect congested links (above) can be written to get just such a traffic break-down. Let the victim destination switch be `Svic`. Then the query

```
in_group(ingress(), [switch]) ^ true*
  ^ out_atom(egress() & switch=Svic)
```

provides the leaves of the "sink tree" to that destination with corresponding volumes. A similar query can be used to determine the spread of traffic originating from a single source of the network (*i.e.*, a "source tree").

Path waypoint violations. It is often useful to set up a live network monitor that checks violations of important waypoint policies in a network. Prior efforts approach this problem by verifying the data plane policy (*e.g.,* [32]), or collecting packet digests from all devices in the network [24]. However, transient conditions can cause invariant violations even if the initial and final policies satisfy the invariants. Further, a data plane query system allows collecting only those packets which violate the invariants on the data plane, saving bandwidth on the most common case where they are not. Consider some concrete examples:

Slice isolation. Suppose an operator has defined two slices of resources in the network (*i.e.*, virtual machines and corresponding host IP addresses), denoted by predicates `slice1` and `slice2`. To catch any packets that go from one slice to the other, the operator can write the query

```
p = ((in_atom(ingress() & slice1) ^ true*
        out_atom(egress() & slice2)) |
     (in_atom(ingress() & slice2) ^ true*
        out_atom(egress() & slice1)))
p.register_callback(alert_fun)
```

Here, `alert_fun` is a general purpose function written by the operator to, say trigger an alert.

Firewall evasion. An operator may choose to be alerted whenever packets leave a network without going through a firewall switch `FW`. To do so, he can write the query

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

The query as written assumes that the firewall isn't the ingress or egress hop for any packets in the network, but it is easy to generalize it.

Middlebox traversal order. An operator may find it useful to verify that all packets in the network traverse middleboxes in a specific order, for correctness or efficiency reasons [51]. For example, she may require that every packet go through a firewall `FW` and an intrusion detection system `IDS` in that specific order:

```
p = ∼(in_atom(ingress()) ^ true*
      ^ in_atom(switch=P) ^ in_atom(switch=IDS)
      ^ true* ^ out_atom(egress()))
```

6 Path Query Compilation

Query compilation translates a collection of independently specified queries, along with the forwarding policy, into data-plane rules that recognize all packets traversing a path satisfying a query. These rules can be installed either on switches with single-stage [36] or multi-stage [10] match-action tables. We describe downstream query compilation in §6.1-§6.3, and upstream compilation in §6.4. Downstream query compilation consists of three main stages:

1. We convert the regular expressions corresponding to the path queries into a DFA (§6.1).
2. Using the DFA as an intermediate representation, we generate state-transitioning (*i.e.*, *tagging*) and accepting (*i.e.*, *capture*) data-plane rules. These allow switches to match packets based on the state value, rewrite state, and capture packets which satisfy one or more queries (§6.2).
3. Finally, the run-time combines the query-related packet-processing actions with the regular forwarding actions specified by other controller applications. This is necessary because the state match and rewrite actions happen on the *same* packets that are forwarded by the switches (§6.3).

The run-time expands group atoms into the corresponding basic atoms by a pre-processing pass over the queries (we elide the details here). The resulting queries only contain `in`, `out` and `in_out` atoms. We describe query compilation through the following simple queries:

```
p1 = in_atom(srcip=H1 & switch=1) ^
       out_atom(switch=2 & dstip=H2)
p2 = in_atom(switch=1) ^
       in_out_atom(true, switch=2)
```

6.1 From Path Queries to DFAs

We first compile the regular path queries into an equivalent DFA,[2] in three steps as follows.

1. We transform every query into one that explicitly matches both at switch ingress and egress. This entails rewriting the `in_` and `out_` atoms in the query into `in_out_atoms`, forming a syntax tree where we can easily analyze overlapping predicates.
2. We convert the transformed queries (regular expressions over predicates) into regular expressions over characters. To generate the characters of the strings, we rewrite every predicate into a *disjunction of non-overlapping packet predicates* and assign a unique character to each element.
3. Finally, we use standard techniques to generate a DFA from the regular expression over characters.

The first step is quite straightforward. For instance, the path query `p1` is rewritten to the following:

```
in_out_atom(srcip=H1 & switch=1, true) ^
in_out_atom(true, switch=2 & dstip=H2)
```

In the second step, we convert the path queries into string regular expressions, by replacing each predicate by a character literal. However, this step is tricky: a key constraint is that different characters of the regular expressions cannot represent overlapping predicates (*i.e.*, predicates that can match the same packet). If they do, we may inadvertently generate an NFA (*i.e.*, a single packet might match two or more outgoing edges in the automaton). To ensure that characters represent non-overlapping predicates, we devise an algorithm that takes an input set of predicates P, and produces the smallest orthogonal set of predicates S that matches all packets matching P. The key intuition is as follows. For each new predicate `new_pred` in P, the algorithm iterates over the current predicates `pred` in S, teasing out new disjoint predicates and adding them to S:

```
int_pred = pred & new_pred
new_pred = new_pred & ∼int_pred
pred = pred & ∼int_pred
```

Finally, the predicates in S are each assigned a unique character. The full algorithm is described in Appendix B.

For the running example, Fig. 5 shows the emitted characters (for the partitioned predicates) and regular expressions (for input predicates not in the partitioned set). Notice in particular that the `true` predicate coming in to a switch is represented not as a single character but as an alternation of three characters. Likewise with `switch=1`, `switch=2`, and `true` (out). The final regular expressions for the queries `p1` and `p2` are:

```
p1: a^(c|e|g)^(a|d|f)^c
p2: (a|d)^(c|e|g)^(a|d|f)^(c|e)
```

Finally, we construct the DFA for `p1` and `p2` together using standard techniques. The DFA is shown in Fig. 6. For clarity, state transitions that reject packets from both queries are not shown.

6.2 From DFA to Tagging/Capture Rules

The next step is to emit policies that implement the DFA. Conceptually, we have two goals. First, for each packet,

[2] We could conceivably use an NFA instead of a DFA, to produce fewer states. However, using an NFA would require each packet to store all the possible states that it might inhabit at a given time, and require switches to have a rule for each subset of states—leading to a large number of rules. Hence, we compile our path queries to a DFA.

Predicate	Regex	Predicate	Regex
`switch=1 & srcip=H1`	`a`	`~switch=1`	`f`
`switch=1 & ~srcip=H1`	`d`	`~switch=2`	`g`
`switch=2 & dstip=H2`	`c`	`switch=1`	`a|d`
`switch=2 & ~dstip=H2`	`e`	`switch=2`	`c|e`
`true` (in)	`a|d|f`	`true` (out)	`c|e|g`

Figure 5: Strings emitted for the running example (§6.1).

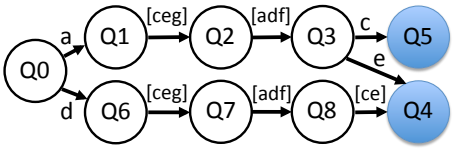

Figure 6: Automaton for `p1` and `p2` together. State `Q4` accepts `p2`, while `Q5` accepts both `p1` and `p2`.

a switch must read the DFA state, identify the appropriate transition, and rewrite the DFA state. This action must be done once at switch ingress and egress. Second, if the packet's new DFA state satisfies one or more queries, we must perform the corresponding query actions, *e.g.,* increment packet or byte counts.

State transitioning policies. The high-level idea here is to construct a "test" corresponding to each DFA transition, and rewrite the packet DFA state to the destination of the transition if the packet passes the test. This is akin to a string-matching automaton checking if an input symbol matches an outgoing edge from a given state. To make this concrete, we show the intermediate steps of constructing the transitioning policy in `pyretic` code.

We briefly introduce the notions of *parallel and sequential composition* of network policies, which we use to construct the transitioning policy. We treat each network policy as a mathematical function from a packet to a set of packets, similar to NetKAT and Pyretic [5, 38]. For example, a *match* `srcip=10.0.0.1` is a function that returns the singleton set of its input packet if the packet's source IP address is `10.0.0.1`, and an empty set otherwise. Similarly, a *modification* `port←2` is a function that changes the "port" field of its input packet to 2. Given two policies `f` and `g`—two functions on packets to sets of packets—the *parallel composition* of these two policies is defined as:

`(f + g)(pkt)` $=$ `f(pkt)` $\cup$ `g(pkt)`

The *sequential composition* of policies is defined as:

`(f >> g)(pkt)` $= \cup_{pkt' \in f(pkt)}$ `g(pkt')`

For example, the policy

```
(srcip=10.0.0.1 + dstip=10.0.0.2) >> (port←2)
```

selects packets with either `srcip 10.0.0.1` or `dstip 10.0.0.2` and forwards them out of port 2 of a switch.

Now we produce a policy fragment for each edge of the DFA. Suppose the helper function `pred_of` takes in a character input `c` and produces the corresponding predicate. For each edge from state `s` to state `t` that reads character `c`, we construct the fragment

```
state=s & pred_of(c) >> state←t
```

We combine these fragments through parallel composition, which joins the tests and actions of multiple edges:

```
tagging = frag_1 + frag_2 + ... + frag_n
```

We produce *two* state transitioning policies, one each for ingress and egress actions. Each edge fragment belongs to exactly one of the two policies, and it is possible to know which one since we generate disjoint characters for these two sets of predicates. For example, here is part of the ingress transitioning policy for the DFA in Fig. 6:

```
in_tagging =
  state=Q0 & switch=1 & srcip=H1 >> state←Q2 +
  state=Q0 & switch=1 & ~srcip=H1 >> state←Q6 +
   ... +
  state=Q7 & ~switch=1 >> state←Q8
```

Accepting policies. The accepting policy is akin to the *accepting* action of a DFA: a packet that "reaches" an accepting state has traversed a path that satisfies some query; hence the packet must be processed by the actions requested by applications. We construct the accepting policy by combining edge fragments which move packets to accepting states. We construct the fragment

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

for each DFA edge from state `s` to `t` through character `c`, where `t` is a state accepting query `p`. Here `p.policy()` produces the action that is applied to packets matching query `p`. Next we construct the *accepting* policy by a parallel composition of each such fragment:

```
capture = frag_1 + frag_2 + ... + frag_n
```

Similar to the transitioning policies, we construct two accepting policies corresponding to switch ingress and egress predicates. For example, for the DFA in Fig. 6, part of the accepting policy looks as follows:

Concept	Example	Description
Modification	`port←2`	Rewrites a packet field
Match	`switch=2`	Filters packets
Parallel composition	`monitor + route`	The union of results from two policies.
Sequential composition	`balance >> route`	Pipe the output from the first in to the second
Edge predicate	`pred_of(c)`	Get predicate of symbol
Path policy	`p.policy()`	Policy to process packets accepted by query `p`.

Figure 7: Syntactic Constructs in Query Compilation.

```
out_capture =
  state=Q3 & switch=2 & dstip=H2 >> p1.policy()
  + ... +
  state=Q8 & switch=2 & dstip=H2 >> p2.policy()
```

Ingress tagging and Egress un-tagging. The run-time ensures that packets entering a network are tagged with the initial DFA state `Q0`. Symmetrically, packets leaving the network are stripped of their tags. We use the VLAN header to tag packets, but other mechanisms are possible.

6.3 Composing Queries and Forwarding

The run-time system needs to combine the packet-processing actions from the transitioning and accepting policies with the forwarding policy. However, this requires some thought, as all of these actions affect the *same* packets. Concretely, we require that:

1. packets are forwarded through the network normally, independent of the existence of queries,
2. packet tags are manipulated according to the DFA,
3. packets matching path queries are processed correctly by the application-programmed actions, and
4. no unnecessary duplicate packets are generated.

To achieve these goals, the run-time system combines the constituent policies as follows:

```
(in_tagging >> forwarding >> out_tagging)
+ (in_capture)
+ (in_tagging >> forwarding >> out_capture)
```

The first sequential composition (involving the two `tagging` policies and the `forwarding`) ensures both that forwarding continues normally (goal 1) as well as that DFA actions are carried out (goal 2). This works because `tagging` policies do not drop packets, and the `forwarding` does not modify the DFA state.[3] The remaining two parts of the top-level parallel composition (involving the two `capture` policies) ensure that packets reaching accepting states are processed by the corresponding query actions (goal 3). Finally, since each parallelly-composed fragment either forwards packets normally or captures it for the accepted query, no unnecessary extra packets are produced (goal 4).

Translating to match-action rules in switches. The run-time system hands off the composed policy above to `Pyretic`, which compiles it down to a single match-action table [22, 38]. To leverage *multi-stage tables* on modern switches [10, 43], we rewrite the joint policy above as follows:

```
(in_tagging + in_capture)
 >> forwarding
 >> (out_tagging + out_capture)
```

[3]The run-time ensures this by constructing `tagging` policies with a virtual header field [38] that regular `forwarding` policies do not use.

This construction preserves the semantics of the original policy provided `in_capture` policies do not forward packets onward through the data plane. This new representation decomposes the complex compositional policy into a *sequential pipeline* of three smaller policies—which can be independently compiled and installed to separate stages of match-action tables.

6.4 Upstream Path Query Compilation

Upstream query compilation finds those packets at network ingress that *would* match a path, based on the current forwarding policy—assuming that packets are not dropped (due to congestion) or diverted (due to updates to the forwarding policy while the packets are in flight). We compile upstream queries in three steps, as follows.

Compiling using downstream algorithms. The first step is straightforward. We use algorithms described in sections §6.1-§6.3 to compile the set of upstream queries using *downstream* compilation. The output of this step is the *effective* forwarding policy of the network incorporating the behavior both of the forwarding policy as well as input queries. Note that we do *not* install the resulting rules on the switches.

Reachability testing for accepted packets. In the second step, we cast the upstream query compilation problem as a standard network *reachability* test [32, 33]. Specifically, we ask: when processed by the *effective* forwarding policy from the previous step, which packets—starting from the network ingress—eventually reach an accepting DFA state? We leverage *header space analysis* [32] to efficiently compute the full set of packet headers that match the query. We briefly review header space analysis and reachability testing below.

Header space analysis (HSA) models packets as objects existing in an L-dimensional geometric space: a packet is a flat vector of 0s and 1s of length L, which is the upper bound on the number of packet header bits. Sets of packet *headers*, h, are hence *regions* in the space of $\{0,1\}^L$. A switch T is modeled as a *transfer function* that maps headers arriving on a port, to set(s) of headers and ports: $T(h,p) : (h,p) \rightarrow \{(h_1,p_1),(h_2,p_2),\ldots\}$. The action of the network topology is also modeled similarly: if p_1 and p_2 are ports on the opposite ends of a link, then $T(h,p_1) = \{(h,p_2)\}$. The inverse of a switch or topology transfer function is well-defined [32].

Reachability testing asks which of all possible packet headers at a source can reach a destination header space and port. To compute this, HSA takes the set of all headers at a source port, and applies the network's transfer function iteratively until the header spaces reach the destination port. If the final header space at the destination port is empty, the source cannot send any packets that

reach the destination port. Otherwise, the *inverses* of the transfer functions are iteratively applied to the headers at the destination port to compute the *full set of packets* that the source can send to reach the destination.

Now we simply ask which packets at network ingress, when forwarded by the *effective* policy above, *reach* header spaces corresponding to accepting states for query `p`: let's call this packet match `upstream(p)`.

Capturing upstream. The final step is to process the resulting packet headers from reachability testing with application-specified actions for each query. We generate an *upstream capture* policy for queries `p1, ..., pn`:

```
(upstream(p1) >> p1.policy()) + ...
+ (upstream(pn) >> pn.policy())
```

We make a few remarks. First, without downstream query compilation, a straightforward application of HSA does *not* provide us the packets matching a regular expression query. Further, we can implement complex applications of HSA like loop detection and detecting leakage between slices [32, §5] *simply* by compiling the corresponding upstream path query (see queries in Table 1). Finally, we can compile spliced queries, *e.g.,* `splice(p1,p2)`, by asking for packets that *reach* `p2`-accepting states starting out as packets accepted by `p1` at any network port.

7 Optimizations

We implemented several key optimizations in our prototype to reduce query compile time and data-plane rule space. Later we show the quantitative impact of these optimizations (§8, Table 2).

Cross-product explosion. We first describe the "cross-product explosion" problem that results in large compilation times and rule-sets when compiling the policies resulting from algorithms in §6. The output of NetKAT policy compilation is simply a prioritized list of match-action rules, which we call a *classifier.* When two classifiers C_1 and C_2 are composed—using parallel or sequential composition (§6.2, Fig. 7)—the compiler must consider the effect of every rule in C_1 on every rule in C_2. If the classifiers have N_1 and N_2 rules (resp.), this results in a $\Theta(N_1 \times N_2)$ operations. A similar problem arises when predicates are partitioned during DFA generation (§6.1). The number of orthogonal predicates may grow exponentially ($O(2^N)$) on the input predicate set (of size N), since in the worst case every input predicate may overlap partially with every other one.

Prior works have observed similar problems [17, 23, 59, 74]. Our optimizations curtail such explosive running times through domain-specific techniques, as follows.

(A) Optimizing Conditional Policies. The policy generated from the state machine (§6.2) has a very special structure, namely one that looks like a conditional statement: *if state=s1 then ... else if state=s2 then ... else if ....* A natural way to compile this down is through the parallel composition of policies that look like `state=s_i >> state_policy_i`. This composition is expensive, because the classifiers of `state_policy_i` for all `i`, $\{C_i\}_i$, must be composed parallelly. We avoid computing these cross-product rule compositions as follows: If we ensure that each rule of C_i is specialized to match on packets disjoint from those of C_j—by matching on state `s_i`—then it is enough to simply *append* the classifiers C_i and C_j. This brings down the running time from $\Theta(N_i \times N_j)$ to $\Theta(N_i + N_j)$. We further compact each classifier C_i: we only add transitions to non-dead DFA states into `state_policy_i`, and instead add a default dead-state transition wherever a C_i rule drops the packets.

Now we describe the details of this optimization. First, we introduce a *switch case* syntactic structure in the `pyretic` intermediate language:

```
switch (field):
  case v0 -> p0
  ...
  case vM -> pM
  default -> [actions]
```

This means that the policy `p_i` is applied when the value of the field `field` is `v_i`, and if no values match, the default actions are applied. Since no two policies `p_i, p_j` act on the same packet (*i.e.*, the packet must differ in the value of `field`), this policy structure can be compiled as follows. Each policy `p_i` is independently compiled, and each rule of C_i is specialized by a match on the `field` value. The classifiers are concatenated with each other, and finally with a single rule that performs the default actions, to produce the final classifier. By getting rid of the cross-product of classifiers $C_0, C_1, ...C_M$, policy compilation now takes $N_0 + ... + N_M$ time instead of $N_0 \times ... \times N_M$.

Second, we introduce the policy structure `p else a`, where `p` is any policy and `a` is an unconditional list of primitive actions (*e.g.,* field modifications). The policy `p else a` takes an input packet and evaluates it by policy `p` first, and if `p` drops the packet, then actions `a` are applied, otherwise it simply acts as `p`. This structure is compiled by first compiling `p`, and checking each rule in the classifier to determine if it drops the packet. If so, the classifier is modified so the actions `a` are executed instead. This produces a classifier in just $N_p + 1$ time, processing any packets that `p` drops.

Now we compile the policy

```
switch (state):
  case s0 -> state-transitioning from s0
```

```
            (to live states)
  ...
  case sM -> state-transitioning from s0
            (to live states)
  default -> [transition to dead state]
else [transition to dead state]
```

which ensures that each state classifier is independently constructed (no "crossing" across states), and that dead state transitions are automatically included on any packets not processed by the switch case policy.

(B) Integrating tagging and capture policies. `Tagging` and `capture` policies have similar conditional structure:

```
tagging =                  capture =
  (cond1 >>  a1) +           (cond1 >>  b1) +
  (cond2 >>  a2) +           (cond2 >>  b2) +
  ...                         ...
```

Rather than supplying Pyretic with the policy `tagging + capture`, which will generate a significant cross-product, we construct a simpler equivalent policy:

```
combined =
  (cond1 >>  (a1 + b1)) +
  (cond2 >>  (a2 + b2)) +
  ...
```

(C) Flow-space based pre-partitioning of predicates. In many queries, we observe that most input predicates are disjoint with each other, but predicate partitioning (§6.1) checks overlaps between them anyway. We avoid these checks by pre-partitioning the input predicates into disjoint flow spaces, and only running the partitioning (Alg. 1) within each flow space. For example, suppose in a network with `n` switches, we define `n` disjoint flow spaces `switch=1, ..., switch=n`. When a new predicate `pred` is added, we check if `pred & switch=i` is nonempty, and then *only* check overlaps with predicates intersecting the `switch=i` flow space.

(D) Caching predicate overlap decisions. Often we find an input predicate (*e.g.,* `true`) checked for overlaps redundantly after a previous occurrence of the same predicate was partitioned. We avoid such checks by caching the latest overlap results for all input predicates[4], and executing the remainder of the partitioning algorithm (Alg. 1) only when the cache is missed. Caching also enables introducing new queries incrementally into the network without recomputing all previous predicate overlaps.

(E) Decomposing query-matching into multiple stages. Often the input query predicates may have significant overlaps: for instance, one query may count on M source IP addresses, while another counts packets on N destination IP addresses. By installing these predicates on a single table stage, it is impossible to avoid using up $M \times N$ rules. However, modern switches [10,44] support several match-action stages, which can be used to reduce rule space overheads. In our example, by installing the M source IP matches in one table and N destination matches in another, we can reduce the rule count to $M+N$. These smaller *logical* table stages may then be mapped to *physical* stages on a multi-stage hardware table [31,57].

We devise an optimization problem to divide queries into groups that will be installed on different table stages. The key intuition is that if we avoid grouping queries that match on dissimilar header fields in one table stage, we can significantly reduce rule count. To formalize this, we specify a cost function that estimates the worst-case rule space when combining predicates (Appendix A). The resulting optimization problem is NP-hard; however, we design a first-fit heuristic to group queries into multiple table stages, given a limit on the number of table stages and rule space per stage. The compilations of different table stages are parallelizable.

(F) Detecting overlaps using Forwarding Decision Diagrams (FDDs). To make intersection between predicates efficient, we implement a recently introduced data structure called *Forwarding Decision Diagram* (FDD) [59]. An FDD is a binary decision tree in which each non-leaf node is a test on a packet header field, with two outgoing edges corresponding to true and false. Each path from the root to a leaf corresponds to a unique predicate which is the intersection of all tests along the path. Inserting a new predicate into the FDD only requires checking overlaps along the FDD paths *which the new predicate intersects,* hence the FDD can efficiently maintain the set of partitioned predicates in its leaves.

8 Performance Evaluation

We evaluated the expressiveness of the query language and the debugging model in Table 1 and §4. Now, we evaluate the prototype performance quantitatively.

Implementation. We implemented the query language and compilation algorithms (§3 and §6) on top of the Pyretic SDN controller [38] and NetKat compiler [59]. We extended the Hassel-C [48] implementation of header space analysis with inverse transfer function application for upstream compilation. NetFlow samples are processed with `nfdump` [42]. The query language is embedded in Python, and the run-time system is a library on top of Pyretic. As the target of compilation, our prototype supports single-stage (OpenFlow 1.0) tables, as well as multi-stage tables (with Open vSwitch Nicira extensions [45]). We use `Ragel` [14] to compile string regular expressions. Our prototype is open-source [66]. We eval-

[4]We index this cache by a hash on the string representation of the predicate's abstract syntax tree.

uate our system using the PyPy compiler [50], and instructions to reproduce results are available online [47].

Metrics. A path-query system should be efficient along the following dimensions:

1. Query compile time: Can a new query be processed at a "human debugging" time scale?
2. Rule set size: Can the emitted match-action rules fit into modern switches?
3. Tag set size: Can the number of distinct DFA states be encoded into existing tag fields?

We do not report on query install time since this is highly switch-dependent (10-300 flow setups/sec have been reported [16, 26]). Further, packet-processing latency and throughput are unaffected as switches are already designed to do the necessary data plane actions at line rate[5].

Experiment Setup. We pick a combination of queries from Table 1, including switch-to-switch traffic matrix, congested link diagnosis, DDoS source detection, counting packet loss per-hop per-path[6], slice isolation between two IP prefix slices, and firewall evasion. These queries involve broad scenarios such as resource accounting, network debugging, and enforcing security policy. We run our single-threaded prototype on an Intel Xeon E3 server with 3.70 GHz CPU (8 cores) and 32GB memory.

Compiling to a multi-stage table is much more efficient than single-stage table, since the former is not susceptible to cross-product explosion (§7). For example, the traffic matrix query incurs three orders of magnitude smaller rule space with the basic multi-stage setup (§6.3), relative to single-stage. Hence, we report multi-stage statistics throughout. Further, since optimization (E) decomposes queries into multiple stages (§7), and the stage compilations are parallelizable, we report the *maximum* compile time across stages whenever (E) is enabled.

(I) Benefit of Optimizations. We evaluate our system on an emulated Stanford campus topology [2], which contains 16 backbone routers, and over 100 network ingress interfaces. We measure the benefit of the optimizations when compiling *all of the queries listed above* together—collecting over 550 statistics from the network.

The results are summarized in Table 2. Some trials did not finish[7], labeled "DNF." Each finished trial shown is an average of five runs. The rows are keyed by optimizations—whose letter labels (A)-(F) are listed in paragraph headings in §7. We enable the optimizations one by one, and show the *cumulative* impact of all enabled optimizations in each row. The columns show statistics of interest—compile time (absolute value and factor reduction from the unoptimized case), *maximum* number of table rules (ingress and egress separately) on any network switch, and required packet DFA bits.

Enabled Opts.	Compile Time		Max # Rules		# State Bits
	Abs. (s)	Rel. (X)	In	Out	
None	> 4920	*baseline*	DNF	DNF	DNF
(A) only	> 4080	1.206	DNF	DNF	DNF
(A)-(B)	2991	1.646	2596	1722	10
(A)-(C)	56.19	87.48	1846	1711	10
(A)-(D)	35.13	139.5	1846	1711	10
(A)-(E)	5.467	894.7	260	389	16

Table 2: Benefit of optimizations on queries running on the Stanford campus topology. "DNF" means "Did Not Finish."

Network	# Nodes	Compile Time (s)	Max # Rules		# State Bits
			In	Out	
Berkeley	25	10.7	58	160	6
Purdue	98	14.9	148	236	22.5
RF1755	87	6.65	150	194	16.8
RF3257	161	44.1	590	675	32.3
RF6461	138	21.4	343	419	29.2

Table 3: Performance on enterprise and ISP (L3) network topologies when all optimizations are enabled.

The cumulative compile-time reduction with all optimizations (last row) constitutes three orders of magnitude—reducing the compile time to about 5 seconds, suitable for interactive debugging by a human operator.[8][9] Further, the maximum number of rules required on any one switch fits comfortably in modern switch memory capacities [10, 16, 26]; and the DFA state bits (2 bytes at most) fit within tag fields like VLANs. Finally, multi-stage query decomposition (E) reduces rule space usage significantly with more state bits.

(II) Performance on Enterprise and ISP networks. We evaluate our prototype on real enterprise and inferred ISP networks, namely: UC Berkeley [8], Purdue University [64], and Rocketfuel (RF) topologies for ASes 1755, 3257 and 6461 [55, 62]. All optimizations are enabled. For each network, we report averages from 30 runs (five runs each of six queries). The results are summarized in Table 3. The average compile time is under 20 seconds in all cases but one; rule counts are within modern switch TCAM capacities; and DFA bits fit in an MPLS header.

(III) Scalability trends. We evaluate how performance scales with network size, on a mix of five synthetic ISP topologies generated from Waxman graphs [69] and IGen, a heuristic ISP topology generator [52]. The edge probability between vertices u, v in a Waxman graph is given by the relationship $P(u,v) = \alpha exp(-d/L\beta)$, where d is the distance between u and v, and L is

[5] except sending packets to the controller.

[6] We use the version of this query from §4, see p2 there.

[7] The reason is that they run out of memory.

[8] Interactive response times within about 15 seconds retain a human in the "problem-solving frame of mind" [37, topic 11].

[9] We disabled FDDs (F) for this experiment as it increases compile time to 98s; empirically we find (F) beneficial for larger networks.

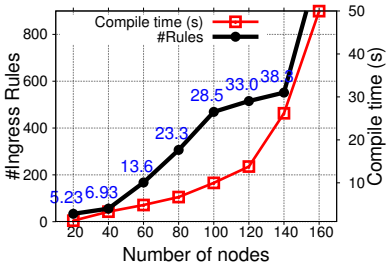

Figure 8: Scalability trends on synthetic ISP topologies. Numeric labels on the curve correspond to # DFA packet bits.

a scale parameter. We generated connected Waxman graphs with four sets of parameters: $(\alpha,\beta) = (0.2,0.4),(0.3,0.3),(0.4,0.2),(0.5,0.15)$, at each network size. On the IGen topology, we generated a topology with Delauney Triangulation enabled, for different network sizes. We report average metrics from 30 runs, *i.e.*, six queries compiled to five networks of each size. The trends are shown in Fig. 8. The average compile time (see red curve) is under $\approx$ 25 seconds until a network size of 140 nodes. In the same size range, the ingress table rule counts (see black curve) as well as the egress (not shown) are each under 700—which together can fit in modern switch TCAM memories. DFA packet bits (see numeric labels on black curve) fit in an MPLS header until 120 nodes.

For networks of about 140 nodes or smaller, our query system supports interactive debugging—continuing to provide useful results beyond for non-interactive tasks. Among large ISP topologies mapped out in literature [62], each ISP can be supported in the "interactive" regime for PoP-level queries. We leave further scaling efforts, *e.g.,* to data centers, to future work.

9 Related Work

We already discussed the most relevant prior works in §2; this section lays out other related work.

Data-plane query systems. Several query languages have been proposed for performing analytics over streams of packet data [9,15,22,67]. Unlike these works, we address the collection of *path-level* traffic flows, *i.e.*, observations of the same packets *across* space and time, which cannot be expressed concisely or achieved by (merely) asking for single-point packet observations.

Control-plane query systems. NetQuery [58], PIER [27], DECOR [11] and Akamai 'Query' [13] allow operators to query configuration or other operational information (*e.g.,* next hop in the forwarding table, attack fingerprints, *etc.*) from tuples stored on network nodes. As such, these works do not query the data plane. SIMON [41] and ndb [35] share our vision of interactive debugging, but focus on isolating control plane bugs.

Program path profiling. In a different context, Ball and Larus [7] determine program paths by instrumenting register arithmetic operations at program branching points. However, a direct application of this technique to record packet paths (including headers) would incur too much packet space (*e.g.,* 10% of an MTU packet, §1).

Summary statistics monitoring systems. DREAM [39], ProgME [74] and OpenSketch [73] answer a different set of monitoring questions than our work, *e.g.,* detecting heavy hitters and changes in traffic patterns.

Programming traffic flow along paths. Several prior works [19, 29, 54, 61] aim to forward packets along policy-compliant paths. However, our work *measures* traffic along operator-specified paths, while the usual forwarding policy continues to handle traffic.

Symbolic DFAs. Our usage of DFAs with packet predicates, and partitioning to form an orthogonal basis, are closely related to symbolic automata and minterms [17, 74]. We reduce DFA construction over predicates to the standard version for character literals.

10 Conclusion

We have shown how to take a declarative specification for path-level measurements, and implement it in the data plane with accurate results at low overhead. We believe that this capability will be useful for network operators for better real-time problem diagnosis, security policy enforcement, and capacity planning.

An in-data-plane implementation of a general "path predicate" enables further possibilities. Packets can be forwarded purely based on the prior path they have taken through a network, to implement path-based forwarding [19, 29, 54, 61]. Limited querying functionality can be deployed using SDN only at the "edge" of a network (with a legacy "core"), by tagging matching packets at network ingress using upstream compilation. Legacy monitoring tools like Flexible NetFlow [12] and ACL-based mirroring in the network's core could leverage the packet tags to capture only the interesting traffic.

References

[1] Sampled Netflow, 2003. http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/12s_sanf.html.

[2] Mini-Stanford backbone topology, 2014. [Online, Retrieved September 10, 2015] https://bitbucket.org/peymank/hassel-public/wiki/Mini-Stanford.

[3] sFlow, 2015. sflow.org.

[4] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. H edera: Dynamic flow scheduling for data center networks. In *Proc. USENIX NSDI* (2010).

[5] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic Foundations for Networks.

[6] ANDREW LERNER. The cost of downtime, 2014. [Online, Retrieved September 10, 2015] http://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/.

[7] BALL, T., AND LARUS, J. R. Efficient path profiling. In *Proc. ACM/IEEE International Symposium on Microarchitecture* (1996).

[8] BERKELEY INFORMATION SERVICES AND TECHNOLOGY. UCB network topology. [Online, Retrieved September 22, 2015] http://www.net.berkeley.edu/netinfo/newmaps/campus-topology.pdf.

[9] BORDERS, K., SPRINGER, J., AND BURNSIDE, M. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security* (2012).

[10] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proc. ACM SIGCOMM* (2013).

[11] CHEN, X., MAO, Y., MAO, Z. M., AND VAN DER MERWE, J. Decor: Declarative network management and operation. *ACM SIGCOMM Computer Communication Review* (2010).

[12] CISCO. Flexible NetFlow. [Online, Retrieved September 23, 2015] http://www.cisco.com/c/en/us/products/ios-nx-os-software/flexible-netflow/index.html.

[13] COHEN, J., REPANTIS, T., MCDERMOTT, S., SMITH, S., AND WEIN, J. Keeping track of 70,000+ servers: The Akamai query system. In *Proc. Large Installation System Administration Conference, LISA* (2010).

[14] COLM NETWORKS. Ragel state machine compiler. [Online, Retrieved September 10, 2015] http://www.colm.net/open-source/ragel/.

[15] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: A stream database for network applications. In *Proc. ACM SIGMOD* (2003).

[16] CURTIS, A. R., MOGUL, J. C., TOURRILHES, J., YALAGANDULA, P., SHARMA, P., AND BANERJEE, S. DevoFlow: Scaling Flow Management for High-performance Networks. In *Proc. ACM SIGCOMM* (2011).

[17] D'ANTONI, L., AND VEANES, M. Minimization of symbolic automata. In *Proc. ACM Symposium on Principles of Programming Languages* (2014).

[18] DUFFIELD, N. G., AND GROSSGLAUSER, M. Trajectory sampling for direct traffic observation. *IEEE/ACM Trans. Networking* (June 2001).

[19] FAYAZBAKHSH, S. K., SEKAR, V., YU, M., AND MOGUL, J. C. Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags. In *Proc. USENIX NSDI* (2014).

[20] FELDMANN, A., GREENBERG, A., LUND, C., REINGOLD, N., AND REXFORD, J. NetScope: Traffic engineering for IP networks. *IEEE Network* (2000).

[21] FELDMANN, A., GREENBERG, A., LUND, C., REINGOLD, N., REXFORD, J., AND TRUE, F. Deriving traffic demands for operational IP networks: Methodology and experience. *IEEE/ACM Trans. Networking* (June 2001).

[22] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: A network programming language. In *Proc. ACM International Conference on Functional Programming* (2011).

[23] GUPTA, A., VANBEVER, L., SHAHBAZ, M., DONOVAN, S. P., SCHLINKER, B., FEAMSTER, N., REXFORD, J., SHENKER, S., CLARK, R., AND KATZ-BASSETT, E. SDX: A software defined Internet exchange. In *Proc. ACM SIGCOMM* (2014).

[24] HANDIGOL, N., HELLER, B., JEYAKUMAR, V., MAZIÉRES, D., AND MCKEOWN, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proc. USENIX NSDI* (2014).

[25] HOPPS, C. Analysis of an equal-cost multi-path algorithm, 2000. RFC 2992 [Online, Retrieved September 10, 2015] http://www.ietf.org/rfc/rfc2992.txt.

[26] HUANG, D. Y., YOCUM, K., AND SNOEREN, A. C. High-fidelity switch models for software-defined network emulation. In *Proc. Hot Topics in Software Defined Networks* (2013).

[27] HUEBSCH, R., HELLERSTEIN, J. M., LANHAM, N., LOO, B. T., SHENKER, S., AND STOICA, I. Querying the Internet with PIER. In *Proc. International Conference on Very Large Data Bases (VLDB)* (2003).

[28] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ZOLLA, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. B4: Experience with a globally deployed software defined WAN. In *Proc. ACM SIGCOMM* (2013).

[29] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ZOLLA, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. B4: Experience with a Globally-deployed Software Defined WAN. In *Proc. ACM SIGCOMM* (2013).

[30] JEYAKUMAR, V., ALIZADEH, M., GENG, Y., KIM, C., AND MAZIÈRES, D. Millions of little minions: Using packets for low latency network programming and visibility. In *Proc. ACM SIGCOMM* (2014).

[31] JOSE, L., YAN, L., VARGHESE, G., AND MCKEOWN, N. Compiling Packet Programs to Reconfigurable Switches. In *Proc. USENIX NSDI* (2015).

[32] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *Proc. USENIX NSDI* (2012).

[33] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying network-wide invariants in real time. In *Proc. USENIX NSDI* (2013).

[34] KIM, C., SIVARAMAN, A., KATTA, N., BAS, A., DIXIT, A., AND WOBKER, L. J. In-band Network Telemetry via Programmable Dataplanes, June 2015. Demo at Symposium on SDN Research, http://opennetsummit.org/wp-content/themes/ONS/files/sosr-demos/sosr-demos15-final17.pdf.

[35] LIN, C.-C., CAESAR, M., AND VAN DER MERWE, K. Toward Interactive Debugging for ISP Networks. In *Proc. ACM Workshop on Hot Topics in Networking* (2009).

[36] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* (2008).

[37] MILLER, R. B. Response Time in Man-computer Conversational Transactions. In *Proc. the December 9-11, 1968, Fall Joint Computer Conference, Part I* (1968).

[38] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing Software-Defined Networks. In *Proc. USENIX NSDI* (2013).

[39] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Proc. ACM SIGCOMM* (2014).

[40] NARAYANA, S., REXFORD, J., AND WALKER, D. Compiling path queries in software-defined networks. In *Proc. Hot Topics in Software Defined Networks* (2014).

[41] NELSON, TIM AND YU, DA AND LI, YIMING AND FONSECA, RODRIGO AND KRISHNAMURTHI, SHRIRAM. Simon: Scriptable interactive monitoring for sdns. In *Proc. ACM Symposium on SDN Research* (2015).

[42] NFDUMP TOOL SUITE. [Online, Retrieved September 20, 2015] `http://nfdump.sourceforge.net/`.

[43] OPENFLOW V1.3 SPECIFICATION. [Online, Retrieved September 19, 2015] `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf`.

[44] OPENVSWITCH. [Online, Retrieved August 15, 2015] `openvswitch.org`.

[45] OPENVSWITCH NICIRA EXTENSIONS. [Online, Retrieved September 10, 2015] `http://git.openvswitch.org/cgi-bin/gitweb.cgi?p=openvswitch;a=blob;f=include/openflow/nicira-ext.h`.

[46] PATEL, P., BANSAL, D., YUAN, L., MURTHY, A., GREENBERG, A., MALTZ, D. A., KERN, R., KUMAR, H., ZIKOS, M., WU, H., KIM, C., AND KARRI, N. Ananta: Cloud Scale Load Balancing. In *Proc. ACM SIGCOMM* (2013).

[47] PATH QUERIES FOR INTERACTIVE NETWORK DEBUGGING. [Online, Retrieved September 19, 2015] `http://www.cs.princeton.edu/~narayana/pathqueries`.

[48] PEYMAN KAZEMIAN. Hassel: header space library. [Online, Retrieved September 20, 2015] `https://bitbucket.org/peymank/hassel-public/wiki/Home`.

[49] POSTEL, J. IP Record Route (Internet Protocol), 1981. RFC 791 [Online, Retrieved September 10, 2015] `http://www.ietf.org/rfc/rfc791.txt`.

[50] PYPY. [Online, Retrieved September 10, 2015] `http://pypy.org`.

[51] QAZI, Z. A., TU, C.-C., CHIANG, L., MIAO, R., SEKAR, V., AND YU, M. Simple-fying middlebox policy enforcement using sdn. In *Proc. ACM SIGCOMM* (2013).

[52] QUOITIN, B., VAN DEN SCHRIECK, V., FRANÃĞOIS, P., AND BONAVENTURE, O. IGen: Generation of router-level Internet topologies through network design heuristics. In *Proc. 21st International Teletraffic Congress, 2009* (2009).

[53] 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 *Proc. ACM SIGCOMM* (2014).

[54] REITBLATT, M., CANINI, M., GUHA, A., AND FOSTER, N. FatTire: Declarative Fault Tolerance for Software-defined Networks. In *Proc. Hot Topics in Software Defined Networks* (2013).

[55] ROCKETFUEL: AN ISP TOPOLOGY MAPPING ENGINE. [Online, Retrieved September 22, 2015] `http://research.cs.washington.edu/networking/rocketfuel/interactive/`.

[56] SAVAGE, S., WETHERALL, D., KARLIN, A., AND ANDERSON, T. Practical network support for IP traceback. In *Proc. ACM SIGCOMM* (2000).

[57] SCHLESINGER, C., GREENBERG, M., AND WALKER, D. Concurrent NetCore: From policies to pipelines. In *Proc. ACM International Conference on Functional Programming* (2014).

[58] SHIEH, A., SIRER, E. G., AND SCHNEIDER, F. B. NetQuery: A knowledge plane for reasoning about network properties. In *Proc. ACM SIGCOMM* (2011).

[59] SMOLKA, S., ELIOPOULOS, S., FOSTER, N., AND GUHA, A. A Fast Compiler for NetKAT. In *Proc. ACM International Conference on Functional Programming* (2015).

[60] SNOEREN, A. C., PARTRIDGE, C., SANCHEZ, L. A., JONES, C. E., TCHAKOUNTIO, F., KENT, S. T., AND STRAYER, W. T. Hash-based IP traceback. In *Proc. ACM SIGCOMM* (2001).

[61] SOULÉ, R., BASU, S., MARANDI, P. J., PEDONE, F., KLEINBERG, R., SIRER, E. G., AND FOSTER, N. Merlin: A language for provisioning network resources. In *Proc. ACM CoNEXT* (Dec. 2014).

[62] SPRING, N., MAHAJAN, R., AND WETHERALL, D. Measuring ISP Topologies with Rocketfuel. In *Proc. ACM SIGCOMM* (2002).

[63] SUN, P., YU, M., FREEDMAN, M. J., AND REXFORD, J. Identifying Performance Bottlenecks in CDNs Through TCP-level Monitoring. In *Proc. of the First ACM SIGCOMM Workshop on Measurements Up the Stack* (2011).

[64] SUNG, Y.-W. E., RAO, S. G., XIE, G. G., AND MALTZ, D. A. Towards systematic design of enterprise networks. In *Proc. ACM CoNEXT* (2008).

[65] TAMMANA, P., AGARWAL, R., AND LEE, M. Cherrypick: Tracing packet trajectory in software-defined datacenter networks. In *Proc. ACM Symposium on SDN Research* (2015).

[66] THE PYRETIC LANGUAGE AND RUN-TIME SYSTEM. [Online, Retrieved September 19, 2015] `https://github.com/frenetic-lang/pyretic`.

[67] UDDIN, M. Real-time search in large networks and clouds, 2013.

[68] WANG, R., BUTNARIU, D., AND REXFORD, J. Openflow-based server load balancing gone wild. In *Proc. USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services* (2011).

[69] WAXMAN, B. Routing of multipoint connections. *IEEE J. on Selected Areas in Communications* (1988).

[70] WIRESHARK. [Online, Retrieved September 10, 2015] `https://www.wireshark.org`.

[71] WU, W., WANG, G., AKELLA, A., AND SHAIKH, A. Virtual network diagnosis as a service. In *Proc. Symposium on Cloud Computing (SOCC)* (2013).

[72] 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 *Proc. USENIX NSDI* (2011).

[73] YU, M., JOSE, L., AND MIAO, R. Software Defined Traffic Measurement with OpenSketch. In *Proc. USENIX NSDI* (2013).

[74] YUAN, L., CHUAH, C.-N., AND MOHAPATRA, P. ProgME: Towards Programmable Network Measurement. In *Proc. ACM SIGCOMM* (2007).

[75] ZHANG, H., LUMEZANU, C., RHEE, J., ARORA, N., XU, Q., AND JIANG, G. Enabling layer 2 pathlet tracing through context encoding in software-defined networking. In *Proc. Hot Topics in Software Defined Networks* (2014).

[76] ZHANG, Y., ROUGHAN, M., DUFFIELD, N., AND GREENBERG, A. Fast accurate computation of large-scale IP traffic matrices from link loads. In *Proc. ACM SIGMETRICS* (2003).

[77] ZHANG, HARVEST AND REICH, JOSHUA AND REXFORD, JENNIFER. Packet traceback for software-defined networks. Tech. rep., Princeton University, 2015. [Online, Retrieved September 10, 2015] `https://www.cs.princeton.edu/research/techreps/TR-978-15`.

[78] ZHU, Y., KANG, N., CAO, J., GREENBERG, A., LU, G., MAHAJAN, R., MALTZ, D., YUAN, L., ZHANG, M., ZHAO, B., AND ZHENG, H. Packet-level telemetry in large data-center networks. In *Proc. ACM SIGCOMM* (2015).

A Multi-stage rule-packing problem

Below we write down the integer optimization problem that minimizes the number of table stages subject to constraints on the number of stages and rule space available per stage. Typically, predicate partitioning time is proportional to the size of the output set of predicates, so this also reduces the compile time significantly:

`minimize: S =` $\sum_j y_j$
`variables:` $q_{ij} \in \{0,1\}, y_j \in \{0,1\}$
`subject to:`
$\forall j:\ \text{cost}(\{q_{ij} : q_{ij} = 1\}) \leq \text{rulelimit} * y_j$
$\forall i:\ \sum_j q_{ij} = 1$
$S \leq \text{stagelimit}$

Here the variable q_{ij} is assigned a value 1 if query `i` is assigned to stage `j`, and 0 otherwise. The variable y_j is 1 if stage `j` is used by at least one query and 0 otherwise. The constraints ensure, respectively, that (i) queries in a given stage respect the rule space limits for that stage, (ii) every query is assigned exactly one table stage, and that (iii) the total number of stages is within the number of maximum stages supported by the switch. The optimization problem minimizes the number of used table stages, which is a measure of the latency and complexity of the packet-processing pipeline.

We now write down the cost function that determines the rule space usage of a bunch of queries together. First, we define the *type* and *count* for each query as the set of header fields the query matches on, and the number of total matches respectively. In the example in §7, the query types and counts would be `q1: (['srcip'], 100), q2: (['dstip'], 200), q3: (['srcip'], 300)`. We estimate the *worst-case* rule space cost[10] of putting two queries together into one stage as follows:

```
cost ((type1, count1), (type2, count2)) :=
  case type1 == φ:
    count2 + 1
  case type1 == type2:
    count1 + count2
  case type1 ⊂ type2:
    count1 + count2
  case type1 ∩ type2 == φ:
    (count1 + 1) * (count2 + 1) - 1
  case default:
    (count1 + 1) * (count2 + 1) - 1
```

The type of the resulting query is `type1` ∪ `type2`, as the predicate partioning (Alg. 1) creates matches with headers involving the union of the match fields in the two queries. Hence, we can construct a function which produces a new query type and count, given two existing query types and counts. It is easy to generalize this function to more than two arguments by iteratively applying it to the result of the previous function application and the next query[11]. Hence, we can compute the worst-case rule space cost of putting a bunch of queries together into one stage.

Our cost function and formulation are different from prior works that map logical to physical switch tables [31, 57] for two reasons. First, query predicates can be installed on any table: there is no ordering or dependency between them, so there are more possibilities to explore in our formulation. Second, our rule space cost function explicitly favors predicates with similar headers in one table, while penalizing predicates with very different header matches.

Reduction of bin-packing to rule-packing. It is straightforward to show that the problem of minimizing the number of bins B of capacity V while packing n items of sizes $a_1, a_2, \cdots, a_n$ can be solved through a specific instance of the rule packing problem above. We construct n queries of the same type, with rule counts $a_1, \cdots, a_n$ respectively. We set the `rulelimit` to the size of the bins V, and `stagelimit` to the number of maximum bins allowed in the bin packing problem (typically n). Since all queries are of the same type, the rule space cost function is just the sum of the rule counts of the queries at a given stage. It is then easy to see that the original bin-packing problem is solved by this instance of the rule-packing problem.

First-fit Heuristic. The first-fit heuristic we use is directly derived from the corresponding heuristic for bin-packing. We fit a query into the first stage that allows

[10] It is in general difficult to compute the exact rule space cost of installing two queries together in one stage without actually doing the entire overlap computation in Alg. 1.

[11] We believe, but are yet to show formally, that this cost function is associative.

Algorithm 1 Predicate partitioning (§6.1).

```
1:  P = set_of_predicates
2:  S = ∅
3:  for new_pred ∈ P do
4:      for pred ∈ S do
5:          if pred is equal to new_pred then
6:              continue the outer loop
7:          else if pred is a superset of new_pred then
8:              difference = pred & ∼ new_pred
9:              S ← S ∪ {difference, new_pred}
10:             S ← S \ {pred}
11:             continue the outer loop
12:         else if pred is a subset of new_pred then
13:             new_pred ← new_pred & ∼ pred
14:         else if intersect then
15:             inter1 = pred & ∼ new_pred
16:             inter2 = ∼ pred & new_pred
17:             inter3 = pred & new_pred
18:             S ← S ∪ {inter1, inter2, inter3}
19:             S ← S \ {pred}
20:             new_pred ← new_pred & ∼ pred
21:         end if
22:     end for
23:     S ← S ∪ {new_pred}
24: end for
```

the worst-case rule space blowup to stay within the prespecified per-stage `rulelimit`. The cost function above is used to compute the final rule-space after including a new query in a stage. We use a maximum of 10 logical stages in our experiments, with a 2000 rule limit per stage in the worst-case.

The logical stages match and modify completely disjoint parts of the packet state. We believe that a packet program compiler, *e.g.,* [31], can efficiently lay out the query rules on a physical switch table, since there are no dependencies between these table stages.

B Predicate Partitioning

To ensure that characters represent non-overlapping predicates, we apply Alg. 1 to partition the input predicates. The algorithm takes an input set of predicates `P`, and produces an orthogonal set of predicates `S`.

The partitioned set `S` is initialized to a null set (line 2). We iterate over the predicates in `P`, teasing out overlaps with existing predicates in `S`. If the current input predicate `new_pred` already exists in `S`, we move on to the next input (lines 5-6). If a predicate `pred` in `S` is a superset of `new_pred`, we split `pred` into two parts, corresponding to the parts that do and don't overlap with `new_pred` (lines 7-11). Then we move to the next input predicate. The procedure is symmetrically applied when `pred` is a subset of `new_pred` (lines 12-13), except that we continue looking for more predicates in `S` that may overlap with `new_pred`. Finally, if `pred` and `new_pred` merely intersect (but neither is a superset of the other), we create three different predicates in `S` according to three different combinations of overlap between the two predicates (lines 14-20). Finally, any remaining pieces of `new_pred` are added to the partitioned set `S`. Under each case above and for each predicate in `P`, we also keep track of the predicates in the partitioned set `S` with which it overlaps (details elided).