Compiling Path Queries
Princeton University

Srinivas Narayana  Mina Tahmasbi  Jen Rexford  David Walker
Management = Measure + Control

Network Controller

Measure

Control

Software-Defined Networking (SDN)
Enabling Easier Measurement Matters

• Networks are asked to do a lot!
  • Partition-aggregate applications
  • Growth in traffic demands
  • Stringent performance requirements
  • Avoid expensive outages

• Difficult to know where things go wrong!
  • Humans are slow in troubleshooting
  • Human time is expensive

• Can we build *programmatic tools* to help?
Example: Where’s the Packet Loss?

Suspect: Faulty network device(s) along the way.
Example: Where’s the Packet Loss?

Idea: “Follow” the path of packets through the network.

1000 pkts  850 pkts

ip.src==a & ip.dst==b  
ip.src==a & ip.dst==b'  
Packet rewrite

Switch ACL counters

ip.src==a & ip.dst==b → count
ip.dst==b → fwd port 2

NetFlow Sampling Inaccuracy
Example: Where’s the Packet Loss?

Complex & Inaccurate Join with multiple datasets: traffic, forwarding, topology

High Overhead of collecting (unnecessary) data to answer a given question
Example: Where’s the Packet Loss?

- Complex & Inaccurate Join with multiple datasets: traffic, forwarding, topology
- High Overhead of collecting (unnecessary) data to answer a given question
Pattern: Combining Traffic & Forwarding

- Traffic matrix
- Uneven load balancing
- DDoS source identification
- Port-level traffic matrix
- Congested link diagnosis
- Slice isolation
- Loop detection
- Middlebox traversal order
- Incorrect NAT rewrite
- Firewall evasion
- ...

Resource management
Policy enforcement
Problem diagnosis
Our Approach

Path Query System

Declarative Query Specification
- Independent of Forwarding
- Independent of Other Measurements
- Independent of Hardware Details

Query-Driven Measurement
- Accurate Answers
- Pay Exactly For What You Query
- Commodity ("Match-Action") Hardware

Path Query Language
Query Run-Time System
Our Approach

1. Path Query Language

2. Query Run-Time System

SDN controller

Query Expressions  Forwarding  Statistics

Payloads
How to design general measurement primitives

… that are efficiently implemented in the network?
Measurement Use Cases

• Traffic matrix
• Uneven load balancing
• DDoS source identification
• Port-level traffic matrix
• Congested link diagnosis
• Slice isolation
• Loop detection
• Middlebox traversal order
• Incorrect NAT rewrite
• Firewall evasion
• ...

What are the common patterns?
(I) Path Query Language

• Test predicates on packets at single locations:
  srcip=10.0.0.1
  port=3 & dstip=10.0.1.10

• Combine tests with regular expression operators!
  sw=1 ^ sw=4
  srcip=A ^ true* ^ sw=3
  ingress() ^ ~(sw=firewall)* ^ egress()
Path Query Language

- **Aggregate** results with SQL-like grouping operators
  
  \[\text{in\_group}(\text{ingress}(), [\text{sw}])\]
  
  \[\land \text{true}\]
  
  \[\land \text{out\_group}\!(\text{egress}(), [\text{sw}])\]

- **Return** packets, counters, or samples (NetFlow/sFlow)

<table>
<thead>
<tr>
<th>ingress() switch</th>
<th>#pkts</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>1000</td>
</tr>
<tr>
<td>S2</td>
<td>500</td>
</tr>
<tr>
<td>S5</td>
<td>700</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(ingress(), egress()) switch pairs</th>
<th>#pkts</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S1, S2)</td>
<td>800</td>
</tr>
<tr>
<td>(S1, S5)</td>
<td>200</td>
</tr>
<tr>
<td>(S2, S5)</td>
<td>300</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Example</td>
<td>Query code</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>A simple path</td>
<td><code>in_atom(switch=S1) ~ in_atom(switch=S4)</code></td>
</tr>
<tr>
<td>Slice isolation</td>
<td>`true* ~ (in_out_atom(slice1, slice2)</td>
</tr>
<tr>
<td>Firewall evasion</td>
<td><code>in_atom(ingress()) ~ (in_atom(~switch=FW))*( ~ out_atom(egress()))</code></td>
</tr>
<tr>
<td>DDoS sources</td>
<td><code>in_group(ingress(), [switch]) ~ true* ~ out_atom(egress(), switch=vic)</code></td>
</tr>
<tr>
<td>Switch-level traffic matrix</td>
<td><code>in_group(ingress(), [switch]) ~ true* ~ out_group(egress(), [switch])</code></td>
</tr>
<tr>
<td>Congested link diagnosis</td>
<td><code>in_group(ingress(), [switch]) ~ true* ~ out_atom(switch=ac) ~ in_atom(switch=dc)</code></td>
</tr>
<tr>
<td>Port-to-port traffic matrix</td>
<td><code>in_out_group(switch=s, true, [inport], [outport])</code></td>
</tr>
<tr>
<td>Packet loss localization</td>
<td><code>in_atom(srcip=H1) ~ in_group(true, [switch]) ~ in_group(true, [switch]) ~ out_atom(dstip=H2)</code></td>
</tr>
<tr>
<td>Loop detection</td>
<td><code>port = in_group(true, [switch, inport]); port ~ true* ~ port</code></td>
</tr>
<tr>
<td>Middlebox order</td>
<td><code>(true* ~ in_atom(switch=FW) ~ true*) &amp; (true* ~ in_atom(switch=P) ~ true*) &amp; (true* ~ in_atom(switch=IDS) ~ true*) &amp; ~(in_atom(ingress())) ** in_atom(switch=FW) ** in_atom(switch=IDS) ** out_atom(egress()))</code></td>
</tr>
<tr>
<td>NAT debugging</td>
<td><code>in_out_atom(switch=NAT &amp; dstip=192.168.1.10, dstip=10.0.1.10)</code></td>
</tr>
</tbody>
</table>
| ICMP debugging                   | `in_out_group(switch=S1 & icmp pred)`                                       | Measure ICMP traffic splitting on switch S1 for a small
How do we implement path queries efficiently?

In general, switches don’t know prior or future packet paths.
How to observe packet paths?

- Analyze packet paths *in the data plane* itself
  - Write path information into packets!

```
[{sw: S1, ...
  port: 1
  srcmac: ...
  srcip: ...
  ...}]]
[{sw: S1, ...},
 {sw: S2
  port: 3
  srcmac: ...
  srcip: ...
  ...}]]
[{sw: S1, ...},
 {sw: S2, ...},
 {sw: S3
  port: 2
  srcmac: ...
  srcip: ...
  ...}]]
```

- Pros: accurate trajectory information 😊
- Cons: too much per-packet information 😞
Reducing Path Information on Packets

- Observation 1: Queries already tell us what’s needed!
  - Only record path state needed by queries

- Observation 2: Queries are regular expressions
  - Regular expressions ⇒ Finite automaton (DFA)
  - Distinguish only paths corresponding to DFA states
Reducing Path Information on Packets

Record only DFA state on packets (1-2 bytes)

Use existing “tag” fields! (e.g., VLAN)
(II) Query Run-Time System

\[ (sw=1 \land srcip=A) \land (sw=4 \land dstip=B) \]

Switch 1:
\[ state=Q0 \land srcip=A \rightarrow state=Q1 \]

Switch 4:
\[ state=Q1 \land dstip=B \rightarrow state=Q2 \]

AND count!
(II) Query Run-Time System

• Each packet carries its own DFA state

• Query DFA transitions *distributed* to switches
  • … as match-action rules!

• Packet satisfies query iff it reaches accepting states
  • “Pay for what you query”
(II) Run-Time: Juicy details in paper…

- Packet forwarding shouldn't be affected by DFA rules
  - No unnecessary duplicate traffic should be created

- Handle query overlap
  - Predicates can also overlap

- Handle groupby aggregation

- Capture *upstream* or *downstream* of queried path
  - Test predicates before or after forwarding on switch

- Optimizations: to make the system practical

\[
\text{srcip}=A \land \text{dstip}=B \\
\text{sw}=1 \land \text{true}^* \land \text{sw}=4
\]
Evaluation

• Prototype on Pyretic + NetKAT + OpenVSwitch
  • Publicly available: http://frenetic-lang.org/pyretic/

• Queries: traffic matrix, DDoS detection, per-hop packet loss, firewall evasion, slice isolation, congested link

• Results on Stanford backbone (all queries together):
  • Compile time: 5 seconds (from > 2 hours)
  • # Rules: ~ 650
  • # State bytes: 2 bytes
Evaluation: Scaling

Miller, “Response time in man-computer conversational transactions”
Summary

• We need good abstractions to measure networks
  • Abstractions must be efficiently implementable

• We implemented declarative queries on packet paths:
  • Packet state akin to a deterministic automaton

• Path queries can simplify network management!

Queries? 😊

http://www.cs.princeton.edu/~narayana/pathqueries
https://youtu.be/Vx0aN9iGPWc
(I) Language: Related Work

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
<th>Prior Work</th>
<th>Our Extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic Predicates</td>
<td>Boolean tests on located packets</td>
<td>[Foster11]</td>
<td>Switch input and output differentiation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Monsanto13]</td>
<td></td>
</tr>
<tr>
<td>Packet Trajectories</td>
<td>Regular expressions on atomic predicates</td>
<td>[Tarjan79],</td>
<td>Additional regex operators (&amp;, ~)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Handigol14]</td>
<td></td>
</tr>
<tr>
<td>Result Aggregation</td>
<td>Group results by location or header fields</td>
<td>SQL groupby,</td>
<td>Group anywhere along a path</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Foster11]</td>
<td></td>
</tr>
<tr>
<td>Capture Location</td>
<td>Get packets before or after queried path</td>
<td>--</td>
<td>N/A</td>
</tr>
<tr>
<td>Capture Result</td>
<td>Actions on packets satisfying queries</td>
<td>[Monsanto13]</td>
<td>Sampling (sFlow)</td>
</tr>
</tbody>
</table>
(1) Capture locations

- Capture upstream, downstream or midstream
(II) Run-Time: Data Plane Rule Layout

"In" table >> Forwarding >> "Out" table

state = Q_i & pred: state ← Q_j AND in_capture

forwarding

state = Q_i' & pred: state ← Q_j' AND out_capture
(II) Query Compilation

Use policy composition operators and compiler

DFA-Transitioning >> Forwarding + DFA-Accepting

All acting on the same data plane packets!

Composing software-defined networks. Monsanto et al., 2013
(II) Query Compilation

( DFA- Transitioning >> Forwarding ) + DFA- Accepting

state=Q0 & switch=S1 & srcip=10.0.0.1 → state←Q1
state=Q1 & switch=S2 & dstip=10.0.0.3 → state←Q2

state=Q0 & switch=S1 & srcip=10.0.0.1 & dstip=10.0.0.2 → state←Q1, fwd(2)

dstip=10.0.0.1 → fwd(1)
dstip=10.0.0.2 → fwd(2)
dstip=10.0.0.3 → fwd(3)
...

Composing software-defined networks. Monsanto et al., 2013
(II) Query Compilation

(DFA-Ingress-Transitioning $\gg$ Forwarding $\gg$ DFA-Egress-Transitioning) +
(DFA-Ingress-Accepting)
+
(DFA-Ingress-Transitioning $\gg$ Forwarding $\gg$ DFA-Egress-Accepting)
(II) Detecting Query Overlaps

• Predicate overlaps:
  • q1: srcip=10.0.0.1; q2: dstip=10.0.0.2
  • Automaton can only have one state!

• Query overlaps:
  • q1: sw=1 ^ sw=2
  • q2: srcip=10.0.0.1 ^ dstip=10.0.0.2
  • q1: in_atom(srcip=10.0.0.1)
  • q2: out_atom(srcip=10.0.0.1)

• Automaton states must distinguish all possibilities!
(II) Detecting Query Overlaps

• Predicate overlaps: Generate orthogonal predicates!
  • q1: srcip=10.0.0.1; q2: dstip=10.0.0.2

• Generated predicates:
  • srcip=10.0.0.1 & dstip=10.0.0.2
  • srcip=10.0.0.1 & ~dstip=10.0.0.2
  • ~srcip=10.0.0.1 & dstip=10.0.0.2
(II) Detecting Query Overlaps

• Query Overlaps:

  • Convert in_ and out_ atoms to in_out_atoms:

    • in_atom(srcip=10.0.0.1) \rightarrow in_out_atom(srcip=10.0.0.1, true)

    • out_atom(dstip=10.0.0.1) \rightarrow in_out_atom(true, dstip=10.0.0.1)
(II) Detecting Query Overlaps

- Query Overlaps:

  - Build one DFA for many expressions together
    - in_atom(srcip=H1 & sw=1) ^ out_atom(sw=2 & dstip=H2)
    - in_atom(sw=1) ^ in_out_atom(true, sw=2)
## Optimizations: Summary

<table>
<thead>
<tr>
<th>Optimization</th>
<th># Rules?</th>
<th>Time?</th>
<th># States?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate query &amp; forwarding actions into separate stages</td>
<td>![Down]</td>
<td>![Down]</td>
<td></td>
</tr>
<tr>
<td>Optimize conditional policy compilation</td>
<td>![Down]</td>
<td>![Down]</td>
<td></td>
</tr>
<tr>
<td>Integrate tagging and capture policies</td>
<td>![Down]</td>
<td>![Down]</td>
<td></td>
</tr>
<tr>
<td>Pre-partition predicates by flow space</td>
<td>![Down]</td>
<td>![Down]</td>
<td></td>
</tr>
<tr>
<td>Cache predicate overlap decisions</td>
<td>![Down]</td>
<td>![Down]</td>
<td></td>
</tr>
<tr>
<td>Decompose query predicates into multiple stages</td>
<td>![Down]</td>
<td>![Down]</td>
<td>![Up]</td>
</tr>
<tr>
<td>Detect predicate overlaps with Forwarding Decision Diagrams</td>
<td>![Down]</td>
<td>![Down]</td>
<td></td>
</tr>
</tbody>
</table>
## Benefit of Optimizations (Stanford)

<table>
<thead>
<tr>
<th>Cumulative Optimization</th>
<th>Time (s)</th>
<th># Rules</th>
<th># State Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>&gt; 7900</td>
<td>DNF</td>
<td>DNF</td>
</tr>
<tr>
<td>Separate query &amp; forwarding actions into separate stages</td>
<td>&gt; 4920</td>
<td>DNF</td>
<td>DNF</td>
</tr>
<tr>
<td>Optimize conditional policy compilation</td>
<td>&gt; 4080</td>
<td>DNF</td>
<td>DNF</td>
</tr>
<tr>
<td>Integrate tagging and capture policies</td>
<td>2991</td>
<td>2596</td>
<td>10</td>
</tr>
<tr>
<td>Pre-partition predicates by flow space</td>
<td>56.19</td>
<td>1846</td>
<td>10</td>
</tr>
<tr>
<td>Cache predicate overlap decisions</td>
<td>35.13</td>
<td>1846</td>
<td>10</td>
</tr>
<tr>
<td>Decompose query predicates into multiple stages</td>
<td>5.467</td>
<td>260</td>
<td>16</td>
</tr>
</tbody>
</table>
Demo: Where’s the Packet Loss?
Demo: Where’s the Packet Loss?

https://youtu.be/Vx0aN9iGPWc
Downstream Query Compilation (3/3)

( DFA-Transitioning $\gg$ Forwarding ) + DFA-Accepting

All acting on the same data plane packets!

Use policy composition operators and compiler

Composing software-defined networks. Monsanto et al., 2013
Downstream Query Compilation (3/3)

(DFA-Transitioning) >> Forwarding + DFA-Accepting

state=Q0 & switch=S1 & srcip=10.0.0.1 → state→Q1
state=Q1 & switch=S2 & dstip=10.0.0.3 → state→Q2

dstip=10.0.0.1 → fwd(1)
dstip=10.0.0.2 → fwd(2)
dstip=10.0.0.3 → fwd(3)
...

state=Q0 & switch=S1 & srcip=10.0.0.1 & dstip=10.0.0.2 → state→Q1, fwd(2)

Composing software-defined networks. Monsanto et al., 2013
Downstream Query Compilation (3/3)

(DFA-Ingress-Transitioning >> Forwarding >> DFA-Egress-Transitioning)
+
(DFA-Ingress-Accepting)
+
(DFA-Ingress-Transitioning >> Forwarding >> DFA-Egress-Accepting)
II. Rule Count

Switch TCAM capacity: 2K-4K rules
III. Packet State Bits

![Graph showing the relationship between the number of nodes and the number of state bits for VLAN and MPLS.]