Declarative Network Path Queries

Srinivas Narayana
May 13, 2016

Advisor: Prof. Jennifer Rexford
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.

A

1000 pkts

---

B

850 pkts 😞
Example: Where’s the Packet Loss?

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

Switch ACL counters

- \( ip\.src=a \) & \( ip\.dst=b \) → count
- \( ip\.dst=b \) → fwd port 2

NetFlow Sampling Inaccuracy

1000 pkts

850 pkts 😞
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

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
Approach

1. Path Query Language

2. Query Run-Time System

3. Optimizations
Approach

1. Path Query Language
   - Expressive measurement specification

2. Query Run-Time System
   - Accurate data plane measurement

3. Optimizations
   - Efficient measurement
Contributions

• Regular-expression-based language for traffic monitoring
  • With SQL-like aggregation and capture locations

• Run-Time: Deterministic finite state automata on packets using match-action switch rules
  • Collect *exactly* those packets that satisfy queries

• Compiler optimizations: to speed up or completely remove expensive overlapping actions on packets

• Result: Debug networks with practical overheads
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?

Path Query Language

- Test predicates on packets at single locations:
  \[ \text{srcip}=10.0.0.1 \]
  \[ \text{port}=3 \land \text{dstip}=10.0.1.10 \]

- Combine tests with regular expression operators!
  \[ \text{sw}=1 \lor \text{sw}=4 \]
  \[ \text{srcip}=A \lor \text{true}^* \lor \text{sw}=3 \]
  \[ \text{ingress}() \lor \neg(\text{sw}=\text{firewall})^* \lor \text{egress}() \]
(I) Path Query Language

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

<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>
</tbody>
</table>

- Return packets, counters, or samples (NetFlow/sFlow)
(I) Path Query Language

- *Capture* upstream, downstream or midstream

- *Match* predicates at switch ingress, egress or both
  - `in_atom(dstip=128.1.2.3)`
  - `in_out_atom(dstip=128.1.2.3, dstip=10.1.2.3)`
(I) Evaluation: Query Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Query code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A simple path</td>
<td><code>in_atom(switch=S1) ~ in_atom(switch=S4)</code></td>
<td>Packets going from switch S1 to S4 in the network.</td>
</tr>
<tr>
<td>Slice isolation</td>
<td>`true* ^ (in_out_atom(slice1, slice2)</td>
<td>in_out_atom(slice2, slice1))`</td>
</tr>
<tr>
<td>Firewall evasion</td>
<td><code>in_atom(ingress()) ~ (in_atom(~switch=FW))* ~ out_atom(egress())</code></td>
<td>Catch packets evading a firewall device FW when moving from any network ingress to egress interface.</td>
</tr>
<tr>
<td>DDoS sources</td>
<td><code>in_group(ingress(), [switch]) ~ true* ~ out_atom(egress(), switch=vic)</code></td>
<td>Determine traffic contribution by volume from all ingress switches reaching a DDoS victim switch vic.</td>
</tr>
<tr>
<td>Switch-level</td>
<td><code>in_group(ingress(), [switch]) ~ true* ~ out_group(egress(), [switch])</code></td>
<td>Count packets from any ingress to any egress switch, with results grouped by (ingress, egress) switch pair.</td>
</tr>
<tr>
<td>traffic matrix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Congested link diagnosis</td>
<td><code>in_group(ingress(), [switch]) ~ true* ~ out_atom(switch=dc) ~ in_atom(switch=dc)</code></td>
<td>Determine flows (switch sources \rightarrow sinks) utilizing a congested link (from switch ac to switch dc), to help reroute traffic around the congested link.</td>
</tr>
<tr>
<td>Port-to-port</td>
<td><code>in_out_group(switch=s, true, [inport], [outport])</code></td>
<td>Count traffic flowing between any two ports of switch s, grouping the results by the ingress and egress interface.</td>
</tr>
<tr>
<td>traffic matrix</td>
<td></td>
<td></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>
<td>Localize packet loss by measuring per-path traffic flow along each 4-hop path between hosts H1 and H2.</td>
</tr>
<tr>
<td>Loop detection</td>
<td><code>port = in_group(true, [switch, inport]); port ~ true* ~ port</code></td>
<td>Detect packets that visit any fixed switch and port twice in their trajectory.</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=P) ~ in_atom(switch=IDS) ~ out_atom(egress()))</code></td>
<td>Packets that traverse a firewall FW, proxy P and intrusion detection device IDS, but do so in an undesirable order [51].</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>
<td>Catch packets entering a NAT with destination IP 192.168.1.10 and leaving with the (modified) destination IP 10.0.1.10.</td>
</tr>
<tr>
<td>ECMP debugging</td>
<td><code>in_out_group(switch=S1, enp pred)</code></td>
<td>Measure ECMP traffic splitting on switch S1 for a small</td>
</tr>
</tbody>
</table>
## (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], [Monsanto13]</td>
<td>Switch input and output differentiation</td>
</tr>
<tr>
<td>Packet Trajectories</td>
<td>Regular expressions on atomic predicates</td>
<td>[Tarjan79], [Handigol14]</td>
<td>Additional regex operators (&amp;, ~)</td>
</tr>
<tr>
<td>Result Aggregation</td>
<td>Group results by location or header fields</td>
<td>SQL groupby, [Foster11]</td>
<td>Group anywhere along a path</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); path-based forwarding</td>
</tr>
</tbody>
</table>
How do we implement path queries efficiently?

In general, switches don’t know prior or future packet *paths*.
Match-Action Packet Processing

(Ternary) Bit Pattern, e.g.,
srcip=A

Forward/Drop/Modify
dstip=B; fwd(2)

match1 → action1
match2 → action2
...

Multiple but limited # stages (e.g., 16)
Limited # rules per stage (e.g., 2K)
How to observe pkt paths downstream?

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

  ![Diagram showing packet paths through switches](image)

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

• Pros: accurate path information 😊
• Cons: too much per-packet information 😞
• Cons: can’t match regular expressions on switches
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 \(\Rightarrow\) 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

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

Switch 1:
state=Q0 \& srcip=A \rightarrow state=Q1

Switch 4:
state=Q1 \& dstip=B \rightarrow state=Q2
AND count!
(II) Query Run-Time System

• Each packet carries its own DFA state

• Query DFA transitions \textit{distributed} to switches
  • … as \textit{match-action} rules!

• Packet satisfies query iff it reaches accepting states
  • “Pay for what you query”
(II) You Pay For What You Query

![Graph showing ratio of overhead to total and ratio of firewall-evading packets to egress packets for all packets, every hop path query.]

- Red line: all packets, every hop path query
- Black line: firewall-evading packets
- Blue line: firewall-evading packets vs egress packets
(II) Run-Time: Deterministic Transitions

- $p_1$: $sw=S_1$
- $p_2$: $dstip=10.0.0.2$

Switch $S_1$
(II) Run-Time: Deterministic Transitions

- p1: $sw=S1$
- p2: $dstip=10.0.0.2$
- **Trouble:** Packet should only be in one automaton state!
(II) Run-Time: Deterministic Transitions

- $p_1$: $sw=S_1$
- $p_2$: $dstip=10.0.0.2$
- **Solution**: Split predicates into disjoint parts

![Diagram]

- $Q_0$:
  - $dstip=10.0.0.2$
  - $Switch\; S_1$

- $Q_1$:
  - $p_1 \& \neg p_2$
  - accept $p_1$ only

- $Q_2$:
  - $\neg p_1 \& p_2$
  - accept $p_2$ only

- $Q_3$:
  - $p_1 \& p_2$
  - accept $p_1$ and $p_2$

- $Q_4$:
  - $\neg p_1 \& \neg p_2$
  - dead
(II) Run-Time: Composition

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
A fast compiler for NetKAT. Smolka et al., 2015
(II) Run-Time: Composition

DFA-Transitioning >> Forwarding + DFA-Accepting

state=Q0 & switch=S1 & srcip=A => state=Q1
state=Q1 & switch=S4 & dstip=A => state=Q2

state=Q0 & switch=S1 & srcip=A & dstip=B => state=Q1, fwd(2)

dstip=A => fwd(1)
dstip=B => fwd(2)
dstip=C => fwd(3)
...

Openflow 1.0 (for example)

Composing software-defined networks. Monsanto et al., 2013
A fast compiler for NetKAT. Smolka et al., 2015
(II) Run-Time: Generate Switch Rules

Bit pattern:
state=Q0 & switch=S1 &
srcip=A & dstip=B

Forward/Drop/Modify
state←Q1;
_fwd(2)

match1 ⇒ action1
match2 ⇒ action2
...

Result: unified switch rules for forwarding and measurement
(II) Run-Time: Other details in paper...

- Handle groupby aggregation
- Testing predicates before and after forwarding
- Upstream query compilation
## (II) Run-Time: Related Work

<table>
<thead>
<tr>
<th>Approach</th>
<th>Expressiveness</th>
<th>Sources of inaccuracy</th>
<th>Sources of overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Policy checking (§1.5.1)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Header space analysis [52, 53]</td>
<td>Locations and headers</td>
<td>No actual packets</td>
<td>Policy analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Only control plane view</td>
<td></td>
</tr>
<tr>
<td><strong>Out-of-band approaches (§1.5.2)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Infer using traffic matrix [32, 119]</td>
<td>Switch-level paths</td>
<td>Forwarding dynamism</td>
<td>Load collection [21]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Downstream packet drop</td>
<td>Traffic collection [1, 14, 87]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opaque multipath routing</td>
<td></td>
</tr>
<tr>
<td>Upstream inference [53, 121]</td>
<td>Locations and headers</td>
<td>Ambiguous upstream path</td>
<td>Traffic collection [1, 14, 87]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Packet modification</td>
<td>Policy analysis</td>
</tr>
<tr>
<td>Join per-hop info [27, 40, 96, 122]</td>
<td>Locations and headers</td>
<td>Ambiguous packet joins</td>
<td>Packet digests (every hop)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Topological sort</td>
</tr>
<tr>
<td><strong>In-band approaches (§1.5.3)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Record interfaces [83, 90]</td>
<td>Interface-level paths</td>
<td>Record few interfaces</td>
<td>Packet space for interfaces</td>
</tr>
<tr>
<td>Path tracing [102, 118]</td>
<td>Interface-level paths</td>
<td>Strong assumptions</td>
<td>Packet space for interfaces</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data plane rules</td>
</tr>
<tr>
<td><strong>Our approach (§1.6)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFA on packet state [65, 66]</td>
<td>Locations and headers</td>
<td><em>None</em></td>
<td>Packet space for DFA state</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data plane rules</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Query compile time</td>
</tr>
</tbody>
</table>
How well does it work?
Evaluation of initial prototype

- 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

- Run all queries together on Stanford backbone
  - Compile time: > 2 hours
  - Switch rules: (estimated per switch) 1M
  - Packet state: 10 bits
Problem: Cross-Products

\[(\text{DFA-Transitioning}) \quad \Rightarrow \quad \text{Forwarding} \quad \Rightarrow \quad \text{DFA-Accepting}\]

- \(\text{state} = Q_0 \quad \& \quad \text{switch} = S_1 \quad \& \quad \text{srcip} = A \quad \Rightarrow \quad \text{state} \leftarrow Q_1\)
- \(\text{state} = Q_1 \quad \& \quad \text{switch} = S_4 \quad \& \quad \text{dstip} = A \quad \Rightarrow \quad \text{state} \leftarrow Q_2\)
- \(\text{dstip} = A \quad \Rightarrow \quad \text{fwd}(1)\)
- \(\text{dstip} = B \quad \Rightarrow \quad \text{fwd}(2)\)
- \(\text{dstip} = C \quad \Rightarrow \quad \text{fwd}(3)\)
- \(\ldots\)

- \(\text{state} = Q_0 \quad \& \quad \text{switch} = S_1 \quad \& \quad \text{srcip} = A \quad \& \quad \text{dstip} = B \quad \Rightarrow \quad \text{state} \leftarrow Q_1, \text{fwd}(2)\)
Problem: Cross-Products

- p1: sw=S1
- p2: dstip=10.0.0.2

Q0 \rightarrow Q1 \rightarrow Q2

dstip 10.0.0.2

Q1

sw=S1

Q2

accept p1

Q0

accept p2

Q1

only

Q2

only

Q3

and p2

Q4

dead
Complexity From Overlaps

# Ingress Rules

# Query predicates

![Graph showing complexity from overlaps with overlapping and nonoverlapping query predicates.](image-url)
Complexity From Overlaps

![Graph showing the relationship between compile time (s) and the number of query predicates, with separate lines for overlapping and nonoverlapping queries.](image)
(III) Optimizations: Reduce Pkt Overlap

- *Construct* non-overlapping policies
  - Use structure of generated Pyretic policies

<table>
<thead>
<tr>
<th>Remove overlapping actions on packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Use pipelined packet processing</td>
</tr>
</tbody>
</table>

- *Speed up* detection of overlapping actions
  - Use better data structures & caching
### Optimization 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><img src="down.png" alt="Down" /></td>
<td><img src="down.png" alt="Down" /></td>
<td></td>
</tr>
<tr>
<td>Optimize conditional policy compilation</td>
<td><img src="down.png" alt="Down" /></td>
<td><img src="down.png" alt="Down" /></td>
<td></td>
</tr>
<tr>
<td>Integrate tagging and capture policies</td>
<td><img src="down.png" alt="Down" /></td>
<td><img src="down.png" alt="Down" /></td>
<td></td>
</tr>
<tr>
<td>Pre-partition predicates by flow space</td>
<td><img src="down.png" alt="Down" /></td>
<td><img src="down.png" alt="Down" /></td>
<td></td>
</tr>
<tr>
<td>Cache predicate overlap decisions</td>
<td><img src="down.png" alt="Down" /></td>
<td><img src="down.png" alt="Down" /></td>
<td></td>
</tr>
<tr>
<td>Decompose query predicates into multiple stages</td>
<td><img src="down.png" alt="Down" /></td>
<td><img src="down.png" alt="Down" /></td>
<td><img src="up.png" alt="Up" /></td>
</tr>
<tr>
<td>Detect predicate overlaps with Forwarding Decision Diagrams</td>
<td><img src="down.png" alt="Down" /></td>
<td><img src="down.png" alt="Down" /></td>
<td></td>
</tr>
</tbody>
</table>
## (III) 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>![down]</td>
</tr>
<tr>
<td>Optimize conditional policy compilation</td>
<td>![down]</td>
<td>![down]</td>
<td>![down]</td>
</tr>
<tr>
<td>Integrate tagging and capture policies</td>
<td>![down]</td>
<td>![down]</td>
<td>![down]</td>
</tr>
<tr>
<td>Pre-partition predicates by flow space</td>
<td>![down]</td>
<td>![down]</td>
<td>![down]</td>
</tr>
<tr>
<td>Cache predicate overlap decisions</td>
<td>![down]</td>
<td>![down]</td>
<td>![down]</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>![down]</td>
</tr>
</tbody>
</table>
(III) Separate Queries from Forwarding

\[(\text{DFA-Transition} \gg \text{Forwarding}) + \text{DFA-Accept} = (\text{DFA-Transition} + \text{DFA-Accept}) \gg \text{Forwarding}\]
(III) Separate Queries from Forwarding

\[(DFA\text{-}\text{Ingress}\text{-}\text{Transitioning} \gg \text{Forwarding} \gg DFA\text{-}\text{Egress}\text{-}\text{Transitioning})\]
\[+\]
\[(DFA\text{-}\text{Ingress}\text{-}\text{Accepting})\]
\[+\]
\[(DFA\text{-}\text{Ingress}\text{-}\text{Transitioning} \gg \text{Forwarding} \gg DFA\text{-}\text{Egress}\text{-}\text{Accepting})\]

==

\[(DFA\text{-}\text{Ingress}\text{-}\text{Transitioning} + DFA\text{-}\text{Ingress}\text{-}\text{Accepting})\]
\[\gg\]
Forwarding
\[\gg\]
\[(DFA\text{-}\text{Egress}\text{-}\text{Transitioning} + DFA\text{-}\text{Egress}\text{-}\text{Accepting})\]
(III) Separating Queries

- Could we run queries in a pipelined fashion?
(III) Separating Queries

• $p_1$: $sw=S1$; $p_2$: $dstip=10.0.0.2$; $p_3$: $dstip=10.0.0.3$
(III) Separating Queries

- p1: sw=S1; p2: dstip=10.0.0.2; p3: dstip=10.0.0.3
- **Problem:** Limited # table stages & rules per stage
(III) Separating Queries

- \( p_1: \text{sw}=S_1; \ p_2: \text{dstip}=10.0.0.2; \ p_3: \text{dstip}=10.0.0.3 \)

- Idea: Group queries by their “similarity”
  - \( p_1 \) in one stage, \( p_2 \) and \( p_3 \) in another
(III) Cost Function for Query Similarity

- Input: a set of queries
- Output: estimate # rules if queries in same table stage

\[
\text{cost } ((\text{type1}, \text{count1}), (\text{type2}, \text{count2})) := \\
\begin{cases} 
  \text{case type1 } = \varnothing: \\
  & \text{count2 } + 1 \\
  \text{case type1 } = \text{type2}: \\
  & \text{count1 } + \text{count2} \\
  \text{case type1 } \subseteq \text{type2}: \\
  & \text{count1 } + \text{count2} \\
  \text{case type1 } \cap \text{type2 } = \varnothing: \\
  & (\text{count1 } + 1) * (\text{count2 } + 1) - 1 \\
  \text{case default}: \\
  & (\text{count1 } + 1) * (\text{count2 } + 1) - 1
\end{cases}
\]
(III) Cost-Aware Query Grouping

- Minimize total # stages \[ S = \sum_j y_j \]

- Subject to:
  - Rule space per stage \[ \text{cost}(\{q_{ij} : q_{ij} = 1\}) \leq \text{rulelimit} \cdot y_j \]
  - Total number of stages \[ S \leq \text{stagelimit} \]
  - One query \( \rightarrow \) one stage \[ \forall i : \sum_j q_{ij} = 1 \]

- Variables (binary integers)
  - Stage \( j \) assigned \( q_{ij} \in \{0, 1\}, y_j \in \{0, 1\} \)
  - Query \( i \) assigned to \( j \)
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

• Run all queries together on Stanford backbone
  • Compile time: > 2 hours $\Rightarrow$ 5 seconds
  • Switch rules: (estimated) 1M $\Rightarrow$ (actual) ~1K
  • Packet state: 10 bits $\Rightarrow$ 16 bits
## 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>
Scalability Trends

• Five synthetic ISP (Waxman) topologies at various network sizes

• At each network size, run mix of queries from before
  
  • Averaged metrics across queries & topologies
Evaluation: Scaling

Miller, “Response time in man-computer conversational transactions”
II. Rule Count

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

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

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

• Query-driven measurement: a useful principle
  • Improves accuracy; and
  • Reduces overheads

• Challenge: finding sufficiently general families of questions with efficient solution techniques

• Path queries can simplify network management!
Thanks! 😊
Demo: Where’s the Packet Loss?

1000 pkts

850 pkts 😞
Demo: Where’s the Packet Loss?

https://youtu.be/Vx0aN9iGPWc
Discussion: Questions

• Control plane versus data plane checking

• Switch performance impact (throughput, delay…)
  • Table stages
  • Memory on the switch
  • Memory on the packet

• Comparison to existing SDN approaches

• System evaluation
Discussion: Extensions

• Multi-packet queries?
  • Performance, security, …
  • What language abstractions? What hardware?

• Post-facto queries

• Improving compiler performance
Approach 1: Join Traffic & Forwarding

Traffic dataset
- e.g., NetFlow, SNMP

Forwarding updates
- e.g., OF/routing protocol updates

Dynamic
- Timestamps not aligned!

Ambiguity between identical packets downstream

Packet traceback for software-defined networks. Zhang et al., 2015
Approach 1: Join Traffic & Forwarding

Packet rewriting compounds the ambiguity!

Ambiguity between identical packets downstream

Dynamic
Timestamps not aligned!

Traffic dataset

Traffic dataset

e.g., NetFlow, SNMP

Forwarding updates

Forwarding updates

e.g., OF/routing protocol updates

Trajectory sampling for direct traffic observation. Duffield et al., 2001
Approach 2: Collect at Every Hop

Using packet histories to troubleshoot networks. Handigol et al., 2014
Hash-based IP traceback. Snoeren et al., 2001
Approach 2: Collect at Every Hop

Using packet histories to troubleshoot networks. Handigol et al., 2014
Hash-based IP traceback. Snoeren et al., 2001
Approach 2: Collect at Every Hop

Too expensive to collect up front!

Sampling to reduce overhead may miss the packets you care about…

Trajectory sampling for direct traffic observation. Duffield et al., 2001
Approach 3: Write Path into Packet

Switches have very accurate info on prior packet path 😊
Match-action HW can’t match regexes :-(
Too much info on packet! :-(

sw: S1, srcip: ___, dstip: ___, ...
sw: S2, srcip: ___, dstip: ___,
    sw: S1, ...;
sw: S3, ...;
sw: S2, ...;
sw: S1, ...;

IP record route, RFC 791. Postel, 1981
Tracing packet trajectory in data-center networks. Tammana et al., 2015
Reachability Testing for Accepted Pkts

Static checking for networks, Kazemian et al. NSDI ‘12
Reachability Testing for Accepted Pkts

"Effective" policy after downstream compilation of query q

state=accepted(q)

Static checking for networks, Kazemian et al. NSDI ‘12
Complexity from Overlaps

![Graph showing the relationship between the number of DFA states and the number of query predicates for overlapping and nonoverlapping cases. The graph indicates a significant increase in DFA states with an increase in the number of query predicates, particularly for overlapping cases.]