SONATA: Scalable Streaming Analytics for Network Telemetry

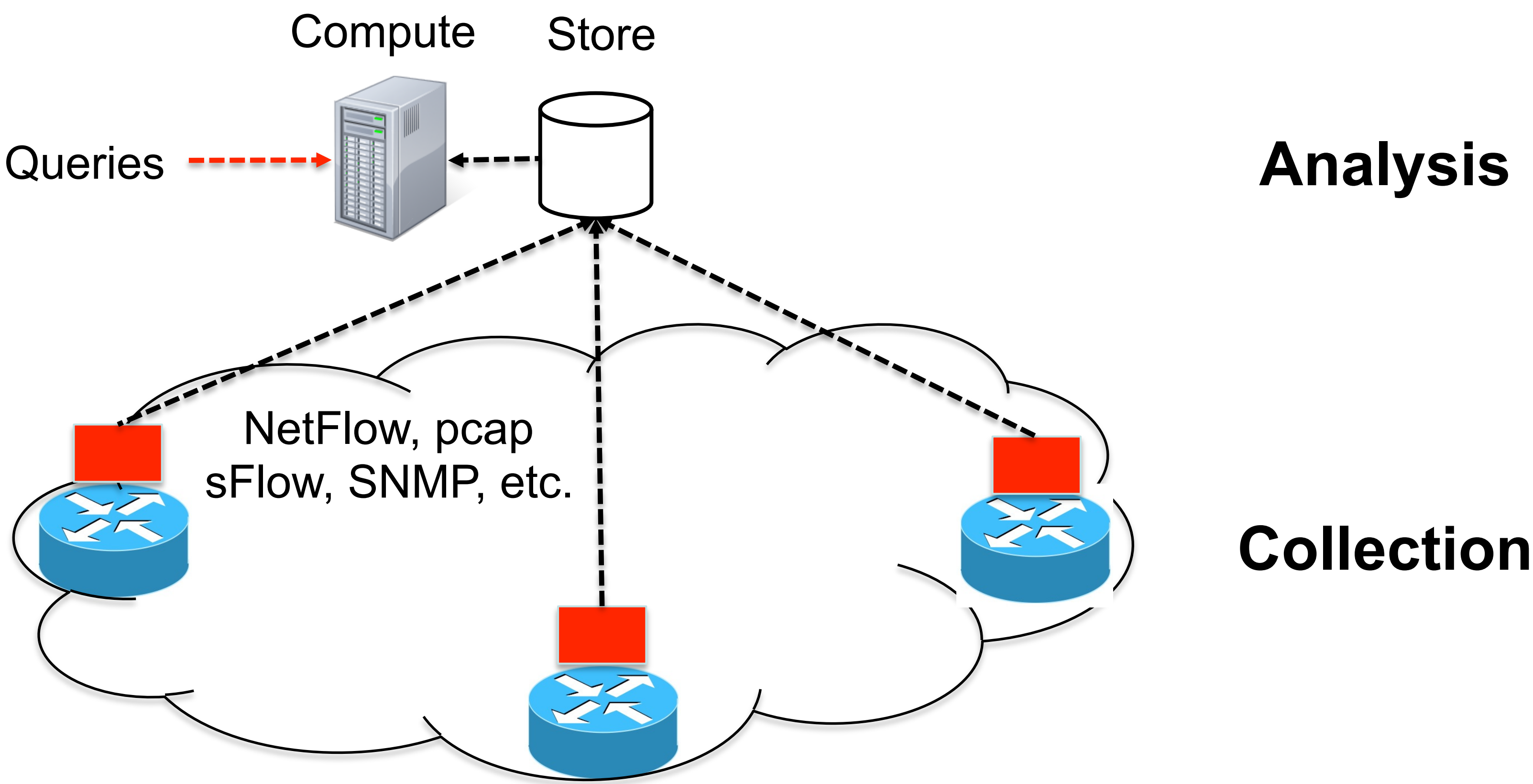
Arpit Gupta

Princeton University

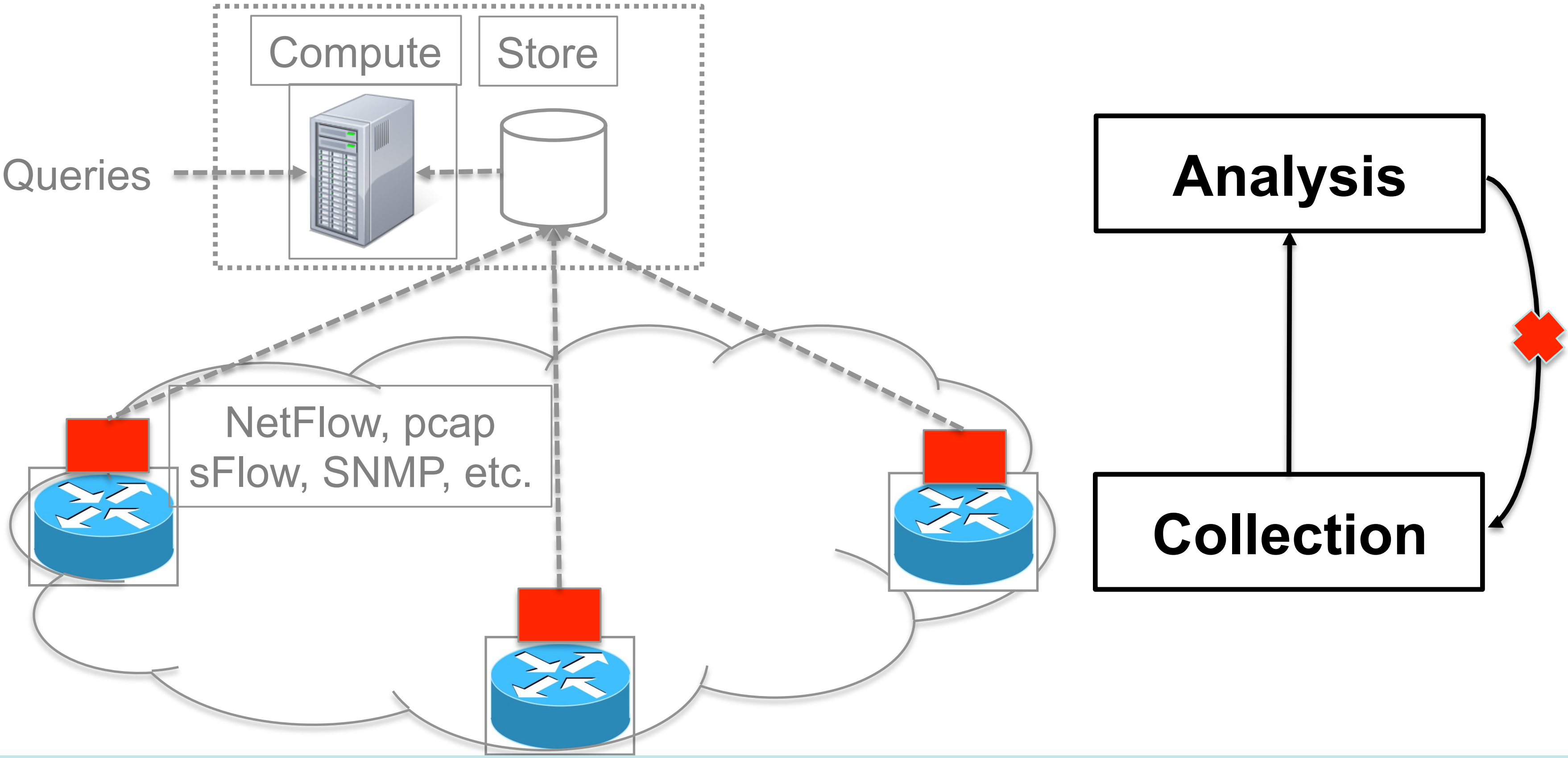
Rob Harrison, Ankita Pawar, Rüdiger Birkner,

Marco Canini, Nick Feamster, Jennifer Rexford, Walter Willinger

Conventional Network Telemetry



Conventional Network Telemetry



Collection is **not** driven by Analysis

Problems with Status Quo

- ***Expressibility***
  - Configure collection & analysis stages separately
  - Static (and often coarse) data collection
  - Brittle analysis setup---specific to collection tools

Problems with Status Quo

- *Expressibility*
  - Configure collection & analysis stages separately
  - Static (and often coarse) data collection
  - Brittle analysis setup---specific to collection tools

- ***Scalability***

  As Traffic Volume or # Monitoring Queries increases
    - Hard to answer queries in real-time

Hard to **express** & **scale** queries for network telemetry tasks!

SONATA: Query-Driven Telemetry

- ***Idea 1: Uniform Programming Abstraction***

  Express queries as dataflow operations over pkt. tuples

- ***Idea 2: Query Partitioning***

  Execute subset of dataflow operations in data plane

- ***Idea 3: Iterative Refinement***

  Iteratively zoom-in on traffic of interests

Makes it easier to **express** and **scale** network telemetry tasks!

Idea 1: Uniform Prog. Abstraction

- ***Extensible Packet-tuple Abstraction***

  Queries operate over all packet tuples, at every location in the network

- ***Expressive Dataflow Operators***
  - Most telemetry applications require
    - collecting aggregate statistics over subset of traffic
    - joining results of one analysis with the other
  - Easy to express them as declarative queries composed of dataflow operators

Example Queries

Detecting Traffic Anomalies

Detect hosts for which # of unique source IPs sending DNS response messages exceeds threshold (Th)

```
pvictimIPs = pktStream(W)
               .filter(p => p.srcPort == 53)
               .map(p => (p.dstIP, p.srcIP))
               .distinct()
               .map((dstIP, srcIP) => (dstIP, 1))
```

Express queries without worrying about ***where*** and ***how*** they get executed

Example Queries

Confirming Reflection Attacks

Detect hosts with **traffic anomalies** that are of type RRSIG

```
victimIPs(t) = pktStream(W)
            .filter(p => p.srcPort == 53)
            .join(pVictimIPs(t), key='dstIP')
            .filter(p => p.dns.rr.type == RRSIG)
            .map(p => (p.dstIP, 1))
```

Join different packet tuple streams

```
            .map((dstIP, count) => dstIP)
```

Changing Status Quo

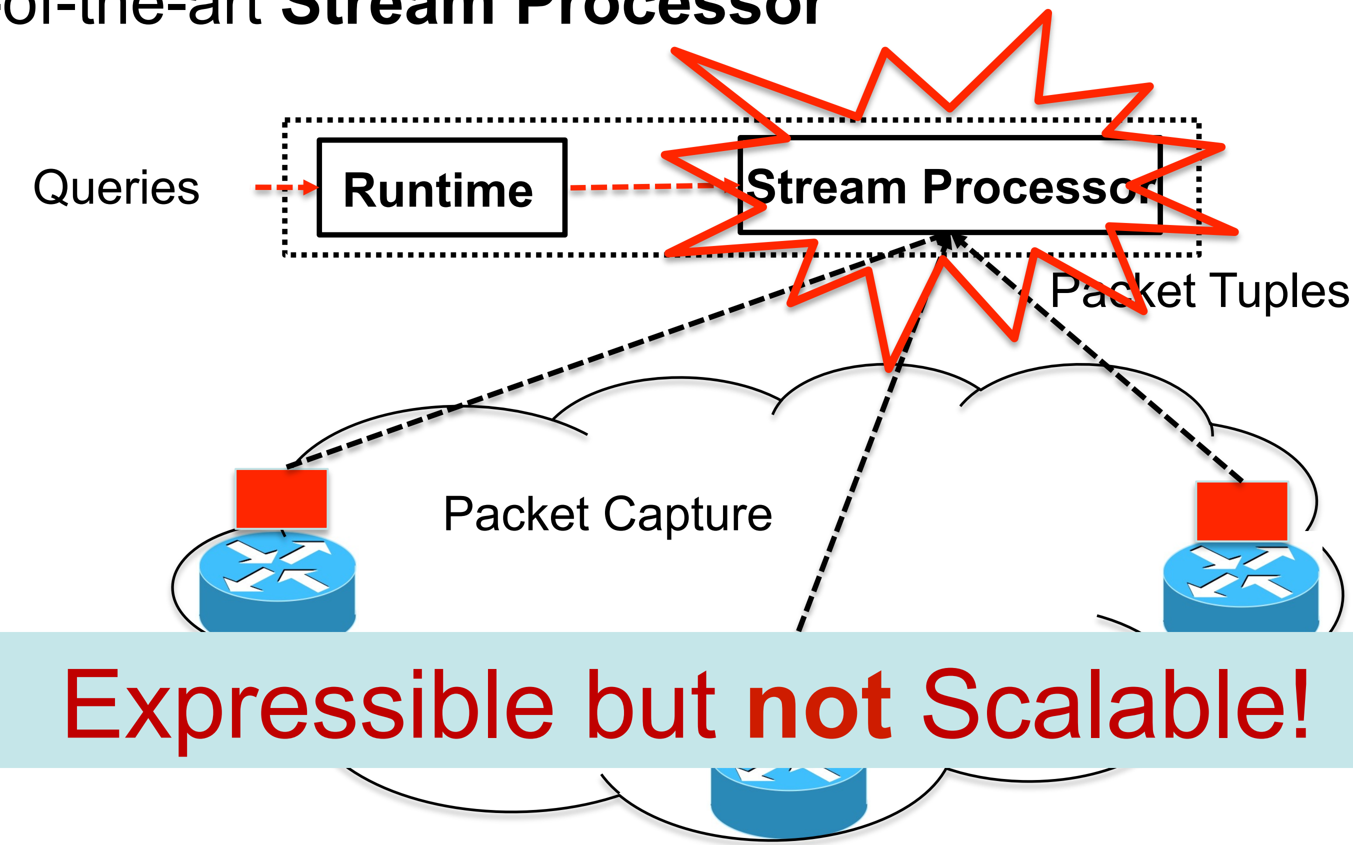
- ***Expressibility***
  - Express dataflow queries over packet tuples
  - Not tied to low-level (3rd party/platform-specific) APIs
  - Trivial to add new queries and change collection tools

Easier to express network telemetry tasks!

Query Execution

Use Scalable Stream Processors

Process all (or subset of) captured packet tuples using state-of-the-art **Stream Processor**



Expressible but **not** Scalable!

Scalable Query Execution

- ***Query Partitioning***
  - Execute subset of dataflow operators in data plane
  - Reduce packet tuples at the cost of additional state in the data plane
- ***Iterative Refinement***
  - Iteratively zoom-in on traffic of interests
  - Reduce state at the cost of additional detection delay

Idea 2: Query Partitioning

- ***Observation***

  Data Plane can process packets at line rate

- ***How it works?*** Dataflow operations in data plane,
  – filter, sample operations for OF-based data plane
  – map, reduce, filter, join, sample operations for PISA-based data plane

- ***Trade-off***

  Trades packet processing cost with additional state in the data plane

PISA Targets for Query Partitioning

- ***Programmable parsing***

  Allow new query-specific header fields for parsing

- ***State in packets & registers***

  Support simple stateful computations

- ***Customizable hash functions***

  Support hash functions over flexible set of fields

- ***Flexible match/action table pipelines***

  Support match/action tables with prog. actions

Compiling Dataflow Operators

- ***Map, Filter & Sample***

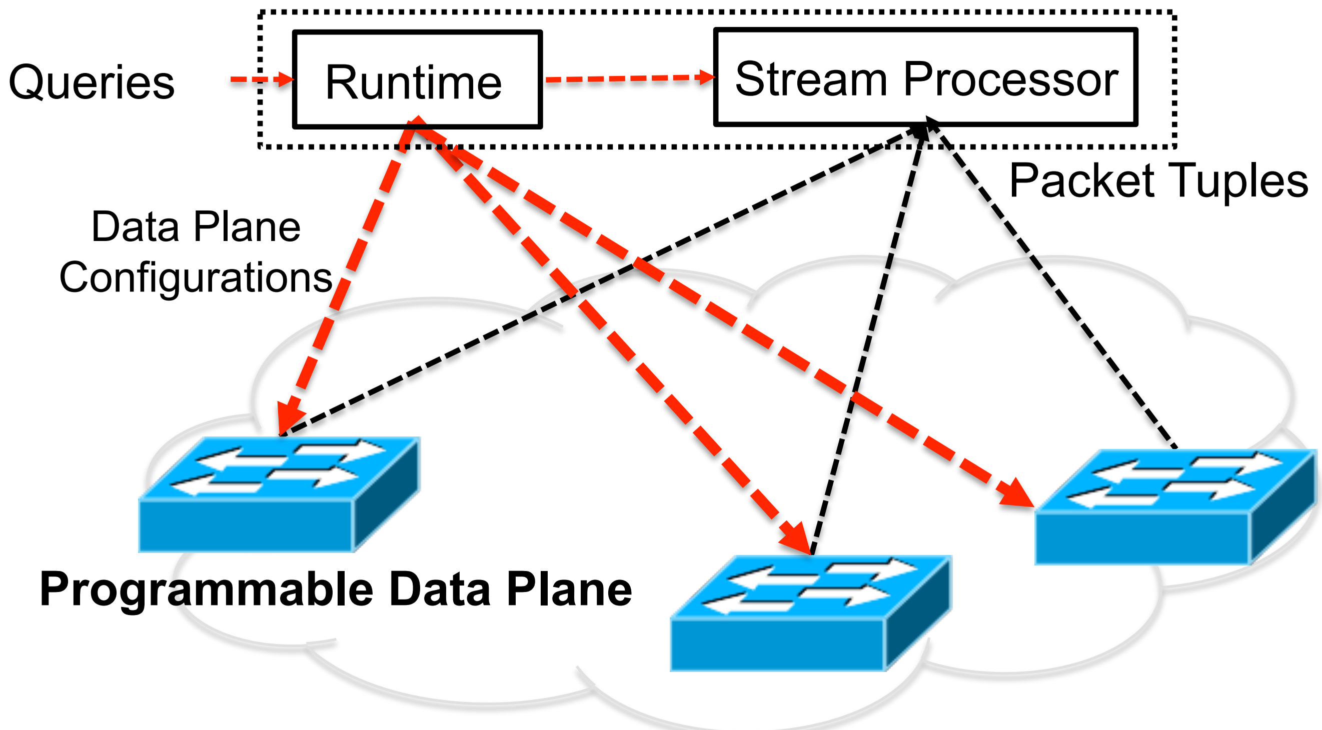
  Apply sequence of match-action tables

- ***Distinct & Reduce***
  - Compute index, & read value from hash tables
  - Apply function (e.g., bit_or for distinct) & then update the hash table
  - Use sketches, e.g. reduce(sum) → CM Sketches

- ***Limitations***

  Complex transformations, e.g. log, regex, etc.

Query Partitioning in Action



Runtime Partitions Input Queries

Idea 3: Iterative Refinement

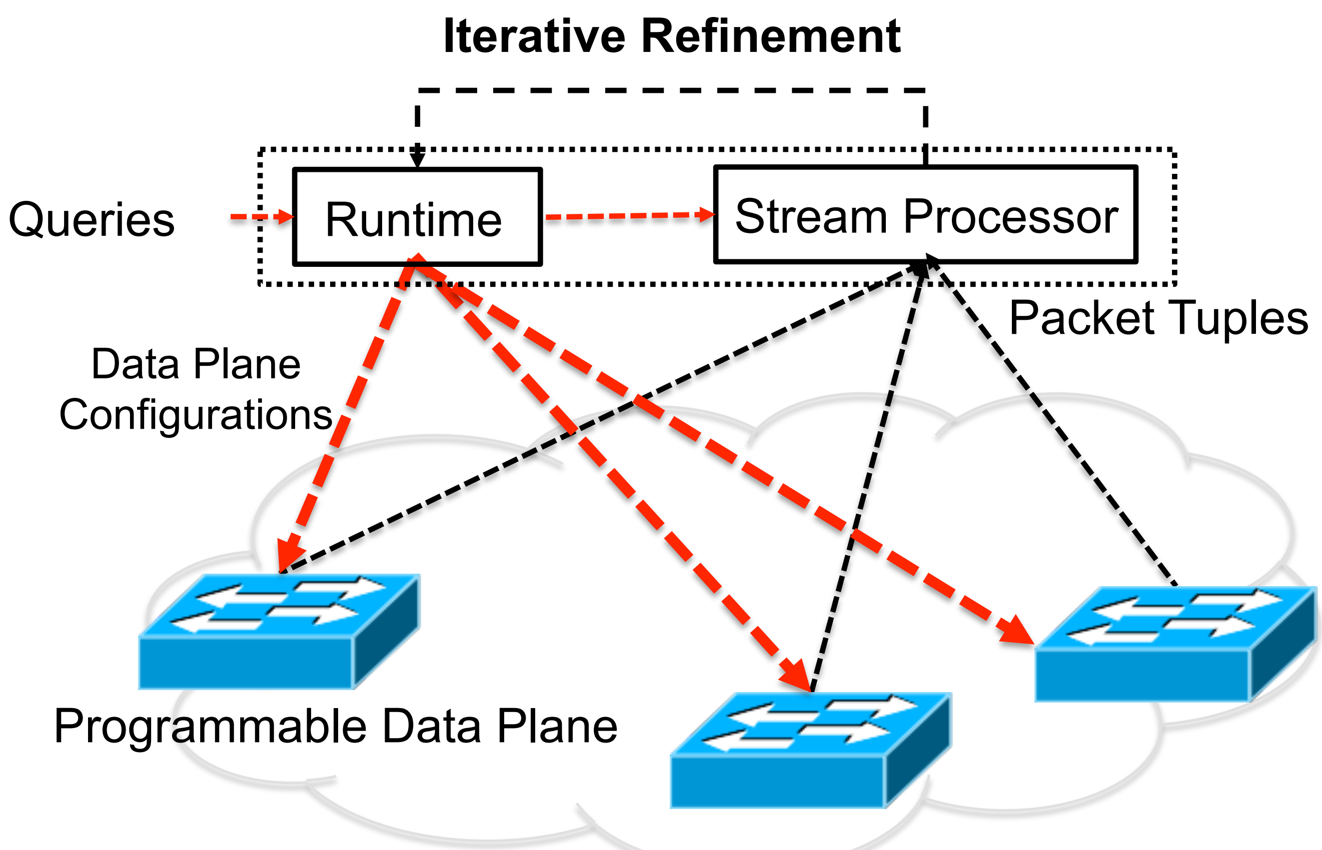
- ***Observation***

  Small fraction of traffic satisfies monitoring queries

- ***How it works***
  - Augment operators’ query to observe at coarser level
  - Iteratively (over successive window intervals) zoom-in to filter out uninteresting traffic

- ***Trade-offs***
  - Reduces packet processing & data plane state cost
  - Introduces additional detection delay cost

Iterative Refinement in Action



Collection is now driven by Analysis!

Scalable Query Execution

- ***Query Partitioning***
  – Execute subset of dataflow operators in data plane
  – Reduce packet tuples at the cost of additional state in the data plane

- ***Iterative Refinement***
  – Iteratively zoom-in on traffic of interests
  – Reduce state at the cost of additional detection delay

How to select the best query plan?

Query Planning

- ***Traffic Anomaly Query***

```
pktStream(W)
.filter(p => p.srcPort == 53)
.map(p => (p.dstIP, p.srcIP))
.distinct()
.map((dstIP, srcIP)=>(dstIP,1))
.reduceByKey(sum)
.filter((dstIP,count)=>count>Th)
.map((dstIP, count) => dstIP)
```

- ***Partitioning Plans***

  Plan 1: Data Plane only

  Plan 2: Stream Processor only

- ***Refinement Plans***
  - Refinement key: dstIP
  - Refinement levels: {/8, /32}

Query Planning

- ***Traffic Anomaly Query***

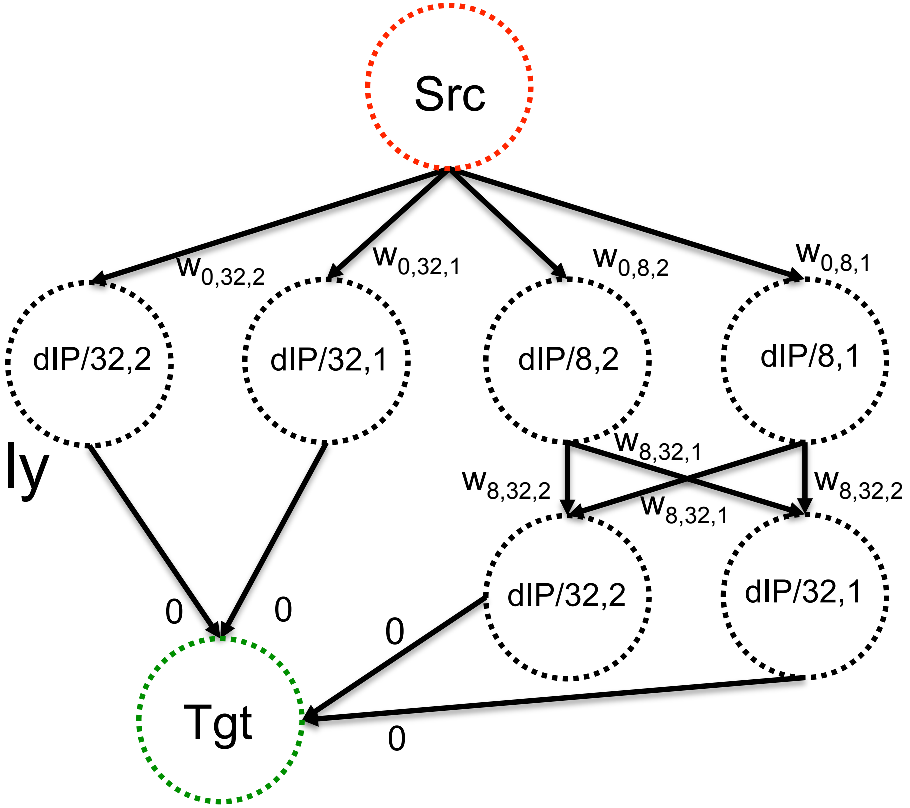
- ***Partitioning Plans***

  Plan 1: Data Plane only

  Plan 2: Stream Processor only

- ***Refinement Plans***
  – Refinement key: dstIP
  – Refinement levels: {/8, /32}

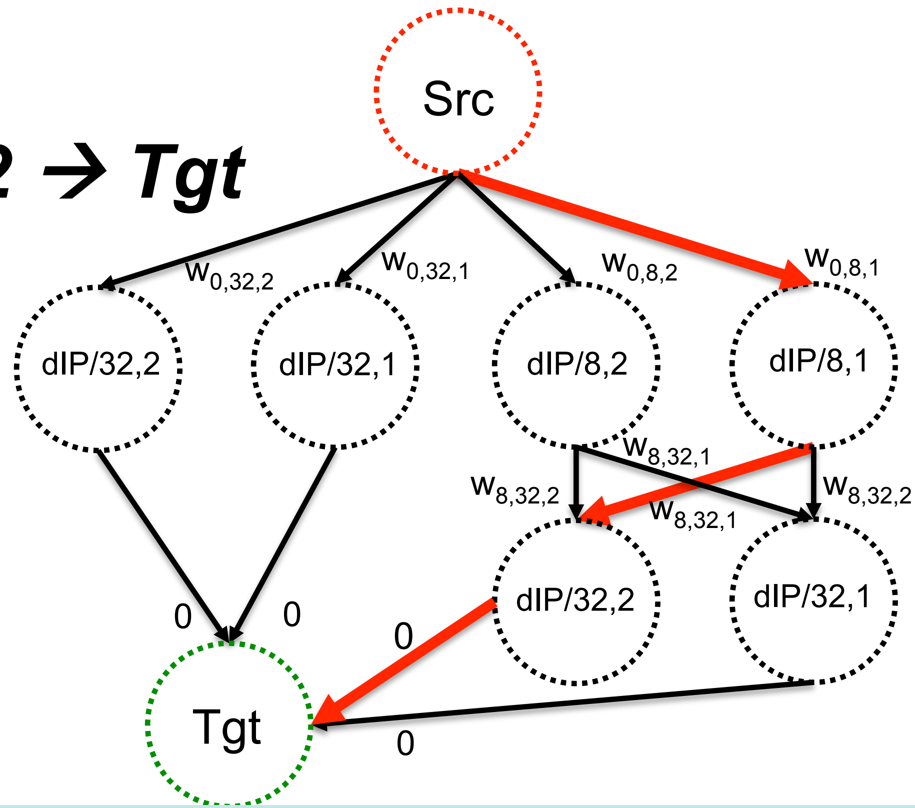
Query Plan Graph



Query Planning

Src → dIP/8,1 → dIP/32,2 → Tgt

Query Plan Graph



Selects plan with smallest weighted cost

SONATA
Query-Driven Network Telemetry

- ***Application Interface***

  Express queries w/o worrying about **where** and **how** they will be executed

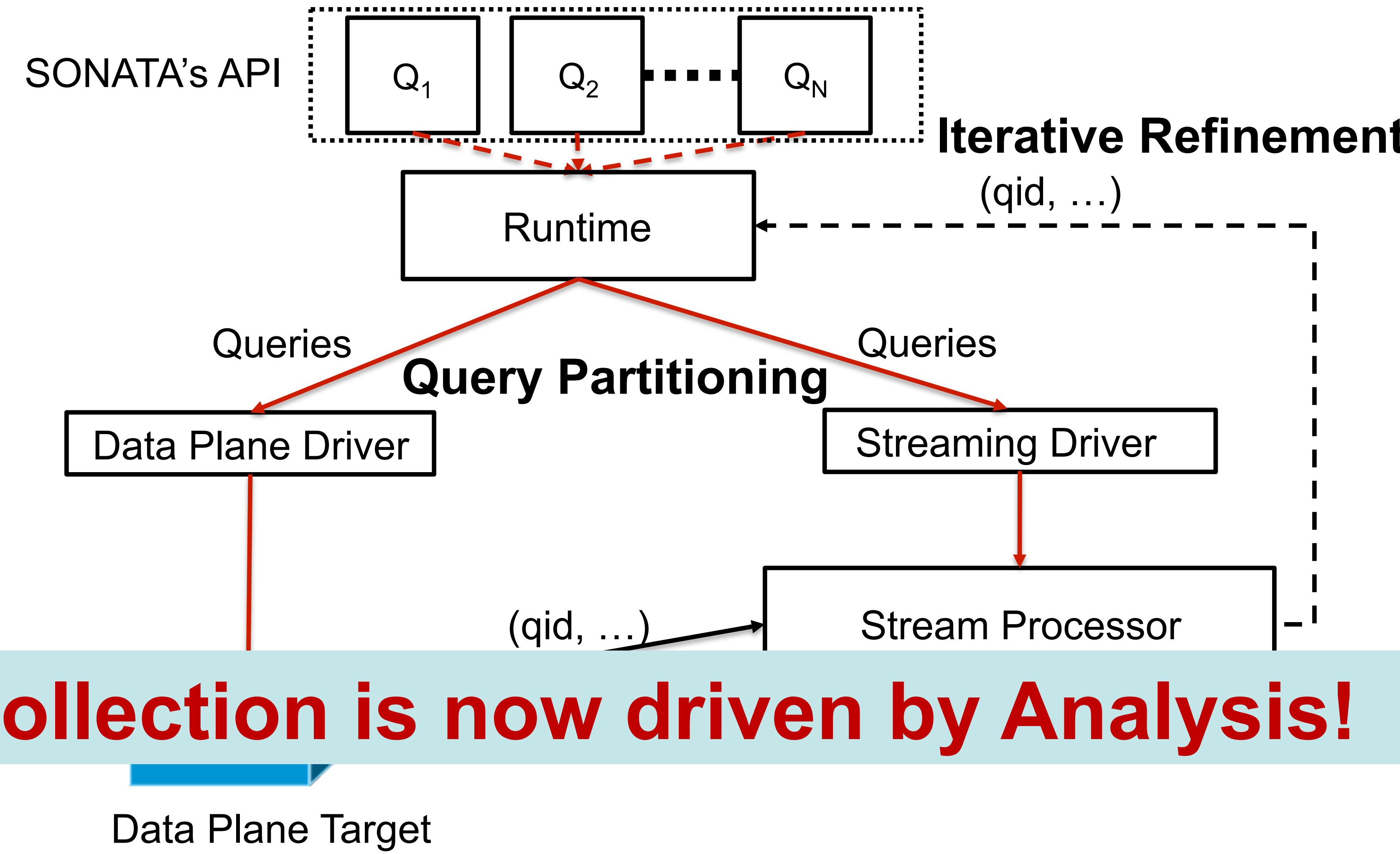
- ***Runtime System***

  Iteratively refines and partitions each input query

- ***Data Plane & Streaming Drivers***

  Compile input queries to target-specific configurations/ queries

Implementation



Collection is now driven by Analysis!

Evaluation

- ***Workload***

  Large-IXP network: 2 hours long IPFIX trace, 3 Tbps peak traffic, packet sampling rate = 1/10K

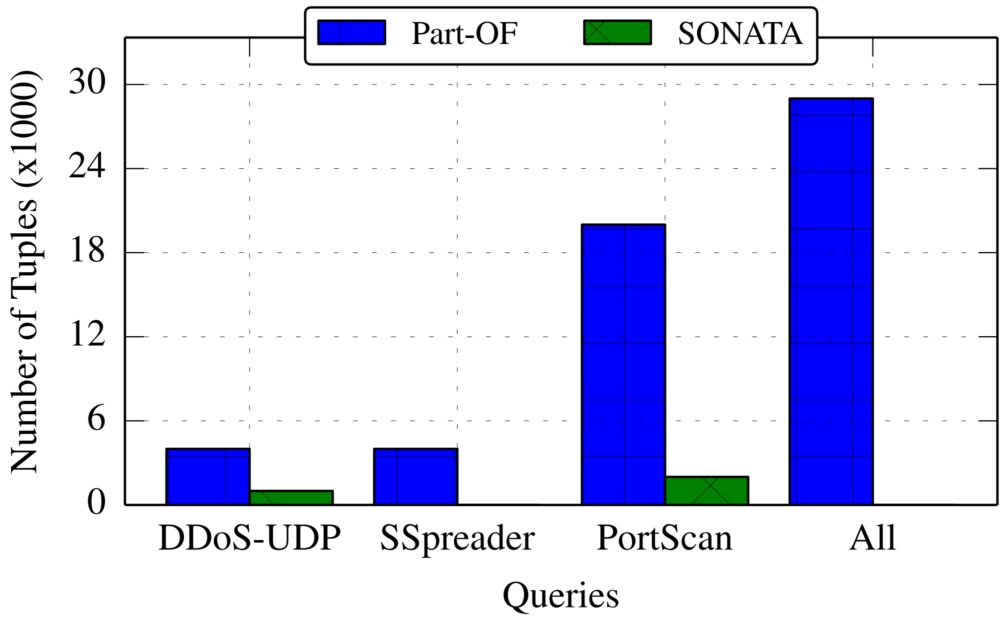
- ***Queries***

  DDoS-UDP, SSpreader, PortScan

- ***Comparisons***

  Part-OF, Part-PISA, Fixed-Refinement

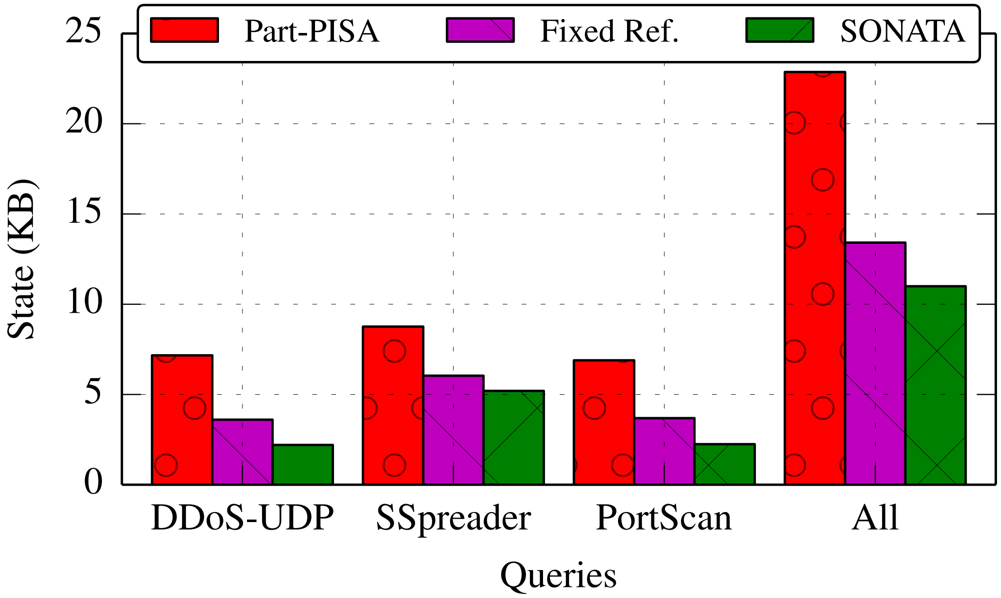
Benefits of Query Partitioning



Number of pkt tuples processed by Stream Processor

Executing stateful operations in data plane reduces workload on Stream Proc.

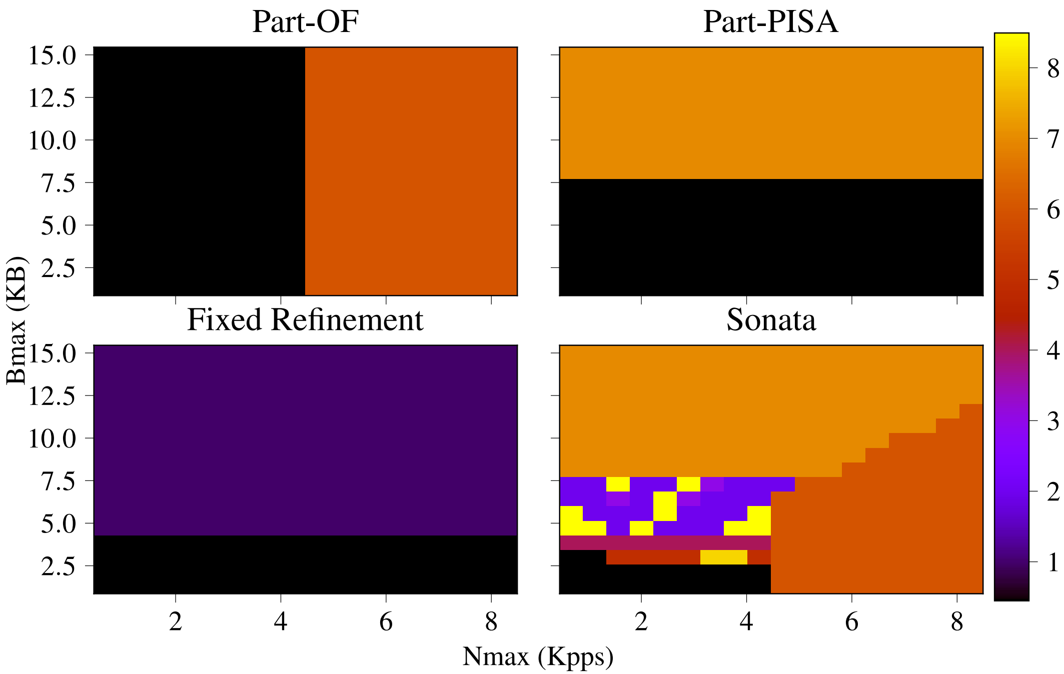
Benefits of Iterative refinement



State (KB) required by data plane targets

Iterative refinement reduces state required by the data plane targets

Benefits of Query Planning



- B_{max}: Max. state data plane can support
- N_{max}: Max. pkt. tuples stream processor can process
- Each color represents a unique query plan

SONATA makes best use of available resources

Changing Status Quo

- *Expressibility*
  - Express Dataflow queries over packet tuples
  - Not worry about how and where the query is executed
  - Adding new queries and collection tools is trivial

- *Scalability*
  - Answers hundreds of queries in real-time for traffic volume as high as few Tb/s

Expressible & Scalable!

    - tuples processed by the stream processor
    - state in the data plane

Summary

- SONATA makes it easier to **express** and **scale** network telemetry tasks using
  - Uniform Programming Abstraction
  - Query Partitioning
  - Iterative Refinement

- Running Code
  - Github: github.com/Sonata-Princeton/SONATA-DEV
  - Run test queries or express new ones

sonata.cs.princeton.edu