Traffic Monitoring as a Streaming Analytics Problem

Arpit Gupta[⋆], Rüdiger Birkner[†], Marco Canini[◇,*],
Nick Feamster[⋆], Chris Mac-Stoker[‡], Walter Willinger[‡]
[⋆]Princeton University [†]ETH Zürich [◇]KAUST [‡]NIKSUN, Inc.

Abstract

Programmable switches potentially make it easier to perform flexible network monitoring queries at line rate, and scalable stream processors make it possible to fuse data streams to answer more sophisticated queries about the network in real-time. However, processing such network monitoring queries at high traffic rates requires both the switches and the stream processors to filter the traffic iteratively and adaptively so as to extract only that traffic that is of interest to the query at hand. While the realization that network monitoring is a streaming analytics problem has been made earlier, our main contribution in this paper is the design and implementation of Sonata, a closed-loop system that enables network operators to perform streaming analytics for network monitoring applications at scale. To achieve this objective, Sonata allows operators to express a network monitoring query by considering each packet as a tuple. More importantly, Sonata allows them to partition the query across both the switches and the stream processor, and through iterative refinement, Sonata's runtime attempts to extract only the traffic that pertains to the query, thus ensuring that the stream processor can scale to satisfy a large number of queries for traffic at very high rates. We show with a simple example query involving DNS reflection attacks and traffic traces from one of the world's largest IXPs that Sonata can capture 95% of all traffic pertaining to the query, while reducing the overall data rate by a factor of about 400 and the number of required counters by four orders of magnitude.

1 Introduction

To ensure that the network is secure and performs well in the face of continually changing network conditions (*e.g.*, failures, attacks, shifts in traffic load), operators need to collect and fuse heterogeneous streams of information from traffic statistics to alerts from intrusion detection systems and other monitoring devices. Operators currently collect these data streams [25, 27, 29], which often arrive at high data rates; yet, despite the fact that these streams contain rich information about the security and performance of the network, operators have difficulty analyzing them [28].

Several factors make it difficult for network operators to analyze network traffic statistics to perform even basic analysis concerning the operation of the network. First, existing hardware offers relatively fixed-function measurement capabilities (*e.g.*, IPFIX [10], SNMP [3]), and enabling these functions is often a costly all-or-nothing decision. Second, because operators have no way of specifying what types of data they are interested in before it is collected, the configuration of these capabilities is often static (*e.g.*, a fixed sampling rate for IPFIX records), resulting in data that is either too high-volume to store or process, or too coarse to be particularly useful in answering questions of interest. Finally, existing approaches have no meaningful way to fuse these data streams, even though it is often the correlation of signals from multiple streams that can lend insight into higher-level performance or security problems.

Advances in both programmable switch hardware and streaming data analysis platforms make it possible to address these challenges. We explore whether improved switch programmability and better data stream processing capabilities improve the utility of network measurement. Programmable switches such as OpenFlow [17] switches make it possible to capture subsets of traffic by inserting rules in the switches that rely on simple match criteria in packet headers; software controllers update these rules and make it possible to update these rules in real-time, creating the potential for closed-loop feedback, where current observations can drive future decisions about which traffic to capture. In addition to better switch capabilities that allow for the collection of richer data streams, new system capabilities are making it easier to process and analyze network data: streaming data processing platforms such as Spark Streaming [22] and Apache Storm [23] make it possible to efficiently process queries on streams of tuples at relatively high data rates, and to issue queries based on combinations of heterogeneous data streams, facilitating complex queries that combine heterogeneous data sources, possibly from multiple distinct network vantage points.

[*]Work partly done when author was at Université catholique de Louvain.

HotNets-XV, November 09-10, 2016, Atlanta, GA, USA
 ISBN 978-1-4503-4661-0/16/11 $15.00
DOI: http://dx.doi.org/10.1145/3005745.3005748

Performing streaming data analysis on existing network traffic streams is more challenging than simply pointing existing streams from switches at off-the-shelf stream-processing systems. One challenge is the quantity of data: when we consider the volume of traffic traversing a backbone network or switch at a large Internet exchange point, it is clear that the volume of data is far too high for a typical stream processing system—although these systems are designed to "scale out" as data rates increase, traffic rates are high, already at several terabits per second, and increasing quickly [26]; and processing data at increasingly high rates raises both cost and complexity. Previous work such as OpenSoc [28] describes the complexity of processing millions of packets per second, which is still several orders of magnitude less than what exists in large backbone networks and Internet exchange points. Instead, we propose to use knowledge of the operator's query to refine the data that each switch collects, reducing the data that individual switches must export but nonetheless allowing for refinements of the measurements later in the process.

Consider the example of detecting a DNS reflection attack, whereby an attacker sends many DNS queries with a source IP address that is spoofed to be that of the victim. In this case, the operator might detect the attack by noticing a sudden increase in query volume or rate from a single source IP address, possibly for entirely new domains. Yet, even tracking this simple trend—the rate of DNS queries from individual source IP addresses—could in principle require creating a counter for each IP address, which is prohibitive, particularly given the increasing prevalence of IPv6. Instead, the operator might want to *express* a query that operates on smaller subsets of the total data and iteratively refines itself to "zoom in" on the attack traffic.

We introduce a system called Sonata[1] that allows an operator to express these types of queries using widely accepted programming idioms in distributed data analytics. Specifically, Sonata allows a network operator to *view each packet as a tuple* and express queries as operations over tuple streams, just as in any other data stream processing system. The Sonata runtime then both: (1) *partitions the workload* between the switches and the stream processing system to ensure that the stream processor does not become overloaded and (2) *iteratively refines* the configurations for the data plane and the stream processor, to allow operators to inspect traffic at finer granularities when anomalies or other interesting scenarios arise. Existing streaming analytics platforms for network monitoring (*e.g.*, [27, 29]) view the data stream as exogenously given. In contrast, Sonata considers the data to be endogenously determined; *i.e.*, it relies critically on a built-in feedback mechanism between the stream processor and the programmable data plane to adaptively refine the data stream itself, thus reducing the load on stream processors and enabling them to process queries for traffic streams at very high rates. It is in this sense that we view *network monitoring* as a new type of *streaming analytics* problem.

[1] Scalable Streaming Network Traffic Analysis (SSNTA, or "SONATA").

We evaluate our initial open-source prototype implementation in the context of DNS reflection attack detection and show that it is possible to "zoom-in" on traffic of interest while capturing far less traffic that does not pertain to the attack itself. We show with a simple example query involving DNS reflection attacks that Sonata can capture 95% of all traffic pertaining to the query, while reducing the overall data rate by a factor of about 400 and the number of required counters by four orders of magnitude.

2 Related Work

One extreme of the spectrum of design options for Sonata is to execute monitoring queries entirely in user space. The data processing component of Sonata follows a long line of related efforts in the database community [1, 8, 12, 32, 36] and also builds on the prior work on streaming data in the form of network traffic [4, 12, 32]. Chimera [4] introduced a new query language based on a streaming SQL for processing network traffic in user-space. Network operators have leveraged the recent advances in the area of scalable streaming data analysis [22–24, 27, 29] to build platforms capable of processing network data at very high rates. The database community has also explored the query optimization problem extensively [2, 21]. Gigascope [12] uses query partitioning to minimize the data transfer within the stream processor. Geo-distributed analytics systems such as Clarinet [34] use forms of query partitioning. Yet, executing all transformations in the user space is costly. As a result, these platforms face major scalability challenges at high data rates [28].

The other extreme of the design spectrum for Sonata is to execute the monitoring queries entirely in the data plane. Executing monitoring queries in the data plane is not new. Before the days of programmable data planes, vertically integrated monitoring programs with limited (and fixed) functionalities like NetFlow [9], SFlow [20], IPFIX [10], and SNMP [3] could execute simple monitoring queries. The advent of programmable data plane broadened the scope of monitoring queries that can be executed in the data plane.

OpenSketch [35] equips switches with a library of predefined functions(*e.g.*, count-min sketch, reversible sketch) in hardware; the controller selects and assembles them for different measurement tasks. UnivMon [16] takes a "RISC-type" approach to measurement, replacing the entire library of predefined functions with a generic monitoring primitive on the routers in the form of a single universal sketch. Similarly, Narayana et al. [19] recently proposed the design of a switch supporting a range of network performance queries that execute on the switch using a programmable key-value store. The queries enabled by the programmable data plane are not suited for applications that (1) require joining multiple data streams; (2) require executing more complex operations such as skyline monitoring [33], or frequent, rare, or persistent itemset mining [7, 13]; and (3) require processing packet payloads, *e.g.*, monitoring applications described in Chimera [4].

ProgME [36] and Jose et al. [14] also explore the idea of iterative refinement for detecting heavy hitter traffic. They

use iterative refinement to minimize the number of counters required to identify hierarchical heavy hitters, but ProgME requires multiple passes over the same packets—making scalability to high data rates very challenging. Unlike Sonata, these systems concentrate on executing queries entirely in the data plane. We posit that the combined use of a general-purpose stream processor and the programmable data plane gives network operators the "best bang for the buck"—the flexibility of stream processing and the speed of the data plane.

3 Example Applications

In this section, we show how three network monitoring problems—reflection attack monitoring, application performance analysis, and port scan detection—can be expressed as streaming analytics problems.

Reflection attack monitoring. Consider the problem of detecting DNS amplification attacks, where compromised machines send spoofed DNS requests to resolvers. These spoofed requests have source IP addresses inside the target network. One such reflection attack on Spamhaus in 2013 [6] used some 30,000 open resolvers around the globe and an amplification factor of about 70 to generate attack traffic with an intensity of around 75 Gbps.

A straightforward approach to detect DNS-based amplification attacks in real-time requires maintaining state for every unique IP address and keeping track of the difference between the observed DNS requests and responses for each IP address; if that difference exceeds a pre-specified threshold, it may indicate the onset of an attack. At an Internet exchange point (IXP), every traffic flow that traverses the IXP switching fabric can be mapped to a source and destination MAC address corresponding to where the traffic enters and leaves the IXP switch. Traffic volumes at such a location are so high [26] that they would overwhelm any reasonably provisioned stream processor [28]; yet, because the traffic of interest is only a small fraction of DNS traffic (which is, in turn, only a small fraction of all traffic), the stream processor can take advantage of data-plane programmability to iteratively push rules into the data plane that only return traffic that satisfies the query.

Real-time application performance analysis. Assume a network with asymmetric network paths, such that data and acknowledgments traverse different paths in the network. Suppose a network operator wishes to construct a distribution of round-trip latency (or other statistics, such as jitter or packet loss) for all video streaming flows. Each network location sees a stream of packets. A stream of packets can be represented as a stream of tuples, having attributes such as timestamp, src IP, src port, dst IP, dst port, and application type. One location in the network will have the tuples corresponding to data traffic, and another may see tuples corresponding to the ACKs.

To create a stream of tuples that includes round-trip times, we must join the tuples at each of the two locations. A filter operation can select for streaming video traffic, and a reduce operation can perform the necessary subtraction and aggregation to compute the round-trip latency over time. Note that the operator can express the query simply as filtering and reduce operations that view the network traffic as a single large collection of tuples, even though traffic may be distributed across multiple locations. Queries might also aggregate these statistics at coarser levels of aggregation (*e.g.*, AS, prefix, user group), iteratively zooming-in on user groups for which the measured round-trip time exceeds a given threshold.

Distributed port scan detection. Suppose an operator wants to detect port scans that may be coming from distributed locations (and, hence, appear at a variety of network locations). Existing intrusion prevention system (IPS) devices often cannot process traffic at high rates, and they typically only operate at a single network location. Instead, a network operator might write a query that counts the number of distinct SYN packets that never have a corresponding ACK packet, as in previous port scan detection work [15]. By viewing each packet as a tuple, writing such a query is straightforward: a simple reduce operation can couple each SYN with a matching ACK, if it exists. Such a query must necessarily be distributed across the network, since SYNs and their corresponding ACKs may not traverse the same network devices.

4 Sonata

Sonata allows network operators to specify monitoring queries and fuse data streams from multiple queries. Sonata has a runtime system that compiles queries to generate a set of rules to install in the switches and processing pipelines at the stream processor. Figure 1 shows how Sonata processes incoming network traffic to extract tuples that satisfy a particular query. In this section, we provide an overview of Sonata and the design insights that allow it to scale to high data rates. Sonata's runtime translates each query into the forwarding table entries for the data plane and data processing pipelines for the stream processor. The data-plane operations ensure that (1) filtering is based on relative sampling rates for different flows, and (2) the rate of the filtered data stream is always less than the system-defined constraints (*e.g.*, span port capacity (P), supported ingestion rate (R) for the streaming platform). We implemented the current prototype in Python; the Ryu [31] controller interacts with software switches running Open vSwitch 2.5 and OpenFlow 1.3. The stream processor is Apache Spark [22].

The rest of the section explains Sonata's design choices in detail, using monitoring of DNS reflection attacks at a large IXP from Section 3 as a running example: Simple detection of DNS reflection might count DNS request and response messages for each IP address at the IXP and compare the obtained values against a threshold at regular interval to detect victim IP addresses. Although we focus on this particular example, other possible applications could include reflection attack monitoring for other UDP protocols [30], detection of distributed port scans, or monitoring TCP traffic across

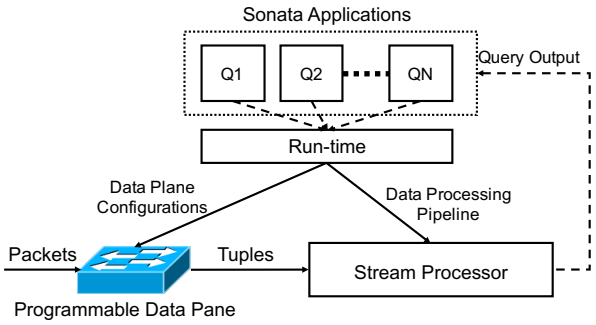

Figure 1: *Sonata architecture for detecting network events in reactive manner.*

asymmetric paths to track the jitter of a video stream over time as discussed in Section 3.

4.1 Packets-as-Tuples Abstraction

Sonata presents network operators with the simple abstraction of packets as a tuple, thus allowing them to write network monitoring queries in terms of operations over a stream of tuples, which is a common model for stream processing frameworks like Apache Storm or Spark Streaming. Our API adopts and extends the functional API of Spark Streaming, familiar to many programmers. Each packet header is a tuple; the payload itself is also represented as a tuple. Thus, each packet tuple is a collection of field values including (`ts`, `locationID`, `sIP`, `sMac`, `sPort`, `dIP`, `dMac`, `dPort`, `bytes`, `payload`), where `locationID` represents the location of the packet in the network (*i.e.*, which switch it is traversing). The example below shows how a network operator might take a raw stream of packets, filter it according to some criterion (*e.g.*, DNS replies), sample the resulting tuple stream at a given rate r, and count the resulting number of tuples within each time interval of length T. The argument to the filter operation is a function literal (or lambda).

```
1  DNS = pktStream.filter(p => p.sPort == 53)
2          .sample(r).countByWindow(T)
```

4.2 Query Partitioning

Sonata allows a programmer to specify whether a particular operation should execute in the data plane (operations denoted with the suffix `D`) or at the stream processor (used by default). For instance, the `filterD` operation applies filtering at the switch, whereas the `filter` operation applies it at the stream processor; a similar distinction applies for the `sampleD` / `sample` operations. Currently, programmers must specify this partitioning manually, but we are exploring algorithms to automate this process. As an example, consider monitoring destination IPs (*dIPs*) for which the number of DNS replies received over a time interval T exceeds a given threshold X. A programmer could specify that the switch should perform the initial filtering and sampling of the raw packet stream, reducing the workload on the stream processor:

```
1  IPs = pktStream
2        .filterD(p => p.sPort == 53)
3        .sampleD(r).map(p => p.dIP)
4        .countByValueAndWindow(T)
5        .filter(t => t.count > X)
```

4.3 Iterative Query Refinement

Sonata allows network operators to express the logic for refining queries based on dynamic conditions. We refer to this process as iterative query refinement. Operators can use domain-specific insights to express their logic for refining queries.

As Sonata enables combining multiple queries that fuse data streams, an interesting form of iterative query refinement occurs when the results from ongoing queries are used to alter existing queries. For example, consider an application with two monitoring queries q_1 and q_2, each executing over a time interval of length T. Assume q_1 is parameterized by some argument A, *i.e.*, $q_1(A)$ and that a new value of A is produced by q_2 after every time interval, *i.e.*, $A(t+1) = q_2^{(t)}$. Then, we observe that $q_2^{(t)}$ refines q_1 at time interval $t+1$: $q_1^{(t+1)}(A(t+1)) = q_1^{(t+1)}(q_2^{(t)})$, since q_1 is affected by the execution of q_2 in the previous time interval.

For instance, our examples in the previous sections uniformly sample all the DNS response traffic; but at very high data rates, this approach may be prohibitive with a high sampling rate. Instead, one could sample the entire DNS response traffic at a lower sampling rate, and at a higher sampling rate only the traffic from "suspicious" IP addresses. The example below shows how an operator would specify this objective using iterative query refinement:

```
1  pvicIPs(t) =
2    pktStream.filterD(p => p.sPort == 53)
3    .sampleD(r0).map(p => (p.dIP,p.sIP))
4    .distinct.map(t => (t.dIP,1))
5    .countByValueAndWindow(T)
6    .filter(t => t.count > X)
7
8  cvicIPs(t) =
9    pktStream.filterD(p => p.sPort == 53)
10   .filterD(p => p.dIP in pvicIPs(t-1))
11   .sampleD(r1).map(p => (p.dIP,p.sIP))
12   .distinct.map(t => (t.dIP,1))
13   .countByValueAndWindow(T)
14   .filter(t => t.count > X')
```

This example identifies confirmed victim IP addresses (*i.e.*, `cvicIPs`) by combining two queries executed over successive time windows of length T. The first query picks up the list of potential victim IPs (*i.e.*, `pvicIPs`); that is, those *dIPs* that receive DNS replies from more than X unique source IP addresses. At the end of each time interval, the most current `pvicIPs` list refines the second query in the sense that it serves as input to the second query which samples traffic from these potential victim IPs at a higher rate ($r1 > r0$) during the next time interval so as to confirm the presence of attack traffic using pre-specified threshold values. Figure 2 shows how query partitioning and iterative refinement are realized in

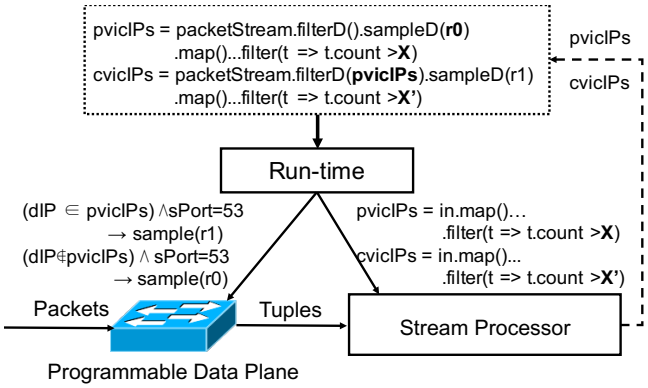

Figure 2: *Sonata's query partitioning and iterative refinement in action.*

Sonata for this example. Note that iterative query refinement only updates the filtering and sampling configuration in the data plane; this particular example involves no updates to the stream processor.

However, iterative queries are not limited to updating only the data plane's filtering and sampling configuration. For instance, one can write iterative queries that process the packet stream differently—making the confirmation process more robust. Notice that the example above still requires maintaining per-IP counters. However, one can instead count at a coarser level of granularity and iteratively refine the queries that count at a finer level. For the case of detecting DNS attacks at an IXP, one can for instance write a query that maintains per-MAC (*i.e.*, IXP's physical ports) counters, compares their values against a given threshold, and populates `pvicMACs`, a list of MAC addresses suspected to receive attack traffic. Note that while per-IP counters are typically in the order of millions for a large IXP, the number of per-MAC counters are just a few hundreds. A query for the next interval can then use this query's output for the current interval to produce the `pvicIPs` list—processing packet tuples for victim MAC addresses only. And finally, over a subsequent time interval, another query can take the list of pvicIPs as input to confirm victim IPs as shown above.

5 Preliminary Evaluation

We perform a trace-driven simulation to evaluate the effectiveness of iterative query refinement and query partitioning. For this simulation, we first use our prototype implementation to express the queries for the DNS-based reflection attack monitoring application. We then use a trace of IPFIX records, collected at a large IXP to measure how these two features help Sonata scale to high data rates. In this preliminary evaluation, we show that together Sonata's query partitioning and iterative refinement features reduce both the traffic rates that the stream processor sees and the total number of counters required for monitoring. Because the attack traffic is typically a small fraction of the total traffic, Sonata's iterative query refinement can dynamically and reactively filter non-attack traffic. The application also benefits from query partitioning,

	Rate (kpps)	*# Counters*	*% of Traffic*
No Filtering	210,794	2.08 B	100%
Simple Filtering	2,006	9.91 M	100%
Sonata			
- No Refinement	500	2.82 M	19%
- DP Refinement	500	1.68 M	88%
- DP & SP Refinement	500	200 K	95%

Table 1: *Sonata's iterative query refinement and partitioning helps efficiently capture traffic for the query.*

which performs certain filtering and sampling tasks in the data plane.

Experiment setup. We use a trace of IPFIX records, collected using a packet sampling rate of 1 in 10,000 from one of the largest IXPs in Europe. On average, this IXP handles about 3 Tbps of traffic and is thus a good use case for the type of traffic rates Sonata is designed to handle. We use a two-hour traffic trace collected from this IXP in August 2015. Our data set does not contain any user data or any personal information that identifies individual users. The data was collected between 2-4 a.m. local time (GMT+2) on a working day in August 2015. Since the collection took place during the non-peak hours, we only observed 128 million flow records in the data. By manually analyzing the trace, we identified the portions of traffic that satisfy the different reflection attack monitoring queries.

To illustrate the benefits of each of Sonata's features, we evaluate the system using five different configurations: (1) No Filtering; (2) Simple Filtering: A filter that sends all DNS traffic to the stream processor, without sampling or query refinement; (3) No Refinement: Partitioning the query across the data plane and stream processor, without performing iterative refinement; (4) DP Refinement: Updating the query expressions dynamically to modify the data-plane configuration over two successive time intervals, as described in Section 4.3; (5) DP & SP Refinement: Modifying both the data plane and the stream processor with iterative query refinement, as described in Section 4.3.

Reducing data rates. We examine whether executing portions of a monitoring query in the programmable data plane reduces the resulting data rate at the stream processor. Table 1 shows the performance of the five modes in terms of the following metrics: (1) the median rate of packet tuples forwarded to the stream processor; (2) the median number of counters required to track the query at the stream processor; and (3) the median fraction of query-related packet tuples forwarded to the stream processor.

Without filtering, the stream processor must process about 210 million packet tuples per second. Applying the filter to only consider DNS traffic reduces rate to about two million tuples per second. Because Sonata's stream processor is configured to accept a fixed maximum data rate, we impose a fixed limit of 500,000 tuples per second and explore how much of the traffic that satisfies the given query is captured by the different versions of query refinements. We observe

that the combination of different iterative refinement modes allows Sonata to capture 95% of all traffic pertaining to the query, while reducing the overall data rate by a factor of about 400.

Reducing counters. Table 1 shows the median number of counters required for every ten-second time interval for all the five modes of our example application. Simpler filtering reduces the number of counters required to detect attack traffic from about 2 billion counters to just under 10 million counters. Sonata's query partitioning reduces the number of counters further, to just under 3 million counters. Performing iterative refinement in the data plane reduces the number of counters to 1.68 million, and performing iterative refinement in both the data plane and at the stream processor (*i.e.*, to refine the granularity of the query in real time) reduces the number of counters by more than a factor of ten compared to performing no query refinement at all. Additionally, query refinement enables the data plane to return a more accurate query stream, given a fixed rate of 500,000 tuples per second. Given this constraint, without iterative query refinement, the data plane returns only 19% of the tuples that satisfies the query to the stream processor; with iterative query refinement, 95 % of the tuples that satisfy the original query are returned.

6 Future Directions

In contrast to existing programmable data planes which are relatively fixed-function (*e.g.*, OpenFlow chipsets), emerging technologies, such as those that enable in-band network telemetry via P4 [5], make it possible to redefine packet-processing control-flow at compile time. This capability may enable a variety of richer measurement applications that could take advantage of a programmable, stateful network data plane.

One example of such an application could be to use in-band network telemetry to attach latency statistics to packets as they travel through network devices, thus making it possible to pinpoint sources of increased latency, packet loss, or congestion. The network devices could affix *additional* data to packet headers (*e.g.*, latency at each hop, the set of switches that the packet traversed) which could subsequently be used as input to a tuple-based query. Another such example is the use of so-called "active" machine learning algorithms that improve their accuracy over time by requesting more examples of labeled data. These algorithms could potentially use iterative refinement to define a query that asks for more examples of attack payloads when the algorithm needs to improve its accuracy.

Moving forward, streaming data—and the corresponding aggregate statistics that the queries produce—could be used to drive real-time control decisions. In particular programmable data plane, driven by a Sonata controller, could produce fine-grained measurements as an input to inference algorithms which could then drive the installation not only of forwarding table rules to refine the measurements, but also of forwarding table rules that affect how traffic is forwarded.

Another interesting research direction concerns how these systems could support approximate queries (*e.g.*, [11, 18]). The current prototype and all of the examples that we have presented thus far return tuple streams or statistics that are based on *exact* filter operations. In practice, however, many network monitoring queries need not be so precise. An attack or a performance degradation may be evident from a large deviation from baseline statistics under normal operation; in these cases, even an approximate result could reveal the existence of a problem. Support for approximate queries that can operate on samples of network traffic is another interesting avenue for future work.

7 Conclusion

Network operators must typically perform network management tasks while coping with fixed-function network monitoring capabilities, such as IPFIX and SNMP. The advent of programmable hardware makes it possible not only to customize packet formats and protocols, but also to install custom monitoring capabilities in network devices that output data in formats that are amenable to the emerging body of scalable, distributed stream processing systems.

In light of these trends, we have argued that it may be possible to think of network monitoring as a stream processing problem, where each packet is represented by a tuple, and streams of packets comprise tuple streams for which many distributed stream processing programming idioms can apply. Due to the inherently high rates of network traffic, realizing this programming abstraction requires reducing the traffic at the stream processor that does not satisfy the original query. Our prototype of Sonata shows that (1) partitioning of function between the switch and the stream processor; and (2) the ability to iteratively refine both the data plane rules for a query and its corresponding stream processing pipeline can reduce data rates at the stream processor by multiple orders of magnitude by pushing many of the filtering operations into the data plane.

Much work clearly lies ahead—such as improving the accuracy of iterative refinement; automating various aspects of query partitioning and iterative refinement, rather than relying on the programmer to specify these parameters; taking more advantage of the increasing programmability in P4-capable data planes; and supporting approximate queries. Yet, the time is right to start thinking about how to apply streaming analytics frameworks to network monitoring. Doing so can ultimately help operators move from the current crippling set of technologies towards defining monitoring problems in terms of the questions they want to answer and the data they need to answer them.

Acknowledgments. We thank our shepherd Fadel Adib, Jennifer Rexford, Rick Porter, Ankita Pawar, Srinivas Narayana and the anonymous reviewers for the feedback and comments. This research was supported by National Science Foundation Awards CNS-1539920, and by European Union's Horizon 2020 program under the ENDEAVOUR project (grant agreement 644960).

References

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, et al. Aurora: A Data Stream Management System. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 666–666. ACM, 2003.

[2] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[3] D. Black, K. McCloghrie, and J. Schoenwaelder. Uniform Resource Identifier (URI) Scheme for the Simple Network Management Protocol (SNMP). RFC 4088 (Proposed Standard), June 2005.

[4] K. Borders, J. Springer, and M. Burnside. Chimera: A Declarative Language for Streaming Network Traffic Analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, pages 365–379. USENIX, 2012.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[6] P. Bright. Spamhaus DDoS grows to Internet-threatening Size. *ArsTechnica*, March 2013.

[7] T. Calders, N. Dexters, J. J. Gillis, and B. Goethals. Mining frequent itemsets in a stream. *Information Systems*, 39:233–255, 2014.

[8] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *VLDB*, pages 215–226, 2002.

[9] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004.

[10] B. Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101 (Proposed Standard), January 2008.

[11] G. Cormode and M. Garofalakis. Approximate Continuous Querying Over Distributed Streams. *ACM Transactions on Database Systems (TODS)*, 33(2):9, June 2008.

[12] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 647–651. ACM, 2003.

[13] D. Huang, Y. S. Koh, and G. Dobbie. Rare Pattern Mining on Data Streams. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 303–314. Springer, 2012.

[14] L. Jose, M. Yu, and J. Rexford. Online measurement of large traffic aggregates on commodity switches. In *Proceedings of Hot-ICE'11*. USENIX, 2011.

[15] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast Portscan Detection using Sequential Hypothesis Testing. In *IEEE Symposium on Security and Privacy*, pages 211–225, 2004.

[16] Z. Liu, G. Vorsanger, V. Braverman, and V. Sekar. Enabling a RISC Approach for Software-Defined Monitoring using Universal Streaming. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 21. ACM, 2015.

[17] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 2008.

[18] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Conference on Innovative Data Systems Research (CIDR)*, January 2003.

[19] S. Narayana, A. Sivaraman, V. Nathan, M. Alizadeh, D. Walker, J. Rexford, V. Jeyakumar, and C. Kim. Co-designing software and hardware for declarative network performance management. In *HotNets*, 2016. To appear.

[20] P. Phaal, S. Panchen, and N. McKee. InMon corporation's sFlow. *RFC3176 (September 2001)*, 2001.

[21] O. Polychroniou, R. Sen, and K. A. Ross. Track Join: Distributed Joins with Minimal Network Traffic. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1483–1494. ACM, 2014.

[22] Apache Spark. `http://spark.apache.org/`.

[23] Apache Storm. `http://storm.apache.org/`.

[24] Cloud DataFlow. `https://cloud.google.com/dataflow/`.

[25] Deepfield Defender. `http://deepfield.com/products/deepfield-defender/`.

[26] Five Year Traffic Growth at DE-CIX. `https://www.de-cix.net/about/statistics/`.

[27] OpenSOC. `http://opensoc.github.io/`.

[28] OpenSOC Scalability. `https://goo.gl/CX2jWr`.

[29] Tigon. `http://tigon.io/`.

[30] UDP-Based Distributed Reflective Denial of Service Attacks. `https://www.us-cert.gov/ncas/alerts/TA14-017A`.

[31] Ryu SDN Framework. `http://osrg.github.io/ryu/`.

[32] M. Sullivan. Tribeca: A Stream Database Manager for Network Traffic Analysis. In *VLDB*, volume 96, page 594, 1996.

[33] S. Sun, Z. Huang, H. Zhong, D. Dai, H. Liu, and J. Li. Efficient Monitoring of Skyline Queries over Distributed Data Streams. *Knowledge and information systems*, 25(3):575–606, 2010.

[34] R. Viswanathan, G. Ananthanarayanan, and A. Akella. Clarinet: WAN-Aware Optimization for Analytics Queries. In *OSDI*, 2016. To appear.

[35] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 29–42, 2013.

[36] L. Yuan, C.-N. Chuah, and P. Mohapatra. Progme: Towards programmable network measurement. *SIGCOMM Comput. Commun. Rev.*, 37(4):97–108, August 2007.