

# Streams, Security and Scalability

Theodore Johnson<sup>1</sup>, S. Muthukrishnan<sup>2</sup>, Oliver Spatscheck<sup>1</sup>, and Divesh Srivastava<sup>1</sup>

<sup>1</sup> AT&T Labs–Research

{johnsont, spatsch, divesh}@research.att.com

<sup>2</sup> Rutgers University

muthu@cs.rutgers.edu

**Abstract.** Network-based attacks, such as DDoS attacks and worms, are threatening the continued utility of the Internet. As the variety and the sophistication of attacks grow, early detection of potential attacks will become crucial in mitigating their impact. We argue that the Gigascope data stream management system has both the functionality and the performance to serve as the foundation for the next generation of network intrusion detection systems.

## 1 Introduction

The phenomenal success of the Internet has revolutionized our society, providing us, e.g., the ability to communicate easily with people around the world, and to access and provide a large variety of information-based services. But this success has also enabled hostile agents to use the Internet in many malicious ways (see, e.g., [10, 9, 36]), and terms like spam, phishing, viruses, worms, DDoS attacks, etc., are now part of the popular lexicon. As network-based attacks increase, the continued utility of the Internet, and of our information infrastructure, critically depends on our ability to rapidly identify these attacks and mitigate their adverse impact.

A variety of tools are now available to help us identify and thwart these attacks, including anti-virus software, firewalls, and network intrusion detection systems (NIDS). Given the difficulty in ensuring that all hosts run the latest version of software, and the limitations of firewalls (e.g., worms have been known to tunnel through firewalls), NIDS are becoming increasingly popular among large enterprises and ISPs. Network intrusion detection systems essentially monitor the traffic entering and/or leaving a protected network, and look for signatures of known types of attacks. In practice, different NIDS use different mechanisms for the flexible specification of attack signatures. Snort [34], e.g., uses open source rules to help detect various attacks (such as port scans) and alert users. Bro [32], e.g., permits a site's security policy to be specified in a high-level language, which is then interpreted by a policy script interpreter.

As the variety and the sophistication of attacks grow, early detection of potential attacks will become crucial in mitigating the subsequent impact of these attacks (see, e.g., [16, 23, 25, 26, 29, 24, 33, 38]). Thus, intrusion detection systems would need to become even more sophisticated, in particular for traffic monitored at high speed (Gbit/sec) links, and it becomes imperative for the next generation of NIDS to:

- provide general analysis over headers and contents of elements in network data streams (e.g., IP traffic, BGP update messages) to detect potential attack signatures.

- provide highly flexible mechanisms for specifying known attack signatures over these network data streams.
- provide efficient (wire-speed) mechanisms for checking these signatures, to identify and mitigate high speed attacks.

In this paper, we explore the utility of a general-purpose data stream management system (see, e.g., [2, 1, 4, 11]), in particular, Gigascope [13–15, 12, 20], for this purpose. We argue that Gigascope has both the *functionality* and the *performance* to serve as the foundation for the next generation of network intrusion detection systems.

The rest of this paper is structured as follows. Section 2 presents the main features of Gigascope’s query language in an example driven fashion. Section 3 describes a few representative network-based attacks, and illustrates how Gigascope can be used to aid in the detection of these attacks. Finally, Section 4 describes aspects of Gigascope’s run-time architecture that enables high performance attack detection.

## 2 Gigascope

Gigascope is a high-performance data stream management system (DSMS) designed for monitoring of networks with high-speed data streams, which is operationally used within AT&T’s IP backbone [13–15, 12, 20]. Gigascope is intended to be adaptable so it can be used as the primary data analysis engine in many settings: traffic analysis, performance monitoring and debugging, protocol analysis and development, router configuration (e.g., BGP monitoring), network attack and intrusion detection, and various ad hoc analyses. In this section, we focus on the query aspects of Gigascope, and defer a discussion of Gigascope’s high-performance implementation until Section 4.

Gigascope’s query language, GSQL, is a pure stream query language with an SQL-like syntax, i.e., all inputs to a GSQL query are data streams, and the output is a data stream [20, 27]. This choice enables the composition of GSQL queries for complex query processing, and simplifies the implementation. Here, we present the main features of GSQL in an example driven fashion. Later, in Section 3, we show how GSQL can be used to detect various network attacks.

### 2.1 Data Model

Data from an external source arrives in the form of a sequence of data packets at one or more interfaces that Gigascope monitors. These data packets can be IP packets, Netflow packets, BGP updates, etc., and are interpreted by a protocol. The Gigascope run-time system interprets the data packets as a collection of fields using a library of interpretation functions. The schema of a *protocol stream* maps field names to the interpretation functions to invoke [20].

```

PROTOCOL packet {
    uint time get_time (required, increasing);
    ullong timestamp get_timestamp (required, increasing);
    uint caplen get_caplen;

```

```

    unit len get_len;
}

PROTOCOL Ethernet (packet) {
    ullong Eth_src_addr get_eth_src_addr (required);
    ullong Eth_dst_addr get_eth_dst_addr (required);
    ...
}

PROTOCOL IP (Ethernet) {
    uint ipversion get_ip_version;
}

PROTOCOL IPV4 (IP) {
    uint protocol get_ipv4_protocol;
    IP sourceIP get_ipv4_source_ip;
    IP destIP get_ipv4_dest_ip;
    ...
}

```

Network protocols tend to be layered, e.g., an IPV4 packet is delivered via an Ethernet link. As a convenience, the protocol schemas have a mechanism for field inheritance (specified in parentheses). For example, the `Ethernet` protocol contains all the fields of the `packet` protocol, as well as a few others.

## 2.2 Filters

A *filter* query selects a subset of tuples of its input stream, extracts a set of fields (possibly transforming them), then outputs the transformed tuples in its output stream. The following query extracts a set of fields for detailed analysis from all TCP (protocol = 6) packets.

```

Q1q: SELECT    time, timestamp, sourceIP, destIP,
           source_port, dest_port, len
FROM      TCP
WHERE     protocol = 6

```

Gigascopex supports multiple data types (include IP), and multiple operations on these data types. The following query extracts a few fields from the IPV4 tuples whose `sourceIP` matches `128.209.0.0/24`, and names the resulting data stream as `fq` (this can then be referenced in subsequent GSQL queries).

```

Q2q: DEFINE    { query_name fq; }
SELECT    time, sourceIP, destIP
FROM      IPV4
WHERE     sourceIP & IP_VAL '255.255.255.0' = IP_VAL '128.209.0.0'

```

### 2.3 User-Defined Functions

While GSQL has a wide variety of built-in operators, there are situations where a user-defined function would be more appropriate. Gigascope permits users to define functions, and reference them in GSQL queries. The following query, for example, uses longest prefix matching on the `sourceIP` address against the local prefix table to extract data about IPV4 packets from local hosts.

```
Q1f: SELECT  time/60, sourceIP
        FROM    IPV4
        WHERE   getlpmid(sourceIP, 'localprefix.tbl') > 0
```

### 2.4 Aggregation

The following *aggregation* query counts the number of IPV4 packets and the sum of their lengths from each source IP address during 60 second epochs.

```
Q1g: SELECT  tb, sourceIP, count(*), sum(len)
        FROM    IPV4
        GROUP BY time/60 as tb, sourceIP
```

Aggregation can be combined with user-defined functions to create sophisticated analyses. The following aggregation query uses a group variable computed using a user-defined function, to count the number of IPV4 packets and the sum of their lengths from each local host during 60 second epochs.

```
Q2g: SELECT  tb, localHost, count(*), sum(len)
        FROM    IPV4
        WHERE   getlpmid(sourceIP, 'localprefix.tbl') > 0
        GROUP BY time/60 as tb,
                 getlpmid(sourceIP, 'localprefix.tbl') as localHost
```

### 2.5 Merges and Joins

A GSQL *merge* query permits the union of streams from multiple sources into a single stream, while preserving the temporal (ordering) properties of one of the (specified) attributes. The input streams must have the same number and types of fields, and the merge fields must be temporal and similarly monotonic (both increasing or both decreasing). For example, the following query can be used to merge data packets from two simplex *physical* (optical) links to obtain a full view of the traffic on a *logical* link. Such merge queries have proven very useful in Gigascope for network data analysis.

```
Q1m: DEFINE  { query_name logicalPktsLink; }
        MERGE  O1.timestamp : O2.timestamp
        FROM    opticalPktsLink1 O1, opticalPktsLink2 O2
```

A GSQL *join* query supports the join of two data streams, with a temporal join predicate (possibly along with other predicates), and will emit a tuple for every pair of tuples from its sources that satisfy the predicate in the GSQL *WHERE* clause. The following query, for example, computes the delay between a `tcp_syn` and a `tcp_ack`.

```

Q1j: SELECT  S.tb, S.sourceIP, S.destIP, S.source_port,
            S.dest_port, (A.timestamp - S.timestamp)
FROM      tcp_syn S, tcp_ack A
WHERE     S.sourceIP = A.destIP and S.destIP = A.sourceIP and
          S.source_port = A.dest_port and S.dest_port = A.source_port
          S.tb = A.tb and S.timestamp <= A.timestamp and
          (S.sequence_number + 1) = A.ack_number

```

Joins can be combined with aggregates for complex GSQL queries.

## 2.6 User-Defined Aggregation and Sampling

GSQL permits users to define aggregate functions (UDAFs), and reference them in queries, just like regular aggregates [12]. The specification of the UDAF consists of multiple functions: *INITIALIZE* (which initializes the state of a scratchpad space), *ITERATE* (which inserts a value to the state of the UDAF), *OUTPUT* (to support multiple return values from the same UDAF computation), and *DESTROY* (which releases UDAF resources).<sup>3</sup>

For example, using GSQL's UDAF mechanism, approximate quantile streaming algorithms can be coded, and accessed like in the following query, to compute the median value of `len` for each source IP address and 60 second epoch:

```

Q1q: SELECT  tb, sourceIP, count(*), percentile(len,50)
FROM      IPV4
GROUP BY  time/60 as tb, sourceIP

```

The UDAF mechanism is useful to obtain point values (e.g., median packet length), but it is cumbersome for obtaining set values, such as in returning a sample of the data stream (e.g., a subset-sums or a reservoir sample). Given the utility of sampling to analyze high-speed streams, GSQL supports a sampling operator that can be specialized by users to implement a wide variety of stream sampling algorithms [21]. The key observation employed is that even though there are many differences between various stream sampling algorithms, they follow a common pattern. First, a number of items are collected from the original data stream according to a certain criterion (possibly with aggregation in the case of duplicates); this is the *insert* phase. Then, if a condition on the sample is triggered (e.g., the sample is too large), the size of the sample is reduced according to another criterion; this is the *compress* phase. This alternation of insert and compress phases can be repeated several times in each epoch. At the end of the epoch, the sample is output; this is the *output* phase. For example, the following query will report the 100 most common source IP addresses within a 60 second epoch.

<sup>3</sup> Additional functions are needed to deal with Gigascope's two-level architecture, which we do not discuss further.

```

Q2#: SELECT  tb, sourceIP
        FROM    IPV4
        GROUP BY time/60 as tb, sourceIP
        CLEANING WHEN    local_count(100) = TRUE
        CLEANING BY      count(*) < current_bucket() - first(current_bucket())

```

## 2.7 Query Set

Complex analyses are best expressed as combinations of simpler pieces. By permitting GSQL queries to be named, and re-used in the `FROM` clause of other GSQL queries, a set of inter-related queries, forming a query DAG, can be defined.

## 3 Attacks

A large variety of network-based attacks have been discussed in the literature, including viruses, worms, DDoS attacks, etc. (see, e.g., [10, 9, 16, 23, 25, 26, 29, 24, 33, 36, 38]). Here, we discuss a few representative attacks, and illustrate how Gigascope can be used to aid in the detection of these attacks.

### 3.1 Denial of Service

A *denial of service* (DoS) attack is characterized by an explicit attempt by attackers to prevent legitimate users of a service from using that service [7]. DoS attacks have been among the most common form of Internet attacks. The basic form of a DoS attack is to consume scarce computer and network resources, such as kernel data structures, CPU time, memory and disk space, and network bandwidth.

*Email Bombing*: An example DoS attack that attempts to consume system and network resources is *Email Bombing*, where attackers send excessively many and large e-mail messages to one or more accounts at a specific victim site [8]. When the attacker makes use of a dispersed set of sources to coordinate such an attack, it is referred to as a distributed DoS (DDoS) attack.

*Email Bombing* can be detected at the victim site if email is sluggish, possibly because the mailer is trying to process too many messages. An alternative way of checking for this possibility is to monitor the SMTP traffic entering a protected network using Gigascope, and check for hosts that show significant deviations in expected traffic at port 25/SMTP. The following simple GSQL query can track the total SMTP traffic for individual destination IP addresses. Deviations can be monitored by comparing recent behavior with more historical trends.

```

Q1dos:DEFINE  { query_name smtp_perhost; }
        SELECT  tb, destIP, count(*), sum(len)
        FROM    TCP
        WHERE   protocol = 6 and dest_port = 25
        GROUP BY time/60 as tb, destIP

```

Note that, since the number of destination IP addresses in a protected network is likely to be limited, the number of groups created by this query would not explode, even under email bombing. This is similar to “semi-streaming” where we maintain statistics per group or entity [31]. Only the count of the number of packets, and the sum of the packet lengths, would increase for victim hosts.

If the number of destination IP addresses in a network is very large, one can use GSQL’s sampling mechanism to keep track of the destination IP addresses, e.g., with the largest counts, using a variant of query  $Q_2^y$ .

*TCP SYN Flood:* A more complex attack against network connectivity, by consuming kernel data structures, is the TCP SYN Flood attack [6], which exploits the 3-way handshake used to establish a TCP connection between a sender and a receiver. In a normal scenario, a sender initiates a TCP connection by sending a SYN packet, the receiver responds with a SYN/ACK packet, and the sender completes the 3-way handshake with an ACK packet. After sending the SYN/ACK packet, the receiver allocates connection resources (kernel data structures) to remember the pending connection for a pre-specified amount of time. A TCP SYN Flood attack occurs when an attacker repeatedly sends SYN packets, typically with different source addresses, causing the receiver to deplete its connection resources, preventing service to legitimate users.

In principle, TCP SYN Flood can be identified by correlating the SYN packets with matching ACK packets in the stream of TCP packets, and alarming when too many SYN packets in a specified time interval appear to be unmatched. The GSQL query set for this purpose,  $Q_2^{dos}$ , makes use of joins, as shown below. The outer join ensures that output tuples will be computed even when there are no matched SYN packets in an epoch. Note that this is an estimate since in certain loss conditions, and due to epoch boundary issues, we might get approximate results.

```

 $Q_2^{dos}$ :DEFINE { query_name toomany_syn; }
SELECT A.tb, (A.cnt - M.cnt)
OUTER JOIN FROM all_syn_count A, matched_syn_count M
WHERE A.tb = M.tb

DEFINE { query_name all_syn_count; }
SELECT S.tb, count(*) as cnt
FROM tcp_syn S
GROUP BY S.tb

DEFINE { query_name matched_syn_count; }
SELECT S.tb, count(*) as cnt
FROM tcp_syn S, tcp_ack A
WHERE S.sourceIP = A.destIP and S.destIP = A.sourceIP and
S.source_port = A.dest_port and S.dest_port = A.source_port
S.tb = A.tb and S.timestamp <= A.timestamp and
(S.sequence_number + 1) = A.ack_number
GROUP BY S.tb

```

Over a high-speed (e.g., 1 Gbit/sec) link, one could see up to 3 million SYN packets per second [29]. In the worst-case, for reasonably large (multi-second) round-trip times, this may require too much memory to compute the join in `matched_syn_count`. In such cases, one could sample random SYN packets in the incoming stream (see Section 4.3), and check if they are matched (see, e.g., [17]). A sampling algorithm like reservoir sampling [37], which has been instantiated using GSQL's sampling operator, would suffice for this task.

Alternatively, one could simply count the number of SYN packets and the number of ACK packets in specified windows, and declare the possibility of an attack if there are more of the former than of the latter (as advocated by [38]). The query in Gigascope for this approach is shown below.

```

Q2dos:DEFINE    { query_name tomany_syn; }
                SELECT  A.tb, (S.cnt - A.cnt)
                OUTER JOIN FROM all_syn_count S, all_ack_count A
                WHERE    S.tb = A.tb and (S.cnt - A.cnt) > 0

                DEFINE  { query_name all_syn_count; }
                SELECT  S.tb, count(*) as cnt
                FROM    tcp_syn S
                GROUP BY S.tb

                DEFINE  { query_name all_ack_count; }
                SELECT  A.tb, count(*) as cnt
                FROM    tcp_ack A
                GROUP BY A.tb

```

### 3.2 Worms and Viruses

A *worm* is self-propagating malicious code [9]. Unlike a *virus*, which requires a user to do something (such as opening an infected email attachment) for its negative impact, a worm exploits vulnerabilities in the underlying operating system to inflict its damage, and to replicate and propagate by itself. They have been widely discussed in the popular press, because of the significant damage they have caused to the productivity and infrastructure of users.

Viruses rely on user action for their propagation, and hence tend to spread slowly. However, the highly automated nature of worms, along with the relatively widespread nature of the vulnerabilities they exploit allows a large number of systems to be quickly compromised. For example, the `Code Red` worm exploited a vulnerability in Microsoft IIS servers, and infected more than 250,000 systems in about 9 hours on July 19, 2001. As another example, the `Slammer` worm exploited a vulnerability in Microsoft's SQL Server 2000 code, and affected nearly 100,000 hosts in 10 minutes on January 25, 2003. Some worms include built-in DoS attack payloads, while others have web site defacement payloads (e.g., `Code Red`). But, often, their biggest impact is in the collateral damage they cause as they rapidly propagate through the Internet.

*Known Worms:* Worms can be identified by their payload, and their specific mechanism of propagation. For example, activity of the `Slammer` worm is identifiable in a network by the presence of 376-byte UDP packets, destined for port 1434/UDP of SQL Server, using the following query.

```

 $Q_1^{wv}$ :DEFINE { query_name slammer_worm; }
      SELECT  tb, destIP, count(*)
      FROM    UDP
      WHERE   protocol = 17 and dest_port = 1434 and total_ipv4_length = 376
      GROUP BY time/60 as tb, destIP

```

A number of such header profiles have been identified by detailed traffic analysis [28], and can be encoded directly as GSQL queries.

*Unknown Worms:* Since worms are self-replicating, ongoing worm propagation should be reflected in the presence of higher than expected *string similarity* among the payloads of network packets. This similarity is due to the unchanging portions of the worm packet payload, which is expected to be present even in polymorphic worms. This intuition has been exploited by various systems like EarlyBird [33] and Autograph [25], which use the frequency of substrings in packet payloads to generate signatures of sources of content similarity (which in turn are indicative of potential worms). A GSQL query akin to  $Q_2^u$  could be used to compute heavy hitters on the substring counts of the payload, for this purpose.

Recent work has also examined the utility of the inverse distribution (for a given frequency  $f$ , the number of substrings that appear with that frequency) to permit faster detection of potential worms [24]. The following GSQL query can be used for computation of the inverse distribution.

```

 $Q_2^{wv}$ :DEFINE { query_name inverse_distrib; }
      SELECT  B.tb, B.cnt, COUNT(*) AS invCnt
      FROM    base_distrib B
      GROUP BY B.tb, B.cnt

      DEFINE { query_name base_distrib; }
      SELECT  C.tb, C.SId, COUNT(*) AS cnt
      FROM    ContentStrings C
      GROUP BY C.tb, C.SId

```

The cost of this query depends on the number of distinct substrings over all payloads, which is independent of the frequency of worm propagation.

### 3.3 Probing for Vulnerability

Attacks exploit known vulnerabilities in services. A typical precursor to attacks is the identification of machines that have specific services available, and hence can be potentially exploited. This takes the form of an attacker probing for open ports on a set of host machines (see, e.g., [23, 29]).

*Ingress Detection:* To determine if a port is open, an attacker sends a packet to a host, attempting to connect to the specific port. If the target host is listening on that port, it will respond by opening a connection with the attacker. This implies that during the probing phase, the attacker would not spoof the `sourceIP` address. By monitoring the number of distinct `(destIP, dest_port)` pairs with the same `sourceIP`, one can check for anomalous activity using the following GSQL query.

```
Q1pv: SELECT  tb, sourceIP, count_distinct(PACK(destIP, dest_port)) AS cnt
          FROM    TCP
          GROUP BY time/60 as tb, sourceIP
```

A simpler GSQL query, below, simply tracks the number of distinct targets probed (potentially from different hosts, as would arise in a distributed vulnerability probe), and uses an anomalous increase in this number as an indicator of suspicious activity.

```
Q2pv: SELECT  tb, count_distinct(PACK(destIP, dest_port)) AS cnt
          FROM    TCP
          GROUP BY time/60 as tb
```

*Egress Detection:* If the target host does not have a listening process on a port, a different kind of response may be generated. For example, a packet sent to such a UDP port may generate an ICMP “port unreachable” response, while a packet sent to such a TCP port may generate an RST packet in response. Vulnerability probes (or, port scans) can hence be also identified by monitoring the number of distinct destination addresses generating such responses [29]. This can be easily captured by a variant of  $Q_2^{pv}$ , above.

## 4 Scalability

GigascopE is designed for monitoring very high speed data streams, using inexpensive processors. For example, in [22], non-trivial query sets were run at over 200,000 packets/sec, while using only 38% of one CPU in a two CPU system. To accomplish this goal, GigascopE uses an architecture optimized for its particular applications, incorporating unblocking using timestamps and heartbeats, a two-level architecture, and sophisticated sampling algorithms, each of which are described below.

### 4.1 Unblocking, Timestamps and Heartbeats

The GigascopE DSMS evaluates queries over potentially infinite streams of tuples. To produce useful output, it must be able to unblock operators such as aggregation, join, and union. In general, this unblocking is done by limiting the scope of output tuples that an input tuple affects. One unblocking mechanism is to define queries over windows of the input stream.

GigascopE’s technique for localizing input tuple scope is to require that some fields of the input data streams be identified as behaving like timestamps, e.g., be monotone increasing [14]. The locality of input tuples is determined by analyzing how the query

references the timestamp fields. For example, a merge or a join query must relate timestamp fields of both inputs, and an aggregation query must have a timestamp field as one of its group-by variables. For example, suppose that `time` is labeled as monotone increasing in the `TCP` stream. Then the `tb` group-by variable in Query  $Q_1^q$  (which counts the packets from each source IP address during 60 second epochs) is inferred also to be monotone increasing. When this variable changes in value, all existing groups and their aggregates are flushed to the operator’s output. The values of the group-by variables thus define epochs in which aggregation occurs, with a flush at the end of each epoch.

The timestamp analysis mechanism is quite effective for unblocking operators as long as all input streams make progress. However, if one of the input streams stalls, operators that combine two streams (such as merge, which preserves timestamp order in the output data stream) can stall, possibly leading to a system failure. This can happen, for example, when merging traffic from a gigabit primary link and a backup link (which is used only when the primary link fails, and hence usually carries almost no traffic), for attack analysis. The main problem is that while the presence of tuples in the stream carries temporal information, their absence does not. In such situations, heartbeats or punctuations (see, e.g., [35]) can be used to unblock operators.

Gigascop’s punctuation-carrying heartbeats [22] are generated by source query operators by regularly injecting the heartbeat messages carrying temporal update tuples into their output streams. A streaming operator in a subsequent query node in the query DAG emits temporal update tuples whenever it receives a heartbeat from one of its source streams. Thus, the heartbeats propagate throughout the query DAG. [22] discusses detailed implementation issues, and demonstrates the effectiveness of these heartbeats (significant reduction in memory load with a negligible CPU cost), using experiments with join and merge queries over very high-speed data streams.

## 4.2 Two-Level Architecture

Gigascop has a two-level query architecture, where the *low* level is used for data reduction and the *high* level performs more complex processing [14, 12]. This approach is employed for keeping up with high streaming rates in a controlled way. High speed data streams from, e.g., a Network Interface Card (NIC), are placed in a large ring buffer. These streams are called source streams to distinguish them from data streams created by queries. The data volumes of these source streams are far too large to provide a copy to each query on the stream. Instead, the queries are shipped to the streams. If a query  $Q$  is to be executed over source stream  $S$ , then Gigascop creates a subquery  $q$  that directly accesses  $S$ , and transforms  $Q$  into  $Q'$  which is executed over the output from  $q$ . In general, one subquery is created for every table variable that aliases a source stream, for every query in the current query set. The subqueries read directly from the ring buffer. Since their output streams are much smaller than the source stream, this *two-level architecture* greatly reduces the amount of copying (simple queries can be evaluated directly on a source stream).

The subqueries (which are called “LFTAs”, or low-level queries, in Gigascop) are intended to be fast, lightweight data reduction queries. By deferring expensive processing (expensive functions and predicates, joins, large scale aggregation), the high

volume source stream is quickly processed, minimizing buffer requirements. The expensive processing is performed on the output of the low level queries, but this data volume is smaller and easily buffered. Depending on the capabilities of the NIC, we can push some or all of the subquery processing into the NIC itself. In general, the most appropriate strategy depends on the streaming rate as well as the available processing resources. Choosing the best strategy is a complex query optimization problem, that attempts to maximize the amount of data reduction without overburdening the low level processor and thus causing packet drops.

Gigascope uses a large number of optimizations to lower the LFTA processing costs. Low-level operators are compiled into C code that are linked directly to the runtime library to avoid expensive runtime query interpretation. To ensure that aggregation is fast, the low-level aggregation operator uses a fixed-size hash table for maintaining the different groups of a `GROUP BY`. If a hash table collision occurs, the existing group and its aggregate are ejected (as a tuple), and the new group uses the old group's slot. That is, Gigascope computes a partial aggregate at the low level which is completed at a higher level. The query decomposition of an aggregate query  $Q$  is similar to that of subaggregates and superaggregates in data cube computations.

The Gigascope DSMS has many aspects of a real-time system: for example, if the system cannot keep up with the offered load, it will drop tuples. To spread out the processing load over time and thus improve schedulability, Gigascope implements traffic-shaping policies in some of its operators. In particular, the aggregation operator uses a *slow flush* to emit tuples when the aggregation epoch changes. One output tuple is emitted for every input tuple which arrives, until all finished groups have been output (or the epoch changes again, in which case all old groups are flushed immediately).

### 4.3 Sampling

The complex query set needed to analyze high-speed streams for attacks would often need to rely on approximations, using streaming algorithms, to keep up with their input. Many of these streaming algorithms compute samples (i.e., a small-sized representative of the data suitable for specific queries) in one pass over a high speed data stream. These stream sampling algorithms include generic sampling methods such as fixed-size reservoir sampling [37], as well as methods for estimating specific user-defined aggregates such as heavy hitters [30], distinct counts [18], quantiles [19], and subset-sums [3].

One approach developed in [21] is to develop a single operator that can be specialized to implement a wide variety of stream sampling algorithms. The sampling algorithms that can be implemented as specializations of the sampling operator permit a very simple communication structure, i.e., only between individual samples and the sample summary. The process of sampling is in some ways similar to that of aggregation, since they both collect and output sets of tuples that are representative of the input, while achieving data reduction. This analogy leads to an efficient implementation, based on the use of multiple hash tables, of all specializations of the sampling operator.

An alternative, more flexible, approach to implementing individual stream sampling algorithms in Gigascope is with user-defined aggregate functions (UDAFs). This approach was explored in [12], where both sampling-based UDAFs and sketch-based

UDAFs were implemented. The added flexibility of the UDAF approach, even for sampling-based algorithms, is that it permits the specification of algorithms that need “inter-sample communication”, especially during the compress phase (such as the quantile algorithm of [19]). Several key performance lessons were identified. First, early data reduction is critical for complex querying of very high speed data streams, and Gigascope’s two-level architecture is highly suitable for this purpose. Second, there is often a range of early data reduction strategies to choose from for processing complex aggregates, including use of appropriate subaggregation. The most appropriate strategy depends on the streaming rate as well as the available processing resources; choosing the best strategy is a complex optimization problem, with the goal of maximizing the amount of data reduction without overburdening the low-level query processor.

## 5 Conclusion

Network-based attacks, such as DDoS attacks, worms, and viruses are now commonplace, and the variety and sophistication of attacks keeps growing over time. Early detection of potential attacks will become crucial in mitigating the subsequent impact of these attacks. Thus, it is imperative for the next generation of NIDS to:

- provide general analysis over headers and contents of elements in network data streams to detect potential attack signatures.
- provide highly flexible mechanisms for specifying known attack signatures over network data streams.
- provide efficient (wire-speed) mechanisms for checking these signatures, to identify and mitigate high speed attacks.

We argue that the Gigascope DSMS has both the functionality and the performance to serve as the foundation for the next generation of network intrusion detection systems. The *functionality* is provided by the expressive, yet high-level, GSQL query language, which supports a rich variety of features including filters, user-defined functions, user-defined aggregation and sampling, and joins. Using example GSQL queries, we have illustrated the utility of these features for discerning and specifying attack signatures. The *performance* is provided by the Gigascope architecture for monitoring very high speed data streams, incorporating features like unblocking using timestamps and heartbeats, a two-level architecture, and sophisticated sampling algorithms.

As network-based attacks evolve, Gigascope will need to evolve as well. Sophisticated cooperation between a distributed set of Gigascope installations will be needed to identify highly distributed attacks on the network infrastructure. Statistical anomaly detection algorithms, both parametric and non-parametric, will need to be expressed in the query language. Sampling and signature computations on the payload, involving re-assembly of network packets, will prove useful. We think that Gigascope will be able to meet these challenges.

## Acknowledgements

We would like to thank Balachander Krishnamurthy, Morley Mao, Shubho Sen, and Kobus Van der Merwe, for many helpful discussions.

## References

1. D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, pages 277–289, 2005. <http://www-db.cs.wisc.edu/cidr/papers/P23.pdf>.
2. D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003. <http://dx.doi.org/10.1007/s00778-003-0095-z>.
3. N. Alon, N. G. Duffield, C. Lund, and M. Thorup. Estimating arbitrary subset sums with few probes. In *PODS*, 2005.
4. A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. Stream: The Stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.
5. V. Atluri, B. Pfitzmann, and P. McDaniel, editors. *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*. ACM, 2004.
6. CERT. Cert advisory ca-1996-21 TCP SYN flooding and IP spoofing attacks. <http://www.cert.org/advisories/CA-1996-21.html>, 2000.
7. CERT. Cert coordination center: Denial of service attacks. [http://www.cert.org/tech\\_tips/denial\\_of\\_service.html](http://www.cert.org/tech_tips/denial_of_service.html), 2001.
8. CERT. Cert coordination center: Email bombing and spamming. [http://www.cert.org/tech\\_tips/email\\_bombing\\_spamming.html](http://www.cert.org/tech_tips/email_bombing_spamming.html), 2002.
9. CERT. Overview of attack trends. [http://www.cert.org/archive/pdf/attack\\_trends.pdf](http://www.cert.org/archive/pdf/attack_trends.pdf), 2002.
10. CERT. Cert/cc advisories. <http://www.cert.org/advisories/>, 2004.
11. S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003. <http://www-db.cs.wisc.edu/cidr2003/program/p24.pdf>.
12. G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Holistic UDAFs at streaming speeds. In G. Weikum, A. C. König, and S. Deßloch, editors, *SIGMOD Conference*, pages 35–46. ACM, 2004. <http://doi.acm.org/10.1145/1007568.1007575>.
13. C. D. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: High performance network monitoring with an SQL interface. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *SIGMOD Conference*, page 623. ACM, 2002. <http://doi.acm.org/10.1145/564691.564777>.
14. C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In A. Y. Halevy, Z. G. Ives, and A. Doan, editors, *SIGMOD Conference*, pages 647–651. ACM, 2003. <http://www.acm.org/sigmod/sigmod03/e proceedings/papers/ind03.pdf>.
15. C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. The Gigascope stream database. *IEEE Data Eng. Bull.*, 26(1):27–32, 2003.
16. H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In Atluri et al. [5], pages 2–11. <http://doi.acm.org/10.1145/1030086>.
17. C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, 2003. <http://doi.acm.org/10.1145/859716.859719>.

18. P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB*, pages 541–550. Morgan Kaufmann, 2001. <http://www.vldb.org/conf/2001/P541.pdf>.
19. M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD Conference*, 2001. <http://www.acm.org/sigs/sigmod/sigmod01/e-proceedings/papers/Research-Greenwald-Khanna.pdf>.
20. T. Johnson. *GSQ* users manual. Accessible from <http://www.research.att.com/~johnson>, 2005.
21. T. Johnson, S. Muthukrishnan, and I. Rozenbaum. Sampling algorithms in a stream operator. In *SIGMOD Conference*, 2005.
22. T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. A heartbeat mechanism and its application in Gigascope. In *VLDB Conference*, 2005.
23. 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. IEEE Computer Society, 2004. <http://csdl.computer.org/comp/proceedings/sp/2004/2136/00/21360211abs.htm>.
24. V. Karamcheti, D. Geiger, Z. Kedem, and S. Muthukrishnan. Detecting malicious network traffic using inverse distributions of packet contents. In *MineNet*, 2005.
25. H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium*, pages 271–286. USENIX, 2004. <http://www.usenix.org/publications/library/proceedings/sec04/tech/kim.html>.
26. R. R. Kompella, S. Singh, and G. Varghese. On scalable attack detection in the network. In A. Lombardo and J. F. Kurose, editors, *Internet Measurement Conference*, pages 187–200. ACM, 2004. <http://doi.acm.org/10.1145/1028812>.
27. N. Koudas and D. Srivastava. Data stream query processing. In *ICDE*, page 1145. IEEE Computer Society, 2005. <http://csdl.computer.org/comp/proceedings/icde/2005/2285/00/22851145abs.htm>.
28. A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. In *SIGCOMM*, 2005.
29. K. Levchenko, R. Paturi, and G. Varghese. On the difficulty of scalably detecting network attacks. In Atluri et al. [5], pages 12–20. <http://doi.acm.org/10.1145/1030087>.
30. G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357, 2002. <http://www.vldb.org/conf/2002/S10P03.pdf>.
31. S. Muthukrishnan. Data stream algorithms and applications. <http://www.cs.rutgers.edu/~muthu/stream-1-1.ps>, 2005.
32. V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999. [http://dx.doi.org/10.1016/S1389-1286\(99\)00112-7](http://dx.doi.org/10.1016/S1389-1286(99)00112-7).
33. S. Singh, C. Egan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *OSDI*, pages 45–60, 2004. <http://www.usenix.org/events/osdi04/tech/singh.html>.
34. Snort. The de facto standard for intrusion detection. <http://www.snort.org>.
35. P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng.*, 15(3):555–568, 2003. <http://www.computer.org/tkde/tk2003/k0555abs.htm>.
36. US-CERT. Technical cyber security alerts. <http://www.us-cert.gov/cas/techalerts/index.html>, 2005.
37. J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.
38. H. Wang, D. Zhang, and K. G. Shin. Detecting SYN flooding attacks. In *INFOCOM*, 2002. <http://www.ieee-infocom.org/2002/papers/800.pdf>.