Count Me In: Viable Distributed Summary Statistics for Securing High-Speed Networks

Johanna Amann[1], Seth Hall[1,2], and Robin Sommer[1,2]

[1] International Computer Science Institute
[2] Lawrence Berkeley National Laboratory

Abstract. Summary statistics represent a key primitive for profiling and protecting operational networks. Many network operators routinely measure properties such as throughput, traffic mix, and heavy hitters. Likewise, security monitoring often deploys statistical anomaly detectors that trigger, e.g., when a source scans the local IP address range, or exceeds a threshold of failed login attempts. Traditionally, a diverse set of tools is used for such computations, each typically hard-coding either the features it operates on or the specific calculations it performs, or both. In this work we present a novel framework for calculating a wide array of summary statistics in real-time, independent of the underlying data, and potentially aggregated from independent monitoring points. We focus on providing a transparent, extensible, easy-to-use interface and implement our design on top of an open-source network monitoring system. We demonstrate a set of example applications for profiling and statistical anomaly detection that would traditionally require significant effort and different tools to compute. We have released our implementation under BSD license and report experiences from real-world deployments in large-scale network environments.

1 Introduction

Researchers and operators alike routinely measure statistical properties of network traffic, such as throughput, traffic mix, and "heavy hitters"; both for traffic profiling and control, as well as for specific security purposes when aiming to spot activity that "doesn't look right". For the latter, statistical anomaly detection proves particularly valuable by reporting activity that exceeds levels one would normally expect so see, such as during port and address scans, login brute-forcing, and application-layer vulnerability probing. Traditionally, we find a diverse set of approaches in use for implementing such monitoring, typically limited to traffic features readily available in existing data sets such as NetFlow records, SNMP counters, IDS output, or system logs; and often implemented in the form of ad-hoc shell scripts processing files offline in batches. While conceptually most profiling and anomaly detection tasks leverage just a rather small set of statistical primitives, existing approaches tend to hard-code either the feature set they operate on or the specific computation they perform; and regularly both. Consequently, sites find it challenging to later adapt a setup to changes in

requirements, miss out on opportunities for reuse in different settings, and see little incentive to optimize an implementation for performance.

In this work we present a novel *summary statistics framework* that facilitates a wide array of typical profiling tasks and security applications. Our system processes high-volume packet streams in real-time, operates transparently on arbitrary features extracted from all levels of the protocol stack, and aggregates results across independent monitoring points distributed across a network. We focus on providing a transparent, easy-to-use user interface that, in particular, hides the communication in distributed setups behind a simple, intent-based API. We target operational deployment in large-scale network environments, with link capacities of 10 GE and beyond; and we implement our design on top of an existing open-source network monitoring system that is regularly deployed in such settings. Our implementation includes a set of probabilistic data structures to support memory-efficient operation, as well as a plugin interface that allows users to extend the supplied range of statistical primitives. We demonstrate a number of real-world example security applications, including computation of traffic matrices, detection of IP scans and SQL injection, and real-time "top-k" measurements to determine, e.g., the most frequent hosts, HTTP destinations, or DNS requests. We furthermore interface the latter to a browser-based visualization library that renders the current "heavy hitters" in real-time for immediate inspection. We evaluate our system in terms of the overhead it imposes on the underlying network monitor with regards to CPU, memory, and inter-node communication; and we find it to scale well in realistic settings. We have released our implementation as open-source software under a BSD-license as part of the recent release of the underlying network monitor. It is in deployment now at a broad range of sites, where it helps operations to protect their networks.

We structure the remainder of this paper as follows. §2 presents the motivation and design of the summary statistics framework, and §3 describes our implementation. §4 demonstrates a number of real-world example applications, along with experiences from operational deployments. In §5 we asses performance characteristics. §6 discusses related work, and we conclude in §7.

2 Design

Our work introduces a novel summary statistics framework that offers a flexible platform to compute a wide variety of summary statistics in large-scale operational network environments. In the following we first review the underlying motivation and then walk through a number of design aspects for the framework.

2.1 Motivation

While summary statistics constitute a crucial ingredient for many operational network monitoring tasks, existing implementations generally cater to a specific application or setting (see §6). Our framework instead aims to enable users to define their *own* statistics, with no limitation on what input or computation to

use. The challenge with this approach lies in designing a system that provides such flexibility while also offering the efficiency required to accommodate large-scale deployment in high-performance settings.

To illustrate our motivation, consider the task of counting. Researchers and operators alike tend to ask questions about their networks such as "How many local IP addresses do we see?", "What system produces the most traffic?", "What are the prevalent application protocols?", and "Is there any host unsuccessfully querying a large number of DNS names?". Traditionally, answering such questions requires using a variety of different tools. While conceptually these questions all come down to counting features, they process conceptually quite different information, from packet-level information like IP addresses to complex application-layer attributes such as rejected DNS requests. Our goal is to unify the computing of such results within a single system that decouples feature extraction from the statistical infrastructure, providing users with a platform for answering a wide range of their questions.

From experience with research and operations, we identify two overall types of applications that network-based statistics tend to support: *(i) network profiling* aims to answer questions as sketched above for characterizing ongoing activity; and *(ii) statistical anomaly detection* identifies situations where observed features exceed an expected range, potentially leading to a security incident. Regarding the former, while the range of possible profiling tasks is large, most consist of a rather small set of computational primitives, such as summation and aggregation of values, standard set operations, computing simple measures such as maximum and average, and also sorting. Turning to statistical anomaly detection, one typically finds conceptually simple measures deployed operationally; often just straightforward threshold schemes that trigger when activity exceeds a predetermined value or ratio. The most common application is scan detection, which finds hosts probing the local network by spotting an excessive number of failed attempts. While traditionally scan detection refers to IP address or TCP/UDP port probing, the concept extends to application-layer features as well, including probing web servers with requests, email servers with destination addresses, DNS servers with lookups, and also probing for vulnerable systems by trying application-layer exploits. While many monitoring systems support profiling and/or statistical anomaly detection, their implementations typically hardcode either the feature set they operate on or the specific calculation they perform.

2.2 Objectives

We identify the following objectives for our summary statistics framework.

Simple, yet flexible user interface. The interface that the framework exposes to the user should be easy to understand and use, yet sufficiently flexible to support computation of a wide range of target statistics.

Data agnostic. The framework should be data agnostic and avoid imposing any constraints on the features it operates on.

Extensibility. The available statistical functionality should be adaptable and extensible to computations not supported out-of-the-box.

Real-time operation. The framework should process input in real-time and provide results, including alarms, as quickly as possible.

Scalability. The framework needs to scale to large networks, including support for multiple traffic sources for either distributed monitoring or load-balancing purposes.

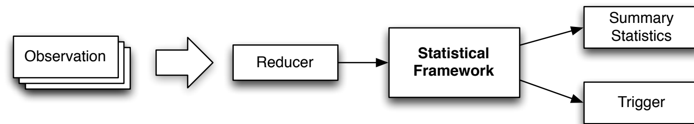

Fig. 1. Basic Architecture.

2.3 Architecture

Figure 1 summarizes the summary statistics framework's high-level architecture. It *observes* a stream of tuples $(key, value)$ in which in general both *key* and *value* represent features derived in real-time from the incoming network traffic. As it processes the stream, the summary statistics framework continuously *reduces* each key's values to an aggregate result. The framework also continuously evaluates a *predicate* on these aggregates to flag specific situations by executing corresponding *triggers*. Finally, at the end of a measurement interval, the framework reports the final *summary statistics* to the user in the form of (key, agg) pairs where *agg* is the final aggregate value for that key.

As one application example consider a simple TCP scan detector. Observations might take the form of tuples (s, d) representing failed connection attempts from a source address s to a destination address d. A reducer *Unique* would compute the number of unique destinations d for each source s, and a predicate *Threshold* would flag if that exceeds a specified limit by executing a *ScanAlarm* trigger that reports an alarm. As another example, if one wanted to compute the most popular DNS names overall, the observation values would be query names extracted from DNS traffic. One would then aggregate all values into a single global result by fixing the observation key to a static value, and deploy a "top-k" reducer that computes the k most frequently seen values among its inputs.

The summary statistics framework supports deployment in settings where the traffic is not just monitored by a single process, yet with sets of physically separated monitors, as long as the instances see disjunct packet streams. This could be at different ingress points of a large network, or in a cluster setting where a load-balancer splits up the overall traffic to sent individual slices to separate monitoring backends (as, e.g., in [25]). In such a setting the summary statistics framework computes results transparently for the overall traffic aggregate, similar to what a single instance would produce if it were seeing all the traffic at one

location. To accommodate such settings, we extend the basic architecture into a distributed setup in which independent sensors reduce values locally first, and then at the end of a measurement interval forward their results to a master server that merges them into global aggregates. That server then also evaluates the predicates and executes the trigger. Figure 2 illustrates the distributed setting. As the reduced intermediary results will typically be small in volume, the architecture scales well with increasing numbers of sensors.

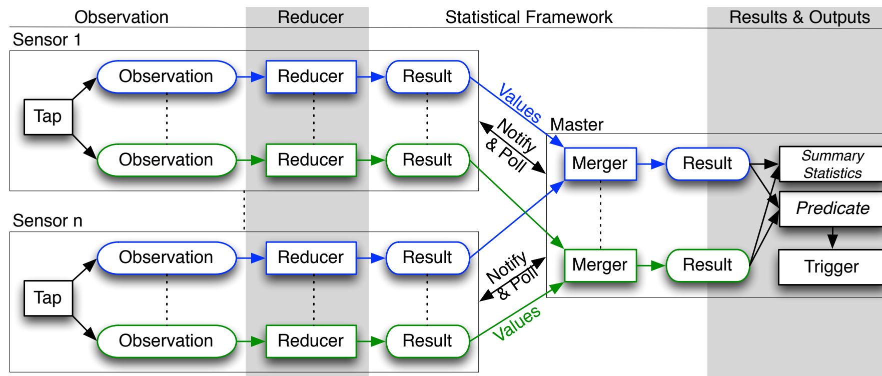

Fig. 2. Distributed Architecture.

As one additional ingredient to the distributed operation, we add *result polling* that allows the server to request intermediary results from the sensors on demand. Normally, the server would evaluate predicates only at the end of a measurement interval once it has received all the local results. As that however might introduce a potentially significant delay until triggers execute, we introduce two additional optimizations. First, we allow the server to poll sensors for their current values on demand, even before the end of the measurement interval. It can then already evaluate the predicate on the received intermediate values. Polling alone however would not reduce trigger latencies sufficiently without also causing significant communication overhead. Hence, we furthermore provide the sensors with a notification mechanism to signal that their intermediate local values have changed sufficiently to warrant requesting an update. For example, for a threshold computation a sensor could notify the server once it has observed 20% of the specified limit locally, with the assumption that other sensors are likely seeing similar activity and, hence, globally the threshold might have been crossed. Upon receiving the notification, the server polls all the sensors, executes the predicate, and runs the trigger if applicable.

2.4 Reducers

We conclude this section by examining the properties of reducers in more detail, as they have to satisfy a number of requirements to fit with the framework's operation. Recall that a reducer processes $(key, value)$ pairs, aggregating them into outputs (key, agg) where agg is an aggregate of all of key's value as determined by the reducer's computation. In the following we first look at constraints we impose on reducers, and then present a set of examples that satisfy these requirements and all come with our implementation.

Composable results. As a crucial property for reducers in the distributed setting, we require *composability*, i.e., support for aggregating the sensor's local results at the server-side. As a simple example, a reducer adding up numerical observations is trivially composable: the global sum is the total of the local results. This constraint can however be challenging to satisfy for other operations, even if conceptually simple. For instance, when sampling input randomly, deciding which samples to choose during merging without biasing the result is non-trivial.

Constant Memory Size. When processing observations reducers typically have to keep internal state during the measurement interval (e.g., the sum of all values so far). However, to reliably support computing statistics on arbitrary input volumes we require a constant bound on the amount of memory a reducer maintains. Due to this restriction, our framework can, e.g., not compute the *median* across observations.

Meaningful intermediary results. To support arbitrary measurement intervals as well as continuous predicate evaluation, a reducer's intermediary values must be meaningful on their own at any time. This is again obviously the case for a sum, which always reflects the current total; but less so for some of the more complex data structures.

Summation, Average, Deviation, Variance, Maximum, Minimum. These standard statistical measures are frequently used for traffic measurement tasks. They all support a stream-based calculation model where the reducer holds just the current result reflecting all observations seen so far, updating it when a new observation comes in.

Unique. Determining the number of unique observations proves highly useful for many network-oriented measurement tasks. However, a naive set-based implementation would have a memory requirement of $O(n)$ with n representing the number of observations, rendering it infeasible to use. Instead, we use a probabilistic version based on the HyperLogLog data structure (HLL; [10]). HLL provides approximate results with well-defined error margins. It uses $O(1)$ memory, is composable, and provides meaningful intermediary results.

Top-k. Finding the top-k "heavy hitters" represents another common task. However, similar to *Unique*, a naive implementation requires $O(n)$ memory, with n the number of observations. We thus likewise choose a probabilistic version instead: Metwally et al's algorithm [17], which in addition also provides estimates on the number of times specific elements were seen. Just as HLL, the algorithm satisfies all our constraints, including composability (see [5]).

Sampler. For many applications it is not only interesting to know the final result itself yet also to receive with it a sample of individual contributing values (e.g., when seeing an unusually high number of DNS requests from a single source, seeing a few example requests can prove illuminating). We support that by providing a "Sample" reducer that maintains a fixed number of k uniformly distributed samples taken out of the complete observation stream. By using reservoir sampling [26], we are able to satisfy all our constraints.

2.5 Comparison with MapReduce

It is no accident that our model, and terminology, shares similarities with MapReduce [6]. They both operate in similar phases. The "Map" step of MapReduce corresponds to taking observations in our model; in either approach, input data maps to a key and a value. The "Reduce" step of MapReduce is equivalent to our server-side merging of results computed locally at the sensors. What we call a "reducer" indeed corresponds to a "combiner" in a refinement of the MapReduce model: combiner functions merge partial results before data gets forwarded [6]. The underlying reason for this naming difference is that in our design the main part of the data reduction does indeed occur already on the sensor nodes.

One difference between the two models concerns the input side. While either approach assumes suitably pre-split sets of input, MapReduce does not tie them to a specific compute node. In our model, by tapping disjunct packet sources yet not further dividing up their inputs, we implicitly link each source with one specific sensor that processes it. While this remains less flexible, it provides a significant performance advantage by effectively leveraging the network itself for partitioning input appropriately, either indirectly by virtue of its structure (in the case of tapping different physical locations), or directly via a front-end load-balancer (in the case of a cluster setup [25]). In either case we avoid the—potentially prohibitive—performance penalty of redistributing traffic within the summary statistics framework.

Overall, we emphasize that the two approaches share significant similarities. As such, we do not consider our framework's abstract computational model the primary contribution of this work, yet rather its integration into an efficient, deployable system that provides a transparent, simple-to-use API to the user.

3 Implementation

We implement our design of the summary statistics framework on top of the Bro network monitoring platform [3,19]. Bro aligns well with our objectives as it *(i)* provides the user with the necessary flexibility through its Turing-complete scripting language; *(ii)* extracts a wide range of features from network traffic to measure; and *(iii)* supports distributed operation in cluster setups. We implement the summary statistics framework completely within Bro's scripting language, with no changes to the system's C++ core for the general functionality. As the only extension to Bro's internals, we add support for the probabilistic data structures

that some of the reducers deploy. Our implementation comes with pre-written analysis scripts that leverage its capabilities for detection of, e.g., host and port scans, traceroutes, and SQL injection attacks. In the following, we discuss our implementation in terms of its user interface (§3.1), cluster integration (§3.2), and computation plugins (§3.3) that reducers can leverage.

3.1 User Interface

The user interface of the summary statistics framework exposes a set of public functions in Bro's scripting language. In the following, we briefly sketch the main functionality available to users. As a simple example we assume the setting of a small network site that aims to track the number of connections that each local host initiates to external destinations, recording them into a log file on a hourly basis.

Measuring. Setting up the analysis requires two steps: *(i)* feeding all outgoing connections into the summary statistics framework as observations, and *(ii)* defining a corresponding summary statistic that aggregates connections by their originator addresses. For the former, the framework provides the `observe()` function, which injects a key/value pair into an observation stream. The framework supports an arbitrary number of independent streams and identifies them by user-chosen names. For the example application we hook into Bro's connection processing and pass on every connection attempt originating from a local host: [3]

```
event connection_attempt(c: connection) {
  [... return if connection does not originate from the local network ... ]
  SumStats::observe(
    stream  = "host-conn-attempts";   # Name of observation stream
    key     = c.originator;           # Observation key (IP address)
    value   = 1;                      # Observation value ("one attempt")
  );
}
```

For the second step we first define a reducer that adds up connection attempts:

```
  local r1: SumStats::Reducer = [
    stream = "host-conn-attempts";    # Name of observation stream
    apply  = SumStats::SUM;           # Reducer plugin to use
   ];
```

Here, we link the reducer to the observation stream to process, *host-conn-attempts*, and specify *Summation* as the statistical operation to apply to the incoming values. For a list of currently supported operations, see §2.4; users can add further ones by supplying custom plugins (see §3.3).

Next, we define the actual summary statistic by calling the framework's `create()` function. In its simplest form, the function takes just four parameters:

[3] In this and later examples we simplify Bro's syntax for better readability.

```
SumStats::create(
  name     = "local-origins";        # Name of the summary statistic
  epoch    = 1 hour;                 # Measurement interval (epoch)
  reducers = set(r1);                # Set of reducers to deploy
  epoch_result = epoch_func;         # End of epoch callback function
);
```

With that, the summary statistic configuration is complete. During runtime, Bro will now call the `epoch_result` function each hour and provide it with the number of outgoing connections per local host. The function can process the data arbitrarily, such as by logging the information into a file.

Thresholding. We now extend the previous example to report hosts that exceed a predefined threshold of connection attempts. Here, our implementation deviates slightly from the discussion in §2. While the design provides for a generic *predicate* to check for arbitrary conditions while a computation is in progress, our implementation currently hardcodes threshold checks as the only available option. In our experience, thresholding represents the dominant application. By specifically targeting it, we can simplify both the interface (making it more intuitive for users) and the implementation (reducing complexity in the distributed setting). However, there's no conceptual limitation that would prevent us from adding the more general case in the future.

Adding a threshold check to the previous example involves passing three more parameters to the `create()` call: a function that retrieves the current measurement value for a key, a numerical threshold to compare that value with, and the trigger function to execute when the value exceeds the threshold:

```
SumStats::create(
  [...]
  threshold         = 10000.0;      # Threshold value
  threshold_val     = val_func;     # Retrieve current value
  threshold_crossed = crossed_func; # Alarm.
);
```

The `val_func` receives a key and the current intermediate reducer values for this key. It uses them to return the value to be checked against the threshold.

```
function val_func(key, val) : double {
  return val["host-conn-attempts"].sum;
}
```

In this example, `val_func` simply returns the current number of connection attempts for a host.[4] However, the function could be more complex than that. In our application, one could for example instead implement a threshold relative to the number of successful connections. For that one would add a second observation stream, say `host-conn-successes`, along with a corresponding reducer $r2$ added to the `create()` call. This modified `val_func` would then calculate percentages:

[4] As the code suggests, the state is maintained in a number of nested table structures (hash maps) indexed by the measurements.

```
function val_func(key, val) : double {
  return val["host-conn-attempts"].sum / val["host-conn-successes"].sum;
}
```

For completeness, we conclude the example by showing the trigger function that turns an exceeded threshold into an alarm via Bro's provided `NOTICE` function:

```
function crossed_func(key, val) {
  NOTICE("Host %s exceeded conn threshold: %d conn attempts", key, val);
}
```

3.2 Cluster Integration

As discussed in §2.3, the summary statistics framework targets deployment in distributed settings where a set of local vantage points contribute to a global measurement. Bro supports distributed setups through *clustering* [25]. In a Bro cluster, a set of worker nodes examines independent traffic streams and share their results through a central manager node. Each node might either monitor a physically separate point in a network or, more commonly, contribute to analyzing a single high-speed link by analyzing a smaller traffic slice that a front-end load-balancer assigns to it. Typically such load-balancing operates on a per-flow basis and, hence, satisfies our design constraint of requiring disjunct input streams in distributed summary statistics framework deployments.

Our Bro implementation closely follows the distributed design presented in §2.3, including the optimized notification/polling scheme for timely trigger execution. We put particular emphasis on hiding the increased complexity of the distributed setting from the user: the framework uses the same API for both single-instance and distributed setups; user-supplied script code works transparently in either setting. In particular, users do not need to specify which parts of their code executes where; the summary statistics framework automatically runs the respective functionality on the correct nodes (i.e., extracting observations and processing reducers on the workers; executing aggregation, thresholding, and triggers on the manager).

3.3 Computation Plugins

The framework includes support for a number of computations for reducers to deploy. Their implementations use a generic plugin interface that also allows users to add further schemes of their own. Each computation plugin implements two functions: one for adding a new observation, and one for merging computation state from different nodes; either function has also access to the time range that a observation stream spans and may include that into its calculations.

As an example, we show the implementation of the *Minimum*[5]:

```
# Update current minimum.
function add(key, val, state) {
  if ( val < state.min )
    state.min = val;
}

# Aggregate two values by taking the smaller.
function aggregate(out, in1, in2) {
  out.min = (in1.min < in2.min) ? in1.min : in2.min;
}
```

In addition to *Minimum*, our implementation also provides plugins for *Maximum*, *Sum*, *Average*, *Standard Deviation* and *Variance*, *Top-k*), *Unique*, and Sampling (see §2.4).

4 Applications and Deployment

In this section we demonstrate the summary statistics framework's capabilities with a set of example applications. The first four (scan detector in §4.1, brute-force login detector in §4.2, SQL injection detector in §4.3, traceroute detector in §4.4) ship with Bro since version 2.2, and many network sites use them operationally now. We furthermore discuss three measurement tasks (traffic matrix in §4.6, top-k in §4.5, visualization in §4.7) that we ran experimentally in production environments. For these we make the corresponding (short) implementation scripts available in a separate repository [2].

Note that these are only example applications demonstrating the capabilities of the framework. In practice, operators will evaluate the suitability of the summary statistics framework for their tasks and implement their own scripts as appropiate.

4.1 Scan Detection

Detecting port and address scans constitutes an important capability for security operations. We implemented a corresponding scan detector as a Bro script on top of the summary statistics framework. The script tracks the number of unique ports and destination addresses that each source IP attempts to connect to, generating alarms when they exceed, by default, 15 or 25 attempts within a 5 minute interval, respectively. Users can easily adjust either threshold, as well as the time interval. The script is about 160 lines long, with the bulk representing logic for connection processing and customization functionality. The core of the

[5] The actual implementation is slightly more verbose to deal with corner cases like undefined values. We also again simplify the syntax to match previous examples. Finally, we omit the definition of the state's `min` attribute, which extends a predefined data type to add plugin-specific storage that maintains the current value.

script consists of just two pairs of function calls setting up the summary statistics and feeding in observations. In particular, there is no need for code to deal with distributed Bro setups. For comparison, older Bro versions used to ship with a manually written, complex scan detection script that consisted of over 600 lines of script code, with most of that focusing on maintaining the necessary counters inside nested hash tables.

Indiana University (IU) has been running versions of our new scan detector script for more than 9 months on their 49-node Bro cluster, monitoring the site's 10 GE upstream link. Their total traffic (incoming and outgoing) peaks at about 13 Gb/s on workdays and generally averages at about 5 Gb/s. Figures 3 and 4 show the number of incoming scans to different destination addresses by time and by weekday, respectively, for subinterval of that time, as identified by our detector. At peak times, there are more than 290 unique external IP addresses conducting scans of the network each hour. In total, IU encountered 33,452 scanners from 2014-02-19 to 2014-03-20. The network operators use the script's output to automatically block external scanners at the border router in near-real time. Note that due to this automated blocking, with blocks often being triggered before the end of a monitoring interval, the numbers in this section represent a lower bound.

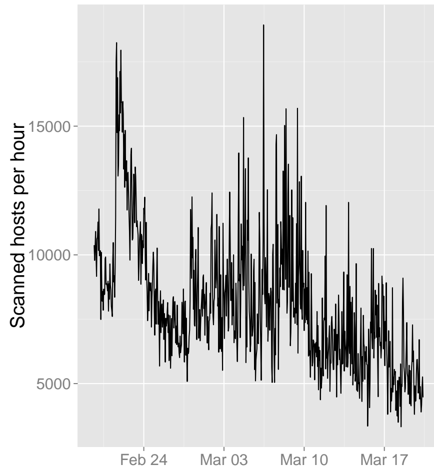

Fig. 3. Incoming address scans per hour from 2014-02-19 to 2014-03-20 at IU.

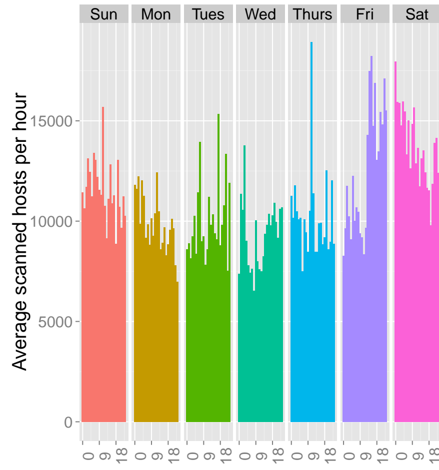

Fig. 4. Aggregate count of incoming addr. scans from 2014-02-19 to 2014-03-20 at IU.

4.2 Brute-force Login Detection

A common type of attack concerns brute-forcing accounts by trying a large number of username and password combinations. We implemented scripts to

detect such attacks for the FTP and SSH protocols. For FTP, the script counts the number of failed FTP authentication attempts and generates an alarm when it sees more than, by default, 20 attempts from a specific source to a particular destination host within 15 minutes. For SSH, Bro provides a heuristic that determines if a login succeeded or failed, based on the volume of data exchanged as well as the number of packets seen during the session. Our script counts the number of times this heuristic reports a successful login and triggers an alarm when that number exceeds 30 in a 30 minute interval. A number of sites, including Indiana University, are currently running the brute-force detection scripts in their production setups.

4.3 SQL Injection Detection

We also created script for detecting automated SQL injection attacks, using a similar thresholding approach as above. When targeting a web server, attackers often iterate through a large library of canned injection URIs within a short time frame. To detect this kind of attack, we first wrote a regular expression that matches typical injection URIs (e.g., `/site.php?site=5' and 1=1 and ''='`).[6] We then set up two summary statistic instances. Both count the number of times the regular expression matches. For the first instance, the key is the source IP address while for the second we use the destination address. In other words, the first identifies attack sources (independent of how many victims each targets), and the second reports servers under attack (independent of the number of their attackers). In addition, both summary statistic instances also apply an additional *Sample* reducer, which keeps 5 URIs that have matched the regular expression. Once one of the instances hits a configured threshold of matching requests (50 in 5 minutes by default), the detector triggers an alert email that summarizes the detected SQL injection attack, including the 5 URIs as additional context.

4.4 Traceroute detection

Traceroute detection constitutes another use-case for the summary statistics framework. While a traceroute does usually not pose a direct security threat, it may indicate reconnaissance preceding an attack. Traceroutes are however challenging to identify in clustered monitoring setups where traffic is load-balanced across different monitoring systems according to its 5-tuple of addresses, ports, and protocol. As the ICMP packets belonging to one execution will often arrive at different nodes, no single node can spot it by itself.

For our detector, we use a single summary statistic instance with two reducers. One of them counts the number of packets per host pair with TTLs lower than 10. The second counts the number of ICMP Time Exceeded messages relating to the same hosts.

[6] This turns out harder than it sounds: We have developed, and continuously refined, this regular expression for more than 5 years now by regularly evaluating network traffic and adding new cases as we discovered them. The expression has a size of more than 1,500 characters today.

We consider a traceroute to be in progress if we see at least one low-TTL packet between a pair of hosts along with at least three matching ICMP Time Exceeded messages. Leveraging the summary statistics framework allows to define such a logic at a semantic level with a single if-statement, without needing to consider the underlying traffic splitting any further. We validated this scheme by running it on the Bro cluster of the National Center for Supercomputing Applications at the University of Illinois, manually executing traceroutes and sampling the corresponding reports during normal operation. Ignoring our own activity, the large number of otherwise incoming traceroutes we saw (more than 2,000 a day) surprised us. Many of them turned out to be targeting a local content management system.

4.5 Top-k

As examples of "top-k" measurements, we wrote a script that tracks *(i)* the top-10 source and destination hosts exhibiting the most established TCP connections; *(ii)* the top-10 second-level domains in DNS queries; and *(iii)* the top-10 `Host` header values present in HTTP requests.[7] We consider only outgoing traffic and calculate rankings over both 10-minute and 1-hour intervals.

DNS domain	Upper bound	ϵ	HTTP host	Upper bound	ϵ
.akamai.net	276,592	0	b.scorecardresearch.com	123,293	0
.akamaiedge.net	185,150	0	www.google-analytics.com	111,760	0
.berkeley.edu	158,938	0	pagead2.googlesyndication.com	87,539	0
.amazonaws.com	148,584	0	ib.adnxs.com	77,521	0
.google.com	137,474	0	ad.doubleclick.net	72,156	0
.akadns.net	135,519	0	pixel.quantserve.com	70,284	0
.yuerengu.com.cn	92,210	0	www.google.com	62,996	0
.cloudfront.net	60,234	0	i1.ytimg.com	59,607	0
.spameatingmonkey.net	57,089	142	googleads.g.doubleclick.net	56,673	0
.ustiming.org	38,108	719	setiboincdata.ssl.berkeley.edu	56,513	0
Total DNS req. (exact)	4,220,837		Total HTTP requests (exact)	10,985,712	

Table 1. Top-10 outgoing DNS 2nd-level lookups and HTTP Host values (19-3-2014, 15:15–16:15).

For demonstration purposes we ran this script on a 28-node Bro research cluster operating at the University of California, Berkeley; monitoring the campus' 2x10 GE uplink connections [25]. Daytime volume averages between 3-4 Gb/s total. Table 1 shows a snapshot of the 1-hour DNS/HTTP statistics from an early Monday afternoon. Recall that the top-k calculation uses a probabilistic data

[7] The `Host` headers provides an application-level view of popular web sites, vs. just looking at IP addresses. Web site addresses have become quite meaningless today with many services running on generic cloud infrastructure.

structure and, hence, the results represent estimates. The table includes what the algorithm reported as upper bounds for the number of times it encountered each value. In addition, the table also shows the corresponding uncertainty ϵ; subtracting ϵ from the upper bound gives the lower bound. This means that, e.g., a DNS request for `.ustiming.org` was encountered between 37,389 and 38,108 times. We see that generally the error rates remain very low, considering the large amount of traffic with high numbers of unique DNS domains and HTTP hosts (154,859 and 100,269, respectively, during the shown time interval; calculated independently from logs). For these measurements, we configured the probabilistic algorithm to keep at most 1,000 different values in memory for each summary statistic at any point of time.

4.6 Traffic Matrix

The summary statistics framework can also be used to compute traffic matrices, such as for breaking down overall volume by subnets. To demonstrate this, we created a small Bro script which sets up a single summary statistics framework instance using two reducers tracking the volume of incoming and outgoing traffic by source, respectively. Additionally, the reducers define a key normalization function, which maps the source address of each individual observation to the containing /24 network in which the host resides. We deployed the top-k script on the Berkeley research cluster discussed in §4.5. Table 2 shows the output for the 5 (anonymized) subnets with the largest amount of total traffic during the observed one-hour period, out of 502 unique local subnets encountered.

4.7 Real-time Visualization

As our final application, we extended the previous "top-k" setup to visualize the results in real-time. See Fig. 5 for a screenshot. Internally, the extended Bro script uses the intermediate value update mechanism of the summary statistics framework to get current values every 15 seconds. It then sends the aggregated valued to Bro's logging framework, which supports a number of different output formats including TSV files and databases. For this application, we added support for Apache's ActiveMQ message queuing framework so that Bro can send the values directly to an ActiveMQ server. We created an HTML page that uses JavaScript for visualizing the values via a persistent WebSocket connection. After each update, the value changes are immediately reflected in the browser window.

5 Evaluation

In this section we evaluate the overhead introduced by the summary statistics framework in terms of computation, memory, and communication. Our objective

Subnet	Bytes In	Out	Total
UCB Subnet A	124G	56.0G	180G
UCB Subnet B	123G	22.7G	146G
UCB Subnet C	39.7G	48.1G	87.9G
UCB Subnet D	23.3G	2.15G	25.5G
UCB Subnet E	18.6G	1.19G	19.8G

Table 2. UCB Top-5 local subnets by total traffic (28-3-2014, 11:41–12:41).

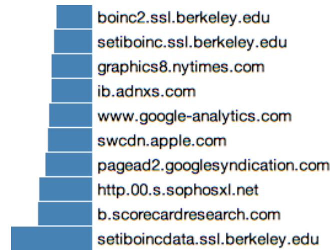

Fig. 5. Screenshot top-10 HTTP hosts (by headers) live visualization (4-4-2014, 9:28).

concerns ensuring that the implementation provides the performance necessary to operate in large-scale distributed environments. We focus on two applications: *Top-k* (§4.5), as the most resource-intensive application; and *scan detection* (§4.1), which stresses the inter-node communication the most.

5.1 Correctness

We first briefly double-check correctness of the summary statistics framework's calculations. While not directly an issue for the simpler calculations, the probabilistic data structures by design introduce errors into their results, along with worst-case bounds derived from their mathematical foundation. In Table 1, we show top-k results along with their error margins for a 1-hour measurement period in a large-scale cluster setup (see §4.5). We cross-check the reported numbers by calculating the actual top-k lists offline out of the log files that the Bro cluster produced during the same execution. We find that despite using the memory-efficient probabilistic data structure: *(i)* the summary statistics framework correctly identifies all entries in the right order in all but two cases, and *(ii)* all the actual values indeed fall within the given error margin. Regarding the former, the two exceptions concern the top sources. During our measurement the counts for 8 of the top 10 IP addresses were very close to each other. In both cases, the reported uncertainty ϵ (see §4.5) was greater than the difference to the next values. Hence, a user can indeed conclude from the numbers that while the reported ordering might not be fully correct, it must be closely matching the actual activity.

5.2 Computational Overhead

Internally, the summary statistics framework is a complex module consisting of several hundred lines of Bro script code for the basic framework, separate scripts for the plugins, and low-level core support for the probabilistic data structures. For evaluating the computational overhead that this extension introduces, we captured a packet trace of about 20-minutes at the Internet uplink of UC Berkeley (see §4.6). To keep the volume manageable we recorded just a subset of the total traffic, corresponding to what one node of the Bro research cluster

processes (i.e., $\frac{1}{28}$ of all flows).[8] The resulting trace includes 19.8 M packets and 516 K flows, at a total volume of about 15 GB.

We measure CPU load with three different configurations: *(i)* Bro's default setup with the summary statistics framework disabled; *(ii)* enabling the scan detector from §4.1; and *(iii)* enabling the top-k script from §4.5; For each configuration, we measure CPU utilization per 1 sec trace interval. The trace is replayed using the *pseudo-realtime* mode [23] of Bro, which was created to facilitate the realistic playback of packet-traces.

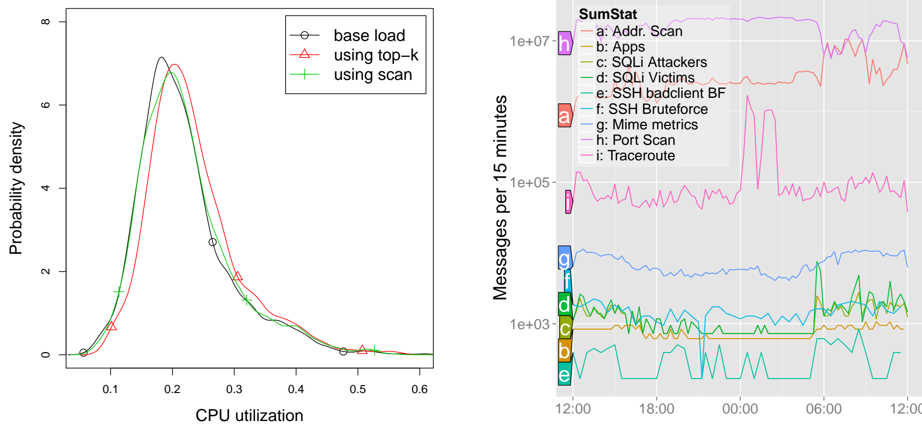

Fig. 6. Single node CPU load comparison. **Fig. 7.** Exchanged messages per sumstat.

Figure 6 shows a corresponding probability density for the three configurations.[9] We see that while using the summary statistics framework imposes overhead, it remains small for scan detection (0.4 percentage points more). The difference with the top-k script (1.6 percentage points more) is more noticeable due to the increased cost per observation that the more expensive maintenance of the probabilistic data structure entails. In either case, we deem the overhead low, relative to the input volume.

5.3 Memory Overhead

We next analyze the memory overhead introduced by the summary statistics framework. For this we follow the same approach as for CPU, measuring memory

[8] In other words, we assess the performance overhead for one worker node. We do not examine the CPU overhead of the manager node merging the data structures as that system is typically not CPU-bound and has sufficient head-room for additional operations.

[9] The measurement was done in a single-system Bro setup. However, we repeated it in cluster setup with a separate manager process, with similar results.

usage while running Bro repeatedly on the same input trace with the same three configurations. In all cases, we find the memory overhead imposed by the summary statistics framework reasonable. Even there the mean overhead is only about 6.7% (max. 179MB) in comparison to the baseline of a standard Bro setup.

5.4 Communication Overhead

Finally, we examine the communication overhead the summary statistics framework incurs in cluster operation. We add a script to the Bro manager node that logs all incoming and outgoing messages triggered by the summary statistics framework. For each message we output its timestamp and further meta-information for identifying its origin (e.g., the name of the reducer and the exact type of the message). We ran this measurement live for 24 hours on a 57-node Bro cluster of a medium-sized research organization that we have access to. The cluster monitors uplink traffic averaging at about 1 Gb/s during day-time hours. The setup used the full set of standard summary statistic scripts that come with a standard Bro installation, including detecting scans, traceroutes, SQL injection attacks, and SSH bruteforcing; as well as using two custom scripts to measure MIME statistics and traffic volume to several large sites (Google, Facebook, etc.).

Figure 7 shows a breakdown of the different summary statistics and the message overhead each caused. We find the scan detector responsible for most of the exchanged messages, due to the large number of incoming connections that it needs to classify. In total, the nodes exchanged 1,930,564,662 messages, with about half of them going from the manager to the worker nodes. This is due to the manager always initiating the exchange of values (i.e., even after a worker's notification, it is the manager that then polls for updates). This means that each node sends about 399.03 messages per second each way. Messages relating to the intermediary updates constitute 0.40% of the overall communication. 69,810 times a worker node notifies the manager that it should request updates. In 27,704, or 39.68%, of these cases, the manager chooses to ignore that request (an optimization that our implementation applies to limit simultaneously outstanding key updates for the case where a set of keys triggers many notifications in short succession; by default, the framework limits the number of simultaneously running updates to 10 per summary statistic). In 15.98% of the cases that the request is accepted by the manager, the target threshold has indeed been crossed, and hence the manager alarms after aggregating the individual values.

Overall, we deem the level of communication realistic for such large-scale, high-volume settings; and clearly within what Bro's communication system is able to handle [23]. This conclusion is supported by the Indiana University setup, which is running the scan detector in operations (§4.1). We note that scan detection represents pretty much the worst case for a distributed monitoring setup as one needs to continuously correlate activity about many addresses across all nodes in a timely manner. While we have not yet performed a more systematic sensitivity analysis, we expect that we could further reduce the messages exchanged by tuning the specifics of the update mechanism.

6 Related Work

Our design and implementation represent a generic framework that supports a wide spectrum of network-based summary statistics. We are not aware of any system that provides similar flexibility with an easy-to-use interface, suitable for real-time processing in distributed deployments.

Summary statistics are widely used throughout the networking and security communities, both in research and operations. To give just a few examples of research efforts presenting applications and/or corresponding data structures, the literature includes work on finding port scanners in backbones [24], efficiently counting the number of network flows in high-speed environments [16,9], detecting attacks against routers [1], computing real-time traffic summaries [15], or identifying elephant flows [8]. However, all of these efforts remain specific to their particular target application, while our work provides a framework on top of which one can implement such analyses.

In operations, appliances from companies like SonicWall and Palo Alto Networks compute traffic summaries and break-downs, however they hardwire the analysis performed. Several open-source utilities can apply statistical computations to network traffic, in particular NetFlow-based toolsets like *SILK* [22] and *flow-tools* [11]. However, they remain restricted to the abstractions their input format provides, are intended mainly for offline/batch usage, and do not provide the flexibility of performing arbitrary computations. Splunk can compute top-k-style statistics flexibly on different features, yet its input remains limited to externally produced log files.

For intrusion detection, Denning pioneered statistical monitoring in her seminal work on the host-based IDES system [7]. Today, scan detectors come with virtually any IDS, including open-source systems such as Snort [21]. Older versions of Bro [19] used to come with four fully separate scan detector implementations, all targeting different traffic features and/or threshold schemes. Our summary statistics framework supports all four directly within its unified API. We refer to, e.g., [18,12] for a broader overview of statistical anomaly detection (as well as other approaches). We note that while we limit our summary statistics framework implementation to threshold-based schemes for now, conceptually it could support further statistical approaches as well.

Cohen et al. [4] present an abstract framework for weighted sampling in distributed settings. It is similar in intent to our work, however, it only considers the case of sampling, and evaluates optimal algorithms for this setting. Peng et al. [20] uses a cumulative sum algorithm to collect statistics at nodes and share information using a machine learning algorithm. In contrast to our work, their usage scenario is limited to cumulative sums and their evaluation focuses on optimizing detection delays and bandwidth, not on providing a generally usable framework for distributed summary statistics.

We use a set of probabilistic data structures to efficiently compute statistics that traditionally would be very resource intensive to maintain on large inputs. We choose data structures that satisfy our constraints (see §2.4), yet note that there are further candidates. For example, there are extensions available for the

HyperLogLog algorithm that we use [10]: Kane et al. [14] propose an algorithm with an even lower memory overhead; it however remains complex and seems impractical to implement [13]. Heule et al. likewise propose a series of improvements to HyperLogLog [13]. As our main contributions concerns the framework itself—not individual computations—we do not further explore such alternatives, though may do so in the future if the current implementation ever turned out to represent a bottleneck.

7 Conclusion

In this work, we present the design and implementation of a novel *summary statistics framework* for network monitoring. As one of its key features, the framework supports computing statistics on arbitrary keys, such as IP addresses, DNS labels, or HTTP server names. Furthermore, our design specifically targets distributed deployment, and can thus be used in environments where sensors are either scattered over independent tapping points, or jointly process a high-volume link in a load-balancing setup. We assess the feasibility of our approach by implementing the summary statistics framework on top of the open-source Bro network monitor, and showcase a set of example applications in realistic large-scale settings.

Overall, we consider the summary statistics framework an extensible platform that enables research and operators to measure and quantify characteristics of their network traffic, with much less effort than they would traditionally require in particular in the distributed setup. Using the summary statistics framework, users can implement powerful statistical measurements in just a handful lines of code, and immediately deploy them for real-time processing.

Acknowledgments

We would like to thank for their collaboration Keith Lehigh and Indiana University; Aashish Sharma and the Lawrence Berkeley National Laboratory; Justin Azoff and the National Center for Supercomputing Applications at the University of Illinois; as well as further unnamed organisations that have operated early versions of the framework. This work was supported by the US National Science Foundation under grants OCI-1032889 and ACI-1348077; by the U.S. Army Research Laboratory and the U.S. Army Research Office under MURI grant No. W911NF-09-1-0553; and by a fellowship within the Postdoc-Programme of the German Academic Exchange Service (DAAD). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators, and do not necessarily reflect the views of the NSF, the ARL/ARO, or the DAAD, respectively.

References

1. Barman, D., Satapathy, P., Ciardo, G.: Detecting Attacks in Routers using Sketches. In: Workshop on High Performance Switching and Routing. HPSR (2007)

2. Bro SumStat Scripts & Repos. *http://www.icir.org/johanna/sumstats*
3. Bro Network Security Monitor Web Site. *http://www.bro.org*
4. Cohen, E., Duffield, N., Kaplan, H., Lund, C., Thorup, M.: Composable, Scalable, and Accurate Weight Summarization of Unaggregated Data Sets. Proc. VLDB Endow. 2(1) (Aug 2009)
5. Das, S., Antony, S., Agrawal, D., El Abbadi, A.: Thread Cooperation in Multicore Architectures for Frequency Counting over Multiple Data Streams. Proc. VLDB Endow. 2(1) (Aug 2009)
6. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM 51(1) (Jan 2008)
7. Denning, D.E.: An Intrusion-Detection Model. IEEE TSE 13(2) (Feb 1987)
8. Estan, C., Varghese, G.: New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, ignoring the Mice. ACM Trans. Comput. Syst. 21(3) (Aug 2003)
9. Estan, C., Varghese, G., Fisk, M.: Bitmap Algorithms for Counting Active Flows on High-Speed Links. IEEE/ACM Trans. Netw. 14(5) (Oct 2006)
10. Flajolet, P., Éric Fusy, Gandouet, O., et al.: Hyperloglog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm. In: Proc. of the International Conference of Analysis of Algorithms. AFOA (2007)
11. Flow-tools information, *http://www.splintered.net/sw/flow-tools*
12. Garcia-Teodoro, P., Díaz-Verdejo, J.E., Maciá-Fernández, G., Vázquez, E.: Anomaly-Based Network Intrusion Detection: Techniques, Systems and Challenges. Computers & Security 28(1–2) (2009)
13. Heule, S., Nunkesser, M., Hall, A.: HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm. In: Proc. EDBT (2013)
14. Kane, D.M., Nelson, J., Woodruff, D.P.: An Optimal Algorithm for the Distinct Elements Problem. In: Proceedings ACM PODS (2010)
15. Keys, K., Moore, D., Estan, C.: A Robust System for Accurate Real-Time Summaries of Internet Traffic. In: Proc. SIGMETRICS (2005)
16. Kim, H.A., O'Hallaron, D.R.: Counting Network Flows in Real Time. In: Proc. IEEE Global Telecommunications Conference. vol. 7 (2003)
17. Metwally, A., Agrawal, D., El Abbadi, A.: Efficient Computation of Frequent and Top-k Elements in Data Streams. In: Proc. ICDT (2005)
18. Patcha, A., Park, J.M.: An Overview of Anomaly Detection Techniques: Existing Solutions and Latest Technological Trends. Computer Networks 51(12) (2007)
19. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks 31(23-24) (1999)
20. Peng, T., Leckie, C., Ramamohanarao, K.: Information Sharing for Distributed Intrusion Detection Systems. Journal of Network and Computer Applications 30(3) (Aug 2007)
21. Roesch, M.: Snort: Lightweight Intrusion Detection for Networks. In: LISA (1999)
22. SILK – System for Internet-Level Knowledge, *http://tools.netsa.cert.org/silk/*
23. Sommer, R., Paxson, V.: Exploiting Independent State For Network Intrusion Detection. In: ACSAC (2005)
24. Sridharan, A., Ye, T.: Tracking Port Scanners on the IP Backbone. In: Proc. Workshop on Large Scale Attack Defense. LSAD (2007)
25. Vallentin, M., Sommer, R., Lee, J., Leres, C., Paxson, V., Tierney, B.: The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In: RAID (2007)
26. Vitter, J.S.: Random Sampling with a Reservoir. ACM TOMS 11(1) (Mar 1985)