Jaal: Towards Network Intrusion Detection at ISP Scale

Azeem Aqil*, Karim Khalil*, Ahmed O.F. Atya*, Evangelos E. Papalexakis*,

Srikanth V. Krishnamurthy*, Trent Jaeger†, K.K. Ramakrishnan *,

Paul Yu‡ and, Ananthram Swami‡

*UC Riverside, †The Pennsylvania State University, ‡U.S. Army Research Laboratory

{aaqil001, karimk, afath001, epapalex, krish, kk}@cs.ucr.edu, tjaeger@cse.psu.edu, {paul.l.yu.civ, Ananthram.swami.civ}@mail.mil

ABSTRACT

We have recently seen an increasing number of attacks that are distributed, and span an entire wide area network (WAN). Today, typically, intrusion detection systems (IDSs) are deployed at enterprise scale and cannot handle attacks that cover a WAN. Moreover, such IDSs are implemented at a single entity that expects to look at all packets to determine an intrusion. Transferring copies of raw packets to centralized engines for analysis in a WAN can significantly impact both network performance and detection accuracy. In this paper, we propose *Jaal*, a framework for achieving accurate network intrusion detection at scale. The key idea in *Jaal* is to monitor traffic and construct in-network *packet summaries*. The summaries are then processed centrally to detect attacks with high accuracy. The main challenges that we address are (a) creating summaries that are concise, but sufficient to draw highly accurate inferences and (b) transforming traditional IDS rules to handle summaries instead of raw packets. We implement *Jaal* on a large scale SDN testbed. We show that on average *Jaal* yields a detection accuracy of about 98%, which is the highest reported for ISP scale network intrusion detection. At the same time, the overhead associated with transferring summaries to the central inference engine is only about 35% of what is consumed if raw packets are transferred.

CCS CONCEPTS

• **Security and privacy** → *Intrusion detection systems*; *Network security*;

ACM Reference format:

Azeem Aqil*, Karim Khalil*, Ahmed O.F. Atya*, Evangelos E. Papalexakis*, Srikanth V. Krishnamurthy*, Trent Jaeger†, K.K. Ramakrishnan *, Paul Yu‡ and, Ananthram Swami‡ *UC Riverside, †The Pennsylvania State University, ‡U.S. Army Research Laboratory {aaqil001, karimk, afath001, epapalex, krish, kk}@cs.ucr.edu, tjaeger@cse.psu.edu, {paul.l.yu.civ, Ananthram.swami.civ}@mail.mil . 2017. Jaal: Towards Network Intrusion Detection at ISP Scale. In *Proceedings of CoNEXT '17, Incheon, Korea, December 12–15, 2017,* 13 pages.

https://doi.org/10.1145/3143361.3143399

CoNEXT '17, December 12–15, 2017, Incheon, Korea

ACM ISBN 978-1-4503-5422-6/17/12…$15.00

https://doi.org/10.1145/3143361.3143399

1 INTRODUCTION

Incorporating cybersecurity capabilities has become integral to networked system design today. However, there are continuing challenges having to deal with very large scale and complex attacks, and the ever-changing nature of attacks. In the recent past, we have seen an alarming increase in network based high-profile security breaches [4, 8, 33].

Today, ISPs predominantly focus on volumetric (e.g., DDoS) attacks by gathering traffic at ingress gateways and analyzing it at centralized scrubbers [3]. These services are delivered to an enterprise by an ISP on demand, i.e., a customer requests the services when it determines that its network is under attack or suspects an attack. The process is to copy packets to and from the enterprise at gateways, and forwarding these to a central Network Intrusion Detection System (NIDS). Unfortunately, the approach of transferring copies of raw packets continuously towards detecting attacks suffers from scalability problems.

In this paper, our goal is to design an efficient ISP-scale NIDS, that has the following properties: (a) it should detect a wide variety of attacks as with smaller scale IDSs such as Snort or Bro and, (b) it should not require the copy and transfer of raw packets to a central inference engine.

The key design principle that we follow is to "extract" the requisite information from a packet stream at monitoring points spread through out the network. Specifically, the monitors, (could be co-located with routers/gateways or placed at IXPs) process packets and create lightweight in-network *packet summaries* of drastically smaller volume compared to raw packets. These summaries can be used to draw inferences using rules similar to those used in smaller scale NIDS (e.g., Snort). They are sent by monitors to a central inference engine which processes them and issues alerts when attacks are detected. In rare cases when a highly accurate inference cannot be made with the summaries, the inference engine queries appropriate monitors (which store packets for short periods) for either finer grained summaries or associated raw packet traces for a time window of interest.

In designing an ISP-scale NIDS based on the above principle, we will need to address the following challenges. (a) How do we construct lightweight packet summaries while still achieving high detection accuracy? (b) How do we transform typical NIDS rules (e.g., that of a system like Snort) to find attack patterns/signatures using these summaries in lieu of raw packets? and, (c) Under what conditions should the system retrieve finer-grained summaries or raw packet traces to ensure a high accuracy of detection while keeping the overhead low? In this paper we design and implement

an ISP-scale NIDS, *Jaal*, that addresses these challenges. Specifically, in designing *Jaal*, we make the following contributions:

- We design a novel algorithm that uses dimensionality reduction techniques to construct concise packet summaries that lend themselves to highly accurate network intrusion detection.
- We design an approach to transform a large set of IDS rules (specifically from Snort) to a new equivalent representation that can be applied to the generated summaries in lieu of the actual packets. We propose simple new, equivalent rules for those that cannot be automatically transformed.
- We implement *Jaal*, on a large scale SDN testbed that allows us to create complicated ISP-scale topologies, with approximately 370 routers, using network function virtualization (NFV).
- We evaluate the performance of *Jaal* using realistic ISP traces [15] and with a wide range of popular attacks. Our results show that with about 65% reduction in communication overhead, *Jaal* can achieve a detection accuracy of ≈98% with respect to these attacks which, we believe is the highest reported at these scales.

2 SYNOPSIS

There are many recent alarming reports of large-scale distributed attacks targeting multiple data centers or enterprise networks simultaneously [5, 7, 12]. Unfortunately, today such attacks are only detected much after the fact, and the targeted entities report them individually.

The Mirai botnet example: To exemplify the problem, consider the example of the DDoS attack caused by the Mirai botnet[1], which targeted Dyn's DNS infrastructure [5]. The attackers used compromised IoT devices (e.g., printers, cameras and home routers) spread across the Internet to launch one of the largest known DDoS attacks crippling various services across the Internet including Twitter, Airbnb, Github and Amazon, among others. In brief, a post-attack analysis revealed that the IoT devices were infected using a simple two-step process [10, 11]. First, such devices were discovered by continuously scanning IP addresses across the Internet for open ports. Second, once an open port was found the associated device was compromised, if it was vulnerable, using a short list of hard coded passwords (most devices were still using default username/password combinations), to gain root access. We analyzed the source code for Mirai (available publicly at [6]) and found that the scanning was primarily directed at destination ports 23 and 2323 (*File*: mirai/bot/scanner.c *Lines:* 117, 219, 223 [6]). A device, once subsumed into the Mirai botnet repeated the exact scanning activity mentioned above [10]. Note here that this scanning was only discovered after the attack had been launched and researchers had analyzed the source code publicly dumped by the attackers.

Need for ISP Scale detection. Such scanning, while simple to carry out, is inherently difficult to detect using current detection capabilities for two reasons. First, most IoT devices are used in homes where consumers typically do not use IDS systems. So both the incoming scan that is looking for open ports and the outgoing scan launched by the device once it is infected, are missed. Second, and perhaps more importantly, even if we consider networks with some intrusion detection capability, the global scope of this scanning activity is only observable via a holistic view of a wide area (ISP-scale) network. This is because, from the perspective of a smaller scale (e.g., enterprise) network with detection capabilities such as those of Snort, a simple scan directed at two TCP ports for a small range of IP addresses (that are observed) is not a serious concern and would not likely trigger port scan alerts. Port scan alerts are only issued if a large volume of packets with a large set of different incoming ports are seen [22]; this was not the case with the Mirai scan which only scanned for two ports but across a large set of devices that were spread across a large number of administrative domains.

One could argue that a scan that targets the same two port numbers across an extremely large gamut of IP addresses (almost the entire IPv4 address range [10]) is concerning and should trigger alerts; however, this characteristic is only evident if the ISP-level traffic is analyzed at a detection engine. With such holistic visibility, a detection system could have identified infected devices long before the DDoS attack was launched (such scanning activity can be potentially detected in seconds as we show later in § 8).

The need for holistic ISP-scale detection capabilities have recently been widely advocated. For example, in response to the attack on Dyn, the security expert Bruce Schneier says that "DDoS prevention works best deep in the network, where the pipes are the largest and the *capability to identify and block the attacks is the most evident*" [9]. The fact that current detection frameworks and techniques lack the capabilities to detect coordinated attacks distributed across multiple organizations (such as the Mirai attack) has also been highlighted by DARPA [14], which exemplifies the urgent need for NIDS that address this shortcoming.

While ISP networks have the global visibility required to detect such attacks, the technical challenges in realizing WAN-scale detection are yet to be overcome. The popularity of open source IDS's like Snort and Bro has proven how effective pattern matching is in detecting network-based attacks. However, using such methods directly in WANs is hard (as discussed later). While enterprises typically have a single entry point where an IDS such as Snort or Bro can be employed, WANs usually have multiple points of entry and egress. This means that no single location in the network can view (monitor) all the traffic. Given that multiple vantage points (or monitors) are needed to completely cover all traffic, the information collected (or generated) by these monitors needs to be "aggregated" to create a global view for analysis.

Challenges: There are various approaches that one could take to create such a global view. The first and possibly the most obvious approach would be for each monitor[2] in the network to forward copies of traversing packets to a central analysis engine. We tested the feasibility of this approach (details in § 8; see Fig. 7) and found that it causes a 70% loss in throughput (average rate at which packets that belong to normal traffic are proceesed at each router) and a 75% loss in detection accuracy (the fraction of correctly classified attacks out of all attacks).

[1] Our characterization of Mirai is consistent with the initial dominant strain of the attack. For a detailed look at the nuances and the latter Mirai inspired attacks see [25].

[2] The monitors can be realized in several ways. They could be implemented using a core router functionality (like Cisco's NetFlow [37]). The monitors can also be dedicated machines deployed at IXPs.

We point out here that it is well known that modern open source DPI-based IDSs cannot cope well with high traffic volumes [32, 42]. In fact, they do not fail gracefully and with traffic rates greater than 20 Gbps, packet losses of over 50% are experienced regularly; this can seriously affect the accuracy in detecting attacks (can result in approximately a 50% loss in detection accuracy). The only way around the loss in accuracy is to provision IDS clusters for peak load which can result very high costs and even with that, a large wastage of computation resources at off-peak times.

Current methods such as sampling/sketching are inadequate. One may argue that the heavy workloads due to copying raw packets, can be overcome by using state of the art packet sampling techniques [30]. In fact, ISP's typically employ rudimentary sampling techniques like NetFlow [37] to obtain a coarse view of network dynamics. However, while sampling can help in heavy-hitter detection, it results in poor accuracy with respect to fine-grained features needed for detecting a wide variety of attacks [46, 49], especially in cases where a information with respect to a large number of successive packets is required (e.g., DDoS). In particular, sampling fails to capture correlations across packets.

An alternative technique to sampling is sketching [43]. While sketching provides strong resource/accuracy guarantees, it is inherently a targeted measurement scheme in which one needs to construct a sketch for every measurement task (e.g., counting the number of unique IP addresses or measuring the entropy of IP addresses in a batch of packets). Unlike the summarization techniques we develop in *Jaal*, modern sketching techniques are also restricted to single dimensional measurements [44]. This lack of generality implies that sketching does not scale. The sketching technique in [44], which is arguably the most general sketch possible today, allows a single sketch to be used for multiple measurement tasks. However, it is still limited to a single dimension.

More concretely, consider the source IP address as the dimension of interest. By using the technique in [44], one can create a sketch that captures the number of unique addresses seen. This sketch can be used to detect heavy hitters, membership testing and entropy estimation [44]. However, even this sketch will only be able to answer queries about the source IP address. Intrusion detection signatures require correlations across packet dimensions. A simple example is a distributed SYN flood attack. To keep track of the source IP and the SYN flag, one would need a sketch that tracks these two fields. However, such a sketch can now no longer be used to answer queries about the IP or SYN flag alone. There is no way to decouple those two header fields in the sketch. Thus, to create sketches to detect attacks based on any combination of TCP/IP headers (18 header fields), a need a total of 2^{18} individual count-min sketches at every monitor. Assuming each count-min sketch to be 500KB [44], this would translate to 128GB of information being transferred by each monitor to a central location per measurement epoch, leading to a prohibitively large communication cost. In addition, such a brute-force combinatorial sketching method would not leverage correlations between header fields which may not be known to us a-priori and would only emerge through the actual traffic generated. Our proposed summarization scheme is able to identify and leverage those correlations automatically, by virtue of low-rank approximation.

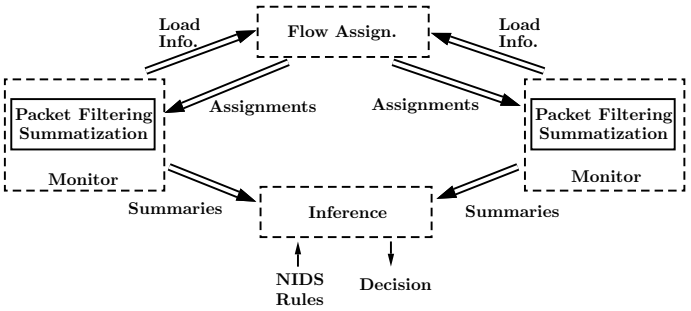


Figure 1: *Jaal* architecture.

Key idea: We envision that by extracting lightweight in-network "packet summaries" that retain the information needed by a NIDS, from traffic that flows through monitors, we can solve the challenge of scale while retaining performance and accuracy. This is in essence the key idea in building our framework *Jaal*.

Threat Model: While *Jaal* can handle all attacks that a NIDS like Snort can handle, we limit the scope of our evaluations to only transport layer attacks. This is because, together they constitute the most widely seen attacks in the wild [4]. We omit attacks that will require us to examine the payload given that these days payloads are often encrypted.

An IDS such as Snort also handles signatures relating to lower-layer attacks such as ARP scans. Since such attacks are local (link level and thus do not require global knowledge), we argue that they can be detected using a local "lower layer attack detector" in each local area network (LAN) independently. Thus, we do not try to detect these with *Jaal*.

We assume that monitors have already been placed in the network and their locations are static. Flexible monitor placement and management is beyond the scope of this work.

3 SYSTEM OVERVIEW

Our goal is to build a NIDS at ISP scale, with capabilities similar to that of today's modern NIDS (e.g., Snort [50] and Bro [48]) deployed in enterprises, based on the key idea described in § 2. Evidence collected in the network and transferred to a detection engine must consist of concise yet informative summaries (instead of raw packets), and the detection engine must be able to process these summaries and provide inferences just as with Snort. One of the driving principles behind designing Jaal was generality. We aim for the techniques we develop to be equally amenable to detecting all attacks. Therefore, we make no assumption on the utility of any packet header field and treat all header fields as equally important. We follow modern IDS systems design which provide pipelines for querying any header field. We choose Snort as our baseline since it is the most popular IDS used today [21].

We meet this goal by designing a modular ISP scale NIDS *Jaal*, which comprises three modules viz., summarization, inference, and flow assignment (see Fig. 1).

Summarization module: This module runs on monitors in the network. Each monitor extracts headers of traversing packets and constructs a "summary" of these headers using a lower dimensional

representation. The summary is then forwarded to the analysis and inference module.

Inference module: *Jaal* contains a centrally located inference module which, given a set of Snort-like, signature-based rules, transforms the rules into a format suitable for use with the summaries. Equivalent rules are proposed (again applied on summaries) for detecting attacks that cannot be detected via simple comparisons with packet signatures. Packet summaries collected from monitors are compared with these new rules to detect threats.

Flow assignment module: The flow assignment module seeks to assign flows to monitors such that each flow is monitored only once and the maximum load across the monitors is minimized. This is important both for correct operation of the NIDS as well as for saving bandwidth. The problem is mapped onto a constrained load balancing problem and solved using a simple yet effective approximation algorithm.

A detailed description of each module in *Jaal* follows in each of the next three sections.

4 PACKET SUMMARIZATION

The goal of packet summarization is to produce a representative summary of a batch of packets that: (a) enables the analysis and detection of a wide class of attacks, and (b) allows *Jaal* to achieve high detection accuracy with low communication overhead.

Henceforth, we refer to the packets and the packet fields as the different *modes* of the data that we are summarizing. The number of dimensions in those modes (i.e., the number of packets that have passed through a monitor and the number of fields in each packet) is large; thus our goal is to reduce the dimension of both modes while preserving correlations between the dimensions of both modalities.

To support our goal, we employ *dimensionality reduction*, a family of techniques that approximate a dataset with high-dimensional modes, using a modified dataset with significantly reduced dimensions, while minimizing the approximation error. This reduction results in a more compact, yet high fidelity data representation which respects correlations in both modes of the data. We propose a practical two-step approach where each step focuses on efficiently reducing the dimensionality of one of the two modes of the data. At the end of the process, we create what we call in-network packet summaries. One may envision a single-step approach to reduce both modes simultaneously; however this objective is computationally hard from an optimization point of view. Our approach can achieve any desired accuracy, by trading off communication cost. Specifically, the design parameters, determining the level of reduction, control the tradeoffs between the costs (i.e., the size of summaries sent to the inference module) and the detection accuracy.

4.1 Packet filtering and Normalization

We assume that there are monitors in the network to aid intrusion detection. Each such monitor filters and processes packets from flows (specified by a four tuple viz., source and destination IP addresses and port numbers) that are assigned to it (assignment of flows to monitors is done by a central engine; this is discussed in § 6). Transport and network layer headers are then buffered until the number of packets in the buffer, regardless of which flows they belong to, is equal to some pre-determined threshold n. We call this a batch of size n. Each batch of packet headers is organized in a matrix $\mathbf{X}$ with dimensions $n \times p$. Each row represents the p fields in the (TCP and IP) headers of a given packet.

Before we construct packet-summaries, we create a normalized version of $\mathbf{X}$ denoted by $\bar{\mathbf{X}}$. In *Jaal*, packet header fields are processed as vectors, and each header field is mapped to an entry in the vector. A measure of distance is used to decide whether two packets are similar. Since the raw magnitudes of the header fields values vary greatly, normalization of these values is needed to ensure that there is no bias towards fields that vary over a larger range. For example, consider a vector with only the TCP SYN flag and the source IP address fields. Without normalization, the distance computations will be heavily dominated by variations in the IP address field. Thus, for any header field with value $x \geq 0$, we apply the transformation $\bar{x} = \frac{x}{\max(x)}$, where $\max(x)$ is the maximum possible value for that header field (x). Thus, we have $0 \leq \bar{x} \leq 1, \forall x$.

4.2 Dimensionality reduction: fields mode

We first apply Singular Value Decomposition (SVD), a dimensionality reduction technique, on the packet fields in $\bar{\mathbf{X}}$ to reduce its rank. By reducing the rank, we form a smaller-size representation of $\bar{\mathbf{X}}$ which provably approximates the original matrix well [40]. First, $\bar{\mathbf{X}}$ is decomposed to

$$\bar{\mathbf{X}} = \mathbf{U}\Sigma\mathbf{V}^T, \tag{1}$$

where the columns of $\mathbf{U}$ are the left singular values of $\bar{\mathbf{X}}$, columns of $\mathbf{V}$ are the principle axes (or directions) of data in $\bar{\mathbf{X}}$, and $\mathbf{V}^T$ is the transpose of $\mathbf{V}$. Σ is a diagonal matrix with the singular values σ_i of $\mathbf{X}$ in descending order. The number of non-zero singular values is the rank of $\bar{\mathbf{X}}$, and the sum of squares of σ_i represents the amount of data variation in $\mathbf{X}$.

In practice, many data matrices exhibit a latent rank much lower than the actual rank measured by the number of non-zero singular values, or rather, by the number of linearly independent columns in the matrix. Intuitively, the reason is that, in practice, many columns (i.e., header fields of the packets we store in the matrix) are not exactly linearly dependent but they may be *highly correlated* and thus, approximately linearly dependent. When this happens, we can reduce the size of the data by approximating the matrix using its latent rank, which is smaller than the observed rank.

Thus, in order to reduce the size of data sent for analysis and inference, while minimizing the approximation error (in the least squares sense), a lower rank representation of $\bar{\mathbf{X}}$ is produced by keeping only the largest $r \leq p$ singular values σ_i of $\bar{\mathbf{X}}$ and setting the remaining ones to zero, to get

$$\bar{\mathbf{X}}_p = \mathbf{U}\Sigma_p\mathbf{V}^T. \tag{2}$$

It is provable that $\bar{\mathbf{X}}_p$ is the optimal rank-r approximation of $\bar{\mathbf{X}}$, in the sense of minimizing the Frobenius norm of $(\bar{\mathbf{X}} - \bar{\mathbf{X}}_p)$ [36]. Here, multiple σ_is may have small values, indicating that the latent rank of the matrix is smaller than the observed one. In that case, the dimensionality of $\bar{\mathbf{X}}$ can be reduced, while still retaining a high percentage of the information in the data by, for example, removing the smallest σ_is such that the sum of the squares of the retained singular values is at least 90% of the sum of the squares of all singular values.

In (2), all matrices have similar dimensionality as their counterparts in (1). However, since the last $p-r$ of singular values are set to zero, the corresponding columns in $\mathbf{U}$ and $\mathbf{V}$ can now be removed. This new representation is the truncated SVD representation of $\bar{\mathbf{X}}$ and is equivalent to $\bar{\mathbf{X}}_p$. It can be written as follows:

$$\bar{\mathbf{X}}_r = \mathbf{U}_r \Sigma_r \mathbf{V}_r^T, \tag{3}$$

where the dimensions of $\mathbf{U}_r, \Sigma_r, \mathbf{V}_r$ are now $(n \times r), (r \times r)$ and $(p \times r)$, respectively. In our system, the design parameters help decide which representation, i.e., $\bar{\mathbf{X}}_r$ or $\bar{\mathbf{X}}_p$, is more efficient as discussed at the end of § 4.3.

4.3 Dimensionality reduction: packets mode

Next, we seek to further reduce the size of the summary produced by each monitor via an even more compact representation of the packets in the reduced rank space. To do so, we seek to reduce the dimensionality *across* packets. We pose the problem to one similar to signal quantization, where the goal is to identify a set of representative values of a digitized signal which minimizes the approximation error. Thus, we seek to find a set of representative packets $\mathcal{R}$ that can approximate $\bar{\mathbf{X}}_p$ (or $\bar{\mathbf{X}}_r$). Ideally, $\mathcal{R}$ is constructed such that packets that are similar are mapped to the same representation. Let the size of the set of representative packets be $|\mathcal{R}| = k$. Formally, our problem can be posed as:

$$\min_{\mathbf{R},\mathbf{B} \in \{0,1\}^{n \times k} \bigcap \mathcal{RS}} \|\bar{\mathbf{X}}_r^T - \mathbf{R}\mathbf{B}^T\|_F^2, \tag{4}$$

where the $\mathcal{RS}$ constraint requires each row of $\mathbf{B}$ to sum up to 1, and the columns of matrix $\mathbf{R}$ are "centroids", effectively containing the packets $\mathcal{R}$, and matrix $\mathbf{B}$ is an assignment of each packet to a centroid. We use the Frobenius norm (i.e., Euclidean distance between a packet and its centroid) because (i) we have no prior knowledge about the distribution of packet vectors and, (ii) the problem admits very efficient approximations under this norm.

The above problem is known as *Vector Quantization* or *K-means clustering* and it is NP-hard in the above form [24]. However, there exist very efficient approximations when using the Frobenius norm as the loss function, with the most widely used being Lloyd's algorithm [45]. In *Jaal*, we employ the "k-means++ algorithm" [26], an improvement upon Lloyd's algorithm that seeks to find a good initialization for the algorithm to converge faster on a good local minimum. We use this clustering algorithm since it is guaranteed to find a solution that is $O(\log k)$-competitive to the optimal clustering solution, and is known for fast convergence.

Note that $\mathcal{R}$ has k packet representatives; each represents a group (i.e., cluster) of similar rows in $\bar{\mathbf{X}}_p$. Increasing k increases resolution (more representatives) but increases the communication cost as well. We study how varying k affects detection accuracy for different attacks in § 8.

We propose *two* methods for processing and sending summaries to the inference engine. In the first, we build a *combined summary*, by applying the clustering algorithm on $\bar{\mathbf{X}}_p$. The output consists of k centroids $\tilde{\mathbf{X}}_p$, one for each cluster, as well as clustering metadata. The latter contains a membership counts vector $\mathbf{c}$, and is appended to $\tilde{\mathbf{X}}_p$ to form $\mathbf{S}_1^m$, which is then sent to the analysis and inference module. It is easy to see that the number of elements of $\mathbf{S}_1^m$ is thus $k(p+1)$, for each update from monitor m.

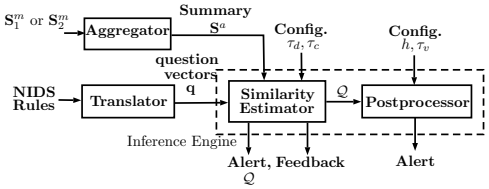


Figure 2: Inference Module.

In the second method, we create what we call a *split summary*. Here, we apply the clustering algorithm on $\mathbf{U}_r$. The output consists of the k centroids $\tilde{\mathbf{U}}_r$ as well as the corresponding metadata $\mathbf{c}$ and $\mathbf{e}$. The summary in this case is the collection $\mathbf{S}_2^m = \{\tilde{\mathbf{U}}_r, \Sigma_r \mathbf{V}_r^T, \mathbf{c}\}$. From (3), and since Σ_r is diagonal and can be sent as a vector of size r, the number of elements in $\mathbf{S}_2^m$ is $r(k+p+1)+k$.

Note that the information compiled in $\mathbf{S}_1^m$ is equivalent to that in $\mathbf{S}_2^m$, but is represented in a different format. More importantly, as discussed above the communication cost is different with each method depending on the design parameters r and k. In particular, when $r(k+p+1)+k < k(p+1)$, our system employs the split summary method and sends $\mathbf{S}_2^m$ to the analysis and inference engine; otherwise, $\mathbf{S}_1^m$ is sent.

5 ANALYSIS AND INFERENCE

Jaal includes a centralized inference module. While this notion of central inference is similar to that with Snort or Bro, the inferences are based on summaries instead of raw packets. In brief, the inference module contains a translator that converts traditional IDS rules (specifically Snort rules) to a format that can be used with summaries. The inputs to this translator are "aggregated summaries" that are single consolidated representations of the summaries received from all monitors; they characterize all flows traversing the network in a given period of time. Then, the consolidated summary is checked against each transformed rule to detect attacks. Fig. 2 shows the different blocks in the inference module.

5.1 Aggregating summaries

Summaries generated by the different monitors need to be aggregated to get a global view. There are two ways to fetch summaries from monitors. In the first, monitors periodically send summaries to the controller. In the second, when a monitor accumulates a batch size (n) of packets, it constructs and ships the summary of this batch to the controller. At this point, the controller requests every other monitor to send its summary. In response, all monitors except those with fewer than n_{min} packets, send their summaries. Summarizing information using less than n_{min} packets incurs accuracy penalties because clustering and SVD generally do not perform well when data dimensionality is small. However, as shown in § 8, n_{min} is very small and not of concern in high speed networks.

Let the number of available monitors in the network be M. The aggregated summary, $\mathbf{S}^a = \left[\tilde{\mathbf{X}}_a \mid \mathbf{c}_a\right]$, with centroids $\tilde{\mathbf{X}}_a$ and membership counts $\mathbf{c}_a$, is composed by concatenating the summaries collected from all monitors in a tall matrix format. In particular, when monitor m sends its summary in the form $\mathbf{S}_1^m$, it is appended

directly to S^a. When S_2^m is sent, the previously removed $p - r$ zero-vectors in $\tilde{U}_r, \Sigma_r$ and V_r are first restored, and the matrices are multiplied to reconstruct $\tilde{X}_p$. Then, the corresponding vector $\mathbf{c}$ is appended to $\tilde{X}_p$ to create the same form as S_1^m; this is appended to S^a.

The number of rows in S^a reflects the total number of representative packets collected from all the monitors, and is thus at most Mk. Our results in § 8 show that this simple aggregation method is enough to achieve high detection accuracy with respect to a wide range of attacks.

5.2 Inference in Jaal

For ease of deployment, we seek to automatically translate Snort rules (since it is the most popular NIDS in use) to handle packet summaries. Snort has two main modules for detecting attacks. First, it contains a signature matching module that matches every signature against every packet, using pattern matching algorithms [17]. In *Jaal*, we translate such rules automatically to handle summaries. Second, Snort contains attack-specific *preprocessors*, to detect attacks that cannot be handled using signatures (e.g., port scanning). In *Jaal*, we design a module called the "postprocessor" that has a functionality that is equivalent to that of Snort's preprocessor.

Translator: This block takes as input a packet signature g (i.e., a Snort rule), and automatically translates it into what we call a *question vector* $\mathbf{q}$ of length p as follows. The value of an entry in $\mathbf{q}$ is the normalized value of the corresponding header field in g, and -1 in the absence of a corresponding header field in g (i.e., the field is irrelevant to g). *Jaal* measures the similarity of the rules captured in a question vector $\mathbf{q}$ to packet representatives ($\tilde{\mathbf{X}}_a$) in the summaries S^a, using a simple distance measure as described later.

As an example, consider the translation of a specific Snort rule viz., a rule that relates to the SSH brute force attack [19]: *"alert tcp $EXTERNAL_NET any -> $HOME_NET 22 (msg: "INDICATORSCAN SSH brute force login attempt"; flow: to_server, established; content: "SSH-"; depth: 4; detection_filter: track by_src, count 5, seconds 60; metadata: service ssh; classtype: misc-activity; sid: 19559; rev:5;)."*

The rule postulates that an alert must be generated if 5 packets destined for the home network were received within the last 60s, with port number 22. To translate this rule into a question vector, *Jaal* initializes a vector of size 18 with -1 set for every position. Then, the position corresponding to the IP address is set to the normalized home network IP address and the position corresponding to port number is set to 22 (normalized version). This question vector is then used to make inferences as discussed below.

Similarity Estimator: Upon being provided with an aggregated summary, S^a, this block measures the similarity between each question $\mathbf{q}$ in the transformed rule set, and every $\mathbf{x} \in \tilde{\mathbf{X}}_a$ in the summary. The distance function used is:

$$d_{\mathbf{q}}(\mathbf{x}) = \frac{\sum_{j:q_j \neq -1} |q_j - x_j|}{\sum_{j:q_j \neq -1} 1}. \quad (5)$$

The denominator in Eq. 5 normalizes the distance measure to account for questions with different lengths. If the distance is below a threshold τ_d, a match is declared and hence an alert is raised for the corresponding threat signature.

Algorithm 1 Similarity Estimation

Input: question vector: $\mathbf{q}$, distance threshold: τ_d, centroids $\tilde{\mathbf{X}}_a$, counts $\mathbf{c}_a$, minimum count τ_c
Output: Binary Attack Classification

```
sum = 0, Q = {}
for x_i in X̃_a do
    if d_q(x_i) ≤ τ_d then
        sum ← sum + c_i
        Q ← Q ∪ x_i
    end if
end for
if sum ≥ τ_c then
    OUTPUT: Alert, Q
end if
```

Algorithm 2 Postprocessing

Input: Header field index h, variance threshold τ_v, centroids $\tilde{\mathbf{X}}_a$, counts $\mathbf{c}_a$
Output: Decentralized attack alert

```
Initialize Empty array Z
for x_i in X̃_a do
    add x_i(h) c_i times to Z
end for
if var(A) ≥ τ_v then
    OUTPUT: Alert
end if
```

For questions that require a minimum number of matches (e.g, SYN floods), the similarity estimator sums all the counts $c_i \in \mathbf{c}_a$ corresponding to $\mathbf{x}_i$ with $d_{\mathbf{q}}(\mathbf{x}_i) \leq \tau_d$, and only raises an alarm if this sum is larger than τ_c. Here τ_d and τ_c are per attack parameters to be configured by a system administrator (τ_c may be directly carried over from Snort). In addition, the packet representatives in $\tilde{\mathbf{X}}_a$ matching $\mathbf{q}$ are collected in Q.

In some cases when more accurate analysis is required at the expense of higher bandwidth cost, feedback is sent to monitors for a raw batch of packets. We discuss this in § 5.3.

Postprocessor: As mentioned earlier, Snort employs preprocessors to handle distributed attacks that cannot be handled using signature-matching. In *Jaal*, we craft rules equivalent to those in the preprocessor, that can be used with summaries. Note here that in typical Snort implementations, these correspond to a small subset of the total attacks handled [22].

In crafting these rules, we observe that a common feature of such attacks (handled by Snort's preprocessor) is that the variance (i.e., the spread in the range of values) in a specific packet header field is large. Consider, for example, port scans, which are characterized by a large number of incoming packets, all with different port numbers. A large variance in port numbers indicates that a large number of distinct port numbers was seen and thus, warrants an alert. Similarly, distributed attacks, such as DDoS, are characterized by a large number of different source IP addresses.

Jaal uses a "postprocessor" to handle attacks exhibiting large variance across a subset of header fields. This postprocessor first processes packet representatives Q, provided by the distance estimator block, that match a given signature $\mathbf{q}$. Next it applies Algorithm 2, to measure the variance in a header field of interest h. It issues an alert for an attack if the variance is larger than a predetermined threshold τ_v. This threshold is to be configured as with Snort [22].

Example: To illustrate how the inference engine operates, consider the example of a distributed SYN flood attack. The $\mathbf{q}$ corresponding to this attack will have the SYN flag entry set and all other fields set to -1. The similarity estimator, using Algorithm 1 will declare a SYN flood attack if the number of packets matching this

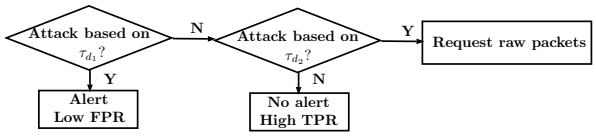


Figure 3: Feedback loop in *Jaal*.

signature is greater than the threshold required to issue an alert with regards to this attack (τ_c). It also outputs the list of packet representatives Q that match the signature $\mathbf{q}$. To classify whether this attack is distributed, the postprocessor applies Algorithm 2 to every element in Q, measuring the variance in the source IP header field. The attack is classified as distributed if the variance is above the predetermined threshold τ_v.

5.3 Trading cost for accuracy

Since *Jaal* uses packet summaries to infer patterns in packets, it is expected that the detection performance will be lower than that achieved if the raw packets were available. Despite this, *Jaal* achieves reasonably high accuracy with low communication overhead as shown in § 8. To improve the detection performance further (with additional communication costs), we design a feedback loop that enables *Jaal* to realize multiple operating points on the detection accuracy-communication overhead tradeoff. In brief, based on the output of the inference engine, *Jaal*'s controller sends explicit feedback to certain monitors (discussed below) requesting finer granularity summaries or even raw packets for specific batches.

With the feedback loop, the inference in *Jaal* is logically performed in two stages using two threshold values τ_{d_1} and τ_{d_2} as shown in Fig. 3. To explain the rationale for this, recall from § 5.2 that τ_d is used to determine whether a specific centroid in the summary matches a given question vector. A large value for τ_d will successfully catch most attacks (high true positive rate or TPR) but may also result in a high false positive rate (FPR). On the flip side, a small value of τ_d will result in low FPR, but also misses attacks more often. Thus, we choose the first threshold τ_{d_1} such that the FPR is small, and choose $\tau_{d_2} > \tau_{d_1}$ such that the miss-detection rate is even smaller than that with τ_{d_1}.

Let the binary result of the threshold-based analysis using τ_{d_1} and τ_{d_2} be t_1 and t_2, respectively. Thus, we have four different output cases. When t_1 is positive and t_2 is positive (case 1), the system has high confidence that there is an attack and an alert is raised. This because using τ_{d_1} results in low FPR. If t_1 is negative and t_2 is negative (case 2), no alert is raised. Here, the system relies on the high confidence with regards to the TPR when τ_{d_2} is used. If t_1 is negative and t_2 is positive (case 3), the controller asks the local monitors with the associated (uncertain) centroids in Q to send the actual packets corresponding to those centroids. The analysis is then done by pattern matching using traditional Snort rules and these raw packets. Requesting raw packets will decrease the overall FPR of the system, but will naturally increase the overhead. However, as we will show in § 8, this overhead is minimal and the feedback results in much improved detection accuracy. Finally, we note that the scenario where t_1 is positive while t_2 is negative (case 4) is unlikely (we never observed these in our experiments), since both analyses are done using the same summary S^a and thus it is expected that what is not missed in t_1 will also not be missed in t_2 (as t_2 guarantees higher TPR).

6 FLOW ASSIGNMENT

The flow assignment module seeks to assign flows to active monitors. In doing so, we have multiple goals. First, all flows passing through at least one monitor must be covered. Second, we require that a flow must be monitored by exactly one monitor. Duplicate monitoring of flows incurs unnecessary processing and bandwidth costs and more importantly, might lead to incorrect detection results. This is because *Jaal* uses summaries which do not retain information that could allow accounting for duplicate packet counts (same packet from multiple monitors) during inference. Third, in order to ensure that no monitor gets overloaded, we seek to ensure that the traffic monitored by the different monitors is balanced, to the extent possible. Finally, flow assignment has to be very efficient and scalable (algorithm must be of low complexity).

This problem is challenging due to multiple reasons. First, each flow may only traverse a specific subset of monitors. Second, a flow can last for an unknown amount of time before it terminates. Moreover, flows can vary drastically in terms of "packet rate," i.e., the rate at which packets belonging to that flow are seen at an assigned monitor. This packet rate is used as a weight to represent the relative workload generated by the flow, in the assignment problem. Since a priori forecasts of flow arrival times, termination times, or weights is not possible, the system has to make its flow assignment decisions that satisfy the objectives above, in *real time*.

The flow assignment problem can be mapped to the *online optimization problem* [23], where the flows are the jobs to be assigned to M machines (monitors) upon their arrival, such that the maximum load across all monitors is minimized (i.e., load balancing). Upon the assignment of a flow to a monitor, the load on the monitor is increased by an amount equal to the weight of the flow. This increase is valid for the duration of the flow. The assignment will have to be non-preemptive, since it is impractical to reassign the other flows when a new flow arrives. We assume that monitors are homogeneous.

The metric used to evaluate online algorithms is the *competitive ratio* [29], the supremum, over all possible input sequences, of the maximum (over time and over monitors) load achieved by the on-line algorithm to the maximum load achieved by the optimal offline algorithm. One online algorithm performs better than another if it has a lower competitive ratio. For load balancing problems, the performance here is measured in terms of the maximum load. Robin-Hood algorithm [28] has been shown to be optimal in solving online load-balancing of unknown duration tasks with assignment restrictions. In particular, it achieves a competitive ratio of $O(\sqrt{M})$, which is the lower bound for this class of problems. However, applying the Robin-Hood algorithm in practice is challenging because it requires the knowledge of incoming flow weights before an assignment decision is made. This is hard to do since the packet rates of a flow are not known a priori; estimates could be made (e.g., using machine learning) but it adds to the complexity. We omit the details of this algorithm in the interest of space (details are found in [28]).

Given the above challenges, we choose a simple greedy algorithm, which has been shown to achieve a competitive ratio of $\frac{(3M)^{2/3}}{2}(1 + o(1))$ [27]. The greedy flow assignment algorithm assigns an incoming flow f to the least loaded monitor within the subset of monitors on its path. This simple algorithm has two advantages. First, it does not require the estimation of flow weights to decide on the monitor to which the incoming flow is assigned. Second, as shown later in § 8, the assignment update is processed very quickly and thus, can potentially scale to WANs.

Since the number of flows traversing an ISP network is typically very large, making assignment decisions on per-flow basis and as new flows arrive and terminate, is not practical. Instead, we observe that subsets of flows, based on routing, can be grouped together. We call these constructs *flow groups*. In particular, a flow group is a set of flows that traverse a given common set of monitors. For a given flow group, we define the corresponding *monitor group* as the subset of monitors on the path of the flow group. Note that a monitor can belong to multiple monitor groups. In *Jaal*, a new flow is greedily assigned to the least loaded monitor in its corresponding monitor group. This setup allows us to implement a very efficient assignment algorithm that collects monitor load updates periodically, and then compute the assignment. Since flow arrivals and terminations events happen at arbitrary times, the performance of our greedy algorithm will approach the above theoretical bound of the greedy algorithm as the periodic update period P becomes smaller. In practice, we show in § 8 that it is comparable to that of Robin Hood algorithm.

7 IMPLEMENTATION

Central Components: The flow assignment, and the inference modules, are implemented at the central SDN controller using the open source Ryu SDN framework [18]. We employ shortest path routing; thus, flow groups are just based on common source and destination prefixes.

As discussed in § 6, each incoming flow is assigned (based on load updates) to the least loaded monitor. This is done via an OpenFlow forwarding rule installed in the switch attached to the monitor. The entire module is written using Python's event-driven framework as a single threaded application. The module maintains a dedicated long lived TCP connection with each monitor. The flow assignment module polls monitors for load updates every $P = 2$ seconds (a larger value resulted in poor load balancing and a smaller value did not yield any significant improvements).

In the inference module, the new IDS rules are created offline and stored. The module executes a single threaded process that waits in an event loop for monitors to send summaries. It also maintains a long-lived TCP connection with each monitor and periodically asks monitors for summaries. Once the summaries are received, they are checked against every locally stored question vector, and processed using the postprocessor, as described in § 5 and alerts if any, are logged.

Monitors: Monitors are implemented as network functions (NFs) at the SDN switches in Python, along with popular math and data mining libraries (NumPy, SciPy, pandas). They are instantiated by activating pre-stored VM images and attaching them to the chosen switches via a VLAN. The process is automated using an Ansible script. Each monitor executes two processes. The first process tracks load and responds to load queries when prompted. The second is responsible for the summarization tasks. Specifically, each monitor stores packets in a local buffer (NumPy array) and computes summaries. The centroids and their memberships are stored for one epoch (the periodicity with which the inference engine requests summaries) as a newly created hash table where the key is the centroid and the value is a list of actual packets associated with those centroids. If the monitor receives a request for raw packet dumps belonging to a specific centroid, it retrieves the list of packets using that centroid as the key and sends these to the inference engine. A hash table that is thus created, is deleted after the relevant epoch (2 seconds).

	Reservoir Sampling	*Jaal*
Distributed Syn Flood	54%	99%
Sock Stress	60%	98%
SSH Brute Force	42%	97%
Sockstress	56%	94%

Table 1: Comparing to reservoir sampling

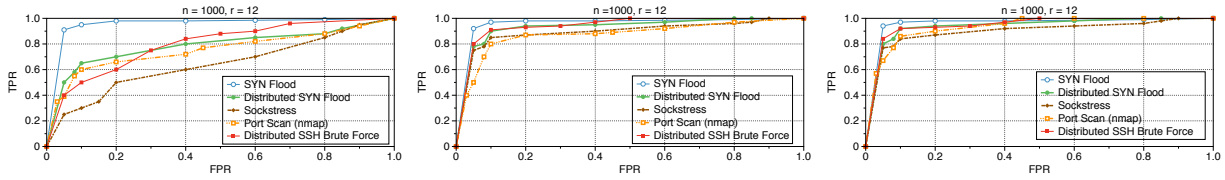
8 EVALUATION

Next, we present our evaluation of *Jaal*. We use two ISP backbone traces from the MAWI group [15], Trace 1(2016/01) and Trace 2 (2016/02). to represent background traffic. We inject attack traffic in conjunction. Specifically, we consider five different kinds of attacks: (i) SYN floods to represent DoS attacks, (ii) distributed SYN floods to represent DDoS, (iii) distributed port scans, (iv) distributed SSH brute forcing, and (v) Sockstress. DDoS, port scans, and brute forcing attacks are among the most common types of network-level attacks today [13]. Hence, we choose one of each type. We choose the Sockstress attack [2, 20] because it is more complex than other DoS attacks. It completes the TCP handshake and sets the TCP window size to 0, forcing the server to keep the connection alive for a long time. Finally, we also do a case study with the Mirai attack, and show *Jaal*'s effectiveness in countering it.

We now describe how we construct the different attacks. We first note that, while the MAWI traces might contain some malicious packets, it is very difficult to analyze the traces to determine which packets are anomalous and which are not as they are not labeled. In addition, running these traces through *Jaal* does not provide any intelligent information because the ground truth is not available. Thus, we treat the traces as benign traffic and explicitly inject malicious packets.

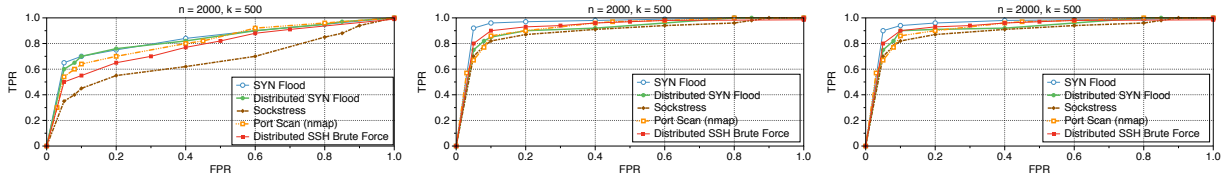
For each attack, we throttle the injected attack traffic to be at most 10% of the overall traffic. We get the volume of benign traffic by parsing the MAWI traces and pre-determining the total volume of traffic that will be replayed at each time instant. The attack scripts then enforces the cap by stopping attack packets if the 10% quota has already been met. Note that socktress is a stealthy DoS attack and as such, does not need a large number of packets to succeed. Consequently, we did not have to enforce the 10% rule for it.

For distributed attacks we generate source IP addresses randomly from different subnets. This ensures packets take different routes and traverse different monitors. The total number of attacking IP addresses for each attack is approximately 200. For port scans, we



(a) k = 100 (b) k = 200 (c) k = 400

Figure 4: ROC curves for various attacks. Batch size = 1000, rank = 12, varying k, Trace 1.



(a) r = 10 (b) r = 12 (c) r = 15

Figure 5: ROC curves for various attacks. Batch size = 2000, k = 500, varying rank, Trace 1.

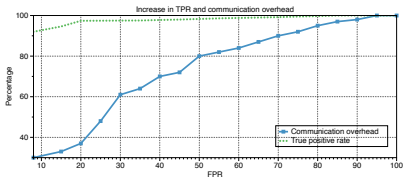


Figure 6: The increase in TPR and communication overhead as the acceptable FPR is increased.

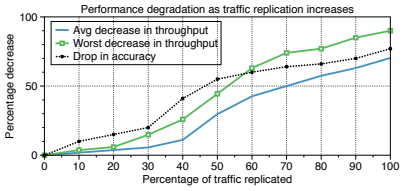


Figure 7: Performance degradation as the percentage of traffic that is replicated increases.

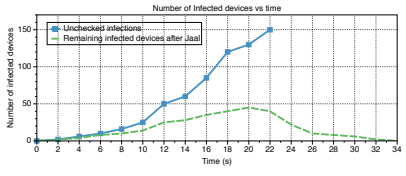


Figure 8: Mirai Attack: Unchecked infections versus infected devices shut off upon detected by Jaal.

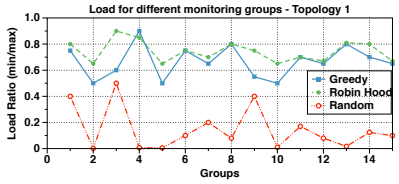


Figure 9: The load across different monitor groups for topology 1.

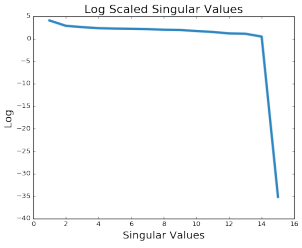


Figure 10: The magnitude of the singular values for a packet matrix of $n = 1000$.

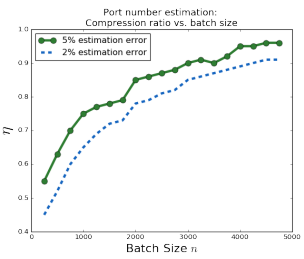


Figure 11: Percentage savings vs. the batch size for fixed error rates.

use the popular Nmap tool [16] to conduct scans. Nmap has a list of ports that it scans and we simply use those defaults.

Testbed and configuration: We implement *Jaal* on an in house SDN testbed that consists of 5 Dell PowerEdge 730 Servers (22 cores, 256GB memory, 12 ports) and two HP 3800 series switches (SDN enabled each with 48 ports) and 2 Arista 7280E switches (each with 72 ports). Note that we use SDN for the purposes of evaluating *Jaal*; it is not a necessary platform for the deployment of *Jaal*.

To represent the network, we use two realistic RocketFuel topologies [55] in our evaluations viz., Abovenet (which we call topology 1) and Exodus (which we call topology 2). Topology 1 has 367 routers and topology 2 has 338 routers. We create the topologies by instantiating the desired number of open vSwitch instances and connecting them using virtual links to match the configuration of the chosen topology. Our virtual topology can handle Gigabit traffic effectively, where the underlying SDN switches have 10GBE ports.

8.1 Detection Accuracy and Overhead

Overall results: In a nutshell, our experiments show that for all the attacks considered, *Jaal* achieves an average true positive rate (TPR)

of ≈98% at about 9% false positive rate (FPR) with a communication cost of only 35% of that incurred by conventional IDS systems (i.e., copying raw packet headers and sending to a central engine for analysis). To achieve these, we need to select parameters that dictate the granularity of summarization, so as to yield good tradeoffs between detection accuracy and cost. We use ROC (Receiver Operating Characteristic) curves towards choosing the appropriate configuration parameters (as discussed next).

ROC based analysis: Consider the case where $\tau_{d_1} = \tau_{d_2} = \tau_d$, i.e., no feedback loop is implemented. Two parameters influence *Jaal*'s tradeoffs between detection accuracy and communication cost, viz., the rank r, and the number of centroids k. We examine how accuracy changes for different values of r and k using ROC curves. ROC curves show the TPR versus the FPR and are typically used for understanding the tradeoff between the two metrics. Each combination of threshold values (i.e., τ_d, τ_c, τ_v) is a single point on the graph, and the TPR/FPR values are computed relative to ground truth. Each point is an average over 15 runs, each of which is 45 minutes long. Our experiments show that setting n to 20% of the size of the batch provides good resource consumption-accuracy tradeoffs. However, this might need to be re-evaluated by system admins in a real deployment. The process would be the same as the one we undertake above.

Varying number of centroids k: First, we fix $r = 12$, $n = 1000$, $n_{min} = 600$ and vary k and consider topology 1. In Fig. 4 the detection accuracy is shown for Trace 1 (similar results are observed for Trace 2 and are omitted in the interest of space). We observe that a value of $k = 200$ (i.e., 20% of the reduced dimension packets are sent for analysis and inference) yields very good detection accuracy for all attacks. Increasing k further yields diminishing returns in accuracy while increasing the communication costs. However, lowering k to 100 results in a significant detection penalty for all attacks except for SYN flood attacks. This is because of the boolean nature of flags. In other words, a packet is either as close as it can possibly be to a representative packet (centroid) with the SYN flag set, or is far away, depending on its SYN flag value.

Varying retained rank r: The rank of a matrix of raw packet headers, $\mathbf{X}$, is $\leq p = 18$. We study the distribution of the singular values of $\mathbf{X}$ and conclude that it is possible to retain 90% of the information in $\mathbf{X}$ by retaining only the top 16 values. To reduce the rank, we consider values for $r < 16$, as shown in Fig. 5 (with Trace 1 and topology 1). As apparent in Figs. 5b and 5c, $r = 12$ yields approximately similar performance to $r = 15$. Dropping r to 10, however, results in a high accuracy penalty for all attacks, as shown in Fig. 5a. We observe the similar results with Trace 2.

The Feedback Loop: From the ROC curves, we see that without the feedback loop, with properly chosen values of k and r, *Jaal* achieves an average TPR of ≈92% at about a 10% FPR, with a communication cost of only 30% of that incurred with raw packets. At the same time, if the TPR is chosen to be 98%, the FPR increases to 20%. We choose *attack specific thresholds* $\tau_{d_1} \neq \tau_{d_2}$, that yield these TPR and FPRs.

In Fig. 6, we plot the communication overhead vs. the TPR with the feedback loop. We also plot the average TPR for easy comparison. We see that with a value of τ_{d_2} chosen such that the TPR improves to 98%, the communication overhead (with the feedback loop implemented) only increases to 35% (from 30%) of that incurred with raw packets. Beyond this, the gain in TPR diminishes while the communication overhead rises sharply. Finally, we point out that with the feedback loop, the FPR is 9.1%, while the TPR is 98%. The reason for not seeing a further increase in TPR (the FPR drops) is that, since the traffic is already classified as attack traffic and only the data relevant to reducing false positives is retrieved, an increase in TPR is not observed.

Communication overhead: The communication overhead savings achieved by *Jaal* compared to sending raw packet headers as in conventional NIDS, is roughly proportional to k/n (if we ignore the raw packet overheads from feedback). After processing 2GB of traffic (in terms of headers only) system wide, *Jaal* only transmitted a total of ≈700MB. This corresponds to a 65% reduction compared to a traditional NIDS that requires sending all the 2GB. This is comparable to other specialized count-based sketches [44] but provides significantly richer information.

Feasibility of the vanilla approach of sending raw packets for inference: To test the feasibility of just copying and forwarding raw packets instead of generating summaries, we set up a realistic ISP topology and replayed backbone traces mixed with attack traffic (identical setup to the experiments above). The monitors (selected as previously) copy packets and transfer them to the central inference engine. We randomly vary the selection of the location of the inference engine across the experiments (a total of 25) to simulate different scenarios. The analysis engine runs Snort for the purposes of detection and we consider all the 5 attacks discussed earlier. Fig. 7 depicts the results of this experiment. We plot the percentage decrease in (a) the throughput and (b) accuracy on the Y-axis. The X-axis represents the percentage of traffic that is replicated for analysis purposes (fraction of packets that are copied and sent).

The network throughput reflects the average rate at which, normal traffic is processed at each switch (this takes a hit when it processes the copied traffic). The accuracy is measured as the percentage of attacks that the detection engine flags from among all the attacks injected. The throughput overhead reflects the "loss in throughput" (compared to the baseline case without replication – this is what is in existence today) due to the introduction of the extra (copied) traffic. The results show that in the worst case (for one particular choice of the central analysis engine), the throughput overhead is 90%; in the average case the throughut overhead is 70%. Such high throughput hits will significantly affect network performance for the ISP (and the users subscribing to that provider). With *Jaal*, the replication level roughly corresponds to 35% which in turn corresponds to an average loss in throughput of less than 10% and a worst case hit of <20%.

We also see a 75% decrease in accuracy when all raw packets are sent to the inference engine. This loss is a direct artifact of missing attacks because of packet losses (arising both due to congestion and overloading of the inference engine). The loss in accuracy shown does not directly reflect *Jaal*'s accuracy – it corresponds to the loss in accuracy due to sampling (since 35% of the packets are replicated and transferred); as shown earlier *Jaal* does dramatically better in terms of accuracy.

Computation Costs: Our tests indicate that each monitor can handle rates of 300Mbps easily, indicating that the combination of SVD and k-means is not compute intensive. We omit detailed results because of space.

Case study of the Mirai attack: To evaluate *Jaal*'s efficacy in countering the Mirai attack, we emulate the attack on our testbed (using the published source code [6]). A randomly chosen node in the network initiates the Mirai scan and infects vulnerable devices. An infected device starts scanning for other devices and in turn infects them if possible. We randomly selected 150 nodes in our network to be vulnerable.

We compare two scenarios. In the first, there is no detection and response in place; we let the emulation run and track the total number of infections as time progresses. In the second, we configured *Jaal* to detect the scan (high variation in destination IP for common target ports, which are in our case 23 and 2323). We assume that the administrator shuts down traffic from an infected node whenever the scan is detected. *Jaal* detects the scan with an accuracy of 95% within 3s. Figure 8 shows the result of this experiment. We see that due to the brute force nature of the scan, the number of infected devices rises almost exponentially if left unchecked. With *Jaal*'s detection and the consequent response in place, the total number of infected devices never rises above 50 (infected devices are detected within 3s regardless). This means that in the worst case, there is a three-fold decrease in the number of devices that could have launched the subsequent DDoS attack. If the DDoS attack is triggered later, the number of devices could be significantly smaller (as seen in the figure) since additional devices are detected and disabled.

Comparison to Reservoir Sampling: Next, we compare *Jaal*'s performance with that of a system using a state of a art sampling technique, viz., *reservoir sampling* [56] [35]. For fair comparison, we set up reservoir sampling with roughly the same communication overhead incurred when *Jaal* uses $r = 12, k = 200, n = 1000$. Specifically, we set the size of the reservoir to 250 and then wait until the sampler has processed 1000 packets prior to shipping the samples. Table 1 shows the comparison results. Since reservoir sampling keeps a fixed-size running uniform sample of the entire stream, attack packets sent over a short period of time will get "diluted" in the sample by a large number of non-attack packets and thus will not be well represented. This results in poor detection accuracy. Note that it is possible to bias the sampling in favor of specific header fields, but this would not be a "general" approach towards detecting a large class of attacks.

8.2 Individual Module Performance

Next, we examine the performance of *Jaal*'s modules.

Greedy flow assignment: We compare *Jaal*'s greedy flow assignment to the optimal online algorithm (i.e., Robin Hood). The weights for Robin Hood are given (we know the ground truth) and the assignment is done on a per-flow basis; this is an ideal but impractical scenario. We also consider an algorithm which randomly assigns flows to any monitor in the corresponding monitor group. We consider Topology 1, and fix the number of monitors to 25. We use a load update period $P = 2$s. In Fig. 9, we plot the time averaged load for different monitor groups j. We see that the greedy assignment closely mirrors the performance of the Robin Hood algorithm (with deviations of 10% on average and 14% in the worst case). The random assignment performs poorly as expected. The results are similar for topology 2 and are omitted in the interest of space.

Dimensionality reduction using SVD: Fig. 10 depicts the variation in the magnitude of singular values (recall § 4.2) for $n = 1000$. The drastic drop in magnitude beyond the top 14 values shows that the lower values are either zero or are very small and can thus be ignored compared to the dominant singular values. In fact, as seen in Fig. 5, $r = 12$ gives us the best trade off between accuracy and communication overhead.

Variance estimation: As discussed in § 5, variance in fields such as port numbers, is used in *Jaal*'s postprocessor to detect distributed attacks. We examine how good is *Jaal*'s estimate of the variance in the destination port header fields, as we vary k, for different batch sizes n. We observe that the error in variance estimation is less than 5% when $k/n > 0.2$, i.e., when summaries cost only 20% compared to sending raw packet headers, and $n \geq 1000$. Next, we study the effect of batch size on the communication costs. In Fig. 11, we plot the compression ratio $\eta = 1 - k/n$ (which is proportional to the communication cost saved) vs. the batch size n for two different maximum variance estimation errors ϵ. As the batch size increases, *Jaal* attains better compression ratios for a given maximum error ϵ. For example, for $\epsilon = 5\%$, and $n = 2000$, *Jaal* achieves a compression ratio of about 85%. To ensure very low ϵ (alternatively very high accuracy) when packet arrival rates are low, a monitor either needs to use a smaller η (which may be acceptable given the lower load) or wait until a larger number of packets are accumulated prior to summarization (thus delaying inference).

Discussion on TPR and FPR: Finally, we analyze why attacks are missed and why there are false positives. We examine the centroids to which packets get assigned by parsing the entire list of centroids and the packets that have been assigned to them and determine which malicious packets are assigned to centroids representing normal traffic and vice versa. We find that both kinds of errors occur when attack packets are similar to background traffic and the clustering is not able to differentiate between them. However, we point out that both the TPR of 98% and FPR of 9.1% that *Jaal* achieves, are well within the reported acceptable performance levels of a NIDS such as Snort or Bro [1, 31, 47].

One could further request finer grained summaries or raw packets, when an alert is to be raised to reduce false positives (with increased cost); conceivably, even random retrieval of full traces when an attack is flagged, could reduce the FPR. However, we leave these possibilities to future work.

9 RELATED WORK

In this section, we discuss relevant related work.

NIDS: There have been attempts to address the scalability issues in centralized NIDS [38, 54] but only to the extent of scaling to enterprise scale networks. These solutions do not hold up to the ISP scale intrusion detection. For example, the approach in [38] assumes that there will be a single entry and exit point into the network which is not true for large WANs. The framework presented in [54] requires the transfer of raw packets to clusters to

make inferences, which as discussed, leads to both performance and inference accuracy degradation.

Network Monitoring, Sampling, Sketching: Packet sampling has been proposed for heavy hitter detection [41], packet length estimation [57] and flow size estimation for small flows [39]. In [34], the authors provide an API for collecting flow statistics at different aggregation levels. These approaches however, are tailored towards measuring only a few pre-specified metrics of interest. In [52, 53] a more general sampling strategy is developed; the strategy however, is only effective in measuring aggregate statistics over streams (e.g. heavy hitter detection, entropy estimation). This is not conducive to general intrusion detection where retaining the correlations across individual headers is necessary.

There has been progressive work on estimating various network related metrics via sketching [44, 49]. While sketches, unlike sampling, offer cost and accuracy guarantees, they suffer from the same fundamental problem of being restricted to measuring some aggregate statistic over a specified dimension. Our solution can handle the generality of NIDS rules by retaining finer grained information.

10 DISCUSSION

In this section, we discuss a few avenues for future work as well as areas where *Jaal* can be improved.

Payload-based Attacks: *Jaal* was not primarily designed to detect payload based attacks. We believe that payload monitoring by ISP's raises some important privacy questions that deserve debate. However, *Jaal* can handle some rudimentary payload-based attacks with a simple extension. Specifically, one approach to detect the presence and/or count of certain keywords (e.g., a specific malicious website, or the term ".exe" which signifies the presence of an executable) is to construct a term frequency matrix using a batch of packets - a popular technique used in sentiment analysis and recommender systems [51]. This matrix can then be treated the same way as the headers-only batch is considered in this paper.

False Positives: One area of concern with deploying *Jaal* is the high FPR. We argue that the TPR and FPR rates achieved by *Jaal* are significantly better than what is possible today. In other words, there are no systems to perform effective intrusion detection at these scales with these TPR/FPR rates. Future work will look into reducing this false positive rate (e.g., we will examine if using multiple windows of packet summaries and correlating the of inferences from those windows can help in this regard). Note that the high FPR is a problem inherent to signature based systems and is not unique to *Jaal*. Since a system such as *Jaal* has never been deployed at ISP-scale networks, it is unclear what the implications are in terms of FPR at that scale. However, we expect analysts to parse logs just as they would for an enterprise IDS. Establishing a dialogue with ISP's about the deployment of *Jaal* is left to future work.

Applicability: While we showcase *Jaal*'s performance with a specific set of attacks, we expect *Jaal* to detect any attack that can be characterized by a Snort/Bro-style packet signature. However, we acknowledge that attacks that only need a few malicious packets to succeed might be more challenging. For example, the TCP reset attack needs only one packet with the RST field set to induce malicious activity. Currently, *Jaal* cannot handle such attacks because a single packet will likely get assigned to an existing centroid that is not representative of it. Fixing this problem requires changes to the clustering algorithm and we defer improvements along this direction to future work.

We also note that *Jaal* can be used alongside smaller-scale (e.g., enterprise) IDSs. *Jaal* can also be used in conjunction with sketches. For example, sketches could be used for simple heavy hitter detection while *Jaal* can be used to detect more complicated attacks requiring correlation across multiple header fields.

Adaptive attackers: Finally, a last avenue of future work is evaluating how robust *Jaal* is to an intelligent attacker that is aware of how *Jaal* works. Whether or not an attacker can craft packets to explicitly bias the summarization process is something we intend to explore in future work.

11 CONCLUSIONS

Challenges of scale, flexibility and complexity have hindered the realization of a NIDS that can be deployed on an ISP scale network. We propose a framework *Jaal* that addresses this long-standing challenge by exploiting in-network processing to generate fine grained, yet concise packet summaries, which can be analyzed centrally to deliver highly accurate inferences with regards to a wide range of attacks. *Jaal* reduces overheads by over 65% compared to sending raw packets (state of the art today) while achieving a detection accuracy of over 98%. This detection accuracy, we believe, is the highest reported for intrusion detection at ISP scale.

Acknowledgment: We thank our shepherd Cristina Nita-Rotaru and the anonymous reviewers for their constructive comments which helped us significantly improve the paper.

The testbed used in this project was supported by the ARO DURIP grant W911NF1510508.The effort described in this article was partially sponsored by the U.S. Army Research Laboratory Cyber Security Collaborative Research Alliance under Cooperative Agreement W911NF-13-2-0045. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to re- produce and distribute reprints for Government purposes, notwithstanding any copyright notation hereon.

REFERENCES

[1] 2001. Strategies to Reduce False Positives and False Negatives in NIDS. https://www.symantec.com/connect/articles/strategies-reduce-false-positives-and-false-negatives-nids. (2001).
[2] 2012. Sockstress Tools & Source Code. https://defuse.ca/sockstress.htm. (2012).
[3] 2015. The Expanding Role of Service Providers in DDoS Mitigation. https://resources.arbornetworks.com/i/481939-the-expanding-role-of-service-providers-in-ddos-mitigation?hubItemID=55526068. (2015). [Online; accessed 23-Jan-2017].
[4] 2016. The Biggest Data Breaches in 2016, So Far. https://www.identityforce.com/blog/2016-data-breaches. (2016). [Online; accessed 10-Jan-2017].
[5] 2016. How the Dyn DDoS attack unfolded. http://www.networkworld.com/article/3134057/security/how-the-dyn-ddos-attack-unfolded.html. (2016).
[6] 2016. jgamblin/Mirai-Source-Code. https://github.com/jgamblin/Mirai-Source-Code. (2016).
[7] 2016. KrebsOnSecurity Hit With Record DDoS. https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/. (2016).
[8] 2016. Large DDoS attacks cause outages at Twitter, Spotify, and other sites. https://techcrunch.com/2016/10/21/many-sites-including-twitter-and-spotify-suffering-outage/. (2016).
[9] 2016. Lessons From the Dyn DDoS Attack. https://www.schneier.com/blog/archives/2016/11/lessons_from_th_5.html. (2016).

[10] 2016. Mirai IoT Botnet Description and DDoS Attack Mitigation. https://www.arbornetworks.com/blog/asert/mirai-iot-botnet-description-ddos-attack-mitigation/. (2016).

[11] 2016. Mirai: what you need to know about the botnet behind recent major DDoS attacks. https://www.symantec.com/connect/blogs/mirai-what-you-need-know-about-botnet-behind-recent-major-ddos-attacks. (2016).

[12] 2016. Someone Is Learning How to Take Down the Internet. https://www.schneier.com/blog/archives/2016/09/someone_is_lear.html. (2016).

[13] 2016. Top 7 types of network attacks. http://www.calyptix.com/top-threats/top-7-network-attack-types-2016/. (2016).

[14] 2017. Cyber-Hunting at Scale (CHASE). https://www.fbo.gov/index?s=opportunity&mode=form&id=a6b09e0661902c71a9c3205db0fff55d&tab=core&_cview=1. (2017).

[15] 2017. MAWI Working Group Traffic Archive. http://mawi.wide.ad.jp/mawi/. (2017).

[16] 2017. Nmap: the Network Mapper. https://nmap.org. (2017).

[17] 2017. Rule Doc Search. https://snort.org/rule-docs. (2017).

[18] 2017. Ryu SDN Framework. https://osrg.github.io/ryu/. (2017).

[19] 2017. Sid 1-19559. https://www.snort.org/rule_docs/1-19559. (2017).

[20] 2017. Sid 3-16294. https://www.snort.org/rule_docs/3-16294. (2017).

[21] 2017. Snort. https://www.snort.org. (2017).

[22] 2017. Snort Users Manual. https://www.snort.org/documents/snort-users-manual. (2017).

[23] Susanne Albers. 2003. Online algorithms: a survey. *Mathematical Programming* 97, 1-2 (2003), 3–26.

[24] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. 2009. NP-hardness of Euclidean sum-of-squares clustering. *Machine learning* 75, 2 (2009), 245–248.

[25] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the Mirai Botnet. (2017).

[26] David Arthur and Sergei Vassilvitskii. 2007. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1027–1035.

[27] Yossi Azar, Andrei Z Broder, and Anna R Karlin. 1994. On-line load balancing. *Theoretical Computer Science* 130, 1 (1994), 73–84.

[28] Yossi Azar, Bala Kalyanasundaram, Serge Plotkin, Kirk R Pruhs, and Orli Waarts. 1997. On-line load balancing of temporary tasks. *Journal of Algorithms* 22, 1 (1997), 93–110.

[29] Allan Borodin, Nathan Linial, and Michael E. Saks. 1992. An Optimal On-line Algorithm for Metrical Task System. *J. ACM* 39, 4 (Oct. 1992), 745–763. https://doi.org/10.1145/146585.146588

[30] Lothar Braun, Cornelius Diekmann, Nils Kammenhuber, and Georg Carle. 2013. Adaptive load-aware sampling for network monitoring on multicore commodity hardware. In *IFIP Networking Conference, 2013*. IEEE, 1–9.

[31] S Terry Brugger and Jedidiah Chow. 2007. An assessment of the DARPA IDS Evaluation Dataset using Snort. *UCDAVIS department of Computer Science* 1, 2007 (2007), 22.

[32] Waleed Bulajoul, Anne James, and Mandeep Pannu. 2013. Network intrusion detection systems in high-speed traffic in computer networks. In *e-Business Engineering (ICEBE), 2013 IEEE 10th International Conference on*. IEEE, 168–175.

[33] Michael Buratowski. 2016. The DNC server breach: who did it and what does it mean? *Network Security* 2016, 10 (2016), 5–7.

[34] Shihabur Rahman Chowdhury, Md Faizul Bari, Reaz Ahmed, and Raouf Boutaba. 2014. PayLess: A low cost network monitoring framework for software defined networks. In *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 1–9.

[35] Nick Duffield, Carsten Lund, and Mikkel Thorup. 2007. Priority sampling for estimation of arbitrary subset sums. *Journal of the ACM (JACM)* 54, 6 (2007), 32.

[36] Carl Eckart and Gale Young. 1936. The approximation of one matrix by another of lower rank. *Psychometrika* 1, 3 (1936), 211–218.

[37] Cristian Estan, Ken Keys, David Moore, and George Varghese. 2004. Building a better NetFlow. In *ACM SIGCOMM Computer Communication Review*, Vol. 34. ACM, 245–256.

[38] Victor Heorhiadi, Michael K Reiter, and Vyas Sekar. 2012. New opportunities for load balancing in network-wide intrusion detection systems. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 361–372.

[39] Chengchen Hu, Sheng Wang, Jia Tian, Bin Liu, Yu Cheng, and Yan Chen. 2008. Accurate and efficient traffic monitoring using adaptive non-linear sampling method. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE, 26–30.

[40] Ian Jolliffe. 2002. *Principal component analysis*. Wiley Online Library.

[41] N Kamiyama and T Mori. [n. d.]. Simple and Accurate Identification of High-Rate Flows by Packet Sampling. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*.

[42] George Khalil. 2015. *Open Source IDS High Performance Shootout*. White Paper. SANS Institute.

[43] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. 2003. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. ACM, 234–247.

[44] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 101–114.

[45] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[46] Jianning Mai, Chen-Nee Chuah, Ashwin Sridharan, Tao Ye, and Hui Zang. 2006. Is sampled data sufficient for anomaly detection?. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM, 165–176.

[47] Samuel Patton, William Yurcik, and David Doss. 2001. An AchillesâĂŹ heel in signature-based IDS: Squealing false positives in SNORT. In *Proceedings of RAID*, Vol. 2001.

[48] Vern Paxson. 1999. Bro: a system for detecting network intruders in real-time. *Computer networks* 31, 23 (1999), 2435–2463.

[49] Anirudh Ramachandran, Srinivasan Seetharaman, Nick Feamster, and Vijay Vazirani. 2008. Fast monitoring of traffic subpopulations. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*. ACM, 257–270.

[50] Martin Roesch. 1999. Snort-Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX conference on System administration*. USENIX Association, 229–238.

[51] Lior Rokach and Oded Maimon. 2005. Web Mining. In *Data mining and knowledge discovery handbook*. Springer, 321–352.

[52] Vyas Sekar, Michael K Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G Andersen. 2008. CSAMP: A System for Network-Wide Flow Monitoring.. In *NSDI*, Vol. 8. 233–246.

[53] Vyas Sekar, Michael K Reiter, and Hui Zhang. 2010. Revisiting the case for a minimalist approach for network flow monitoring. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 328–341.

[54] Praveen Kumar Shanmugam, Naveen Dasa Subramanyam, Joe Breen, Corey Roach, and Jacobus Van der Merwe. 2014. DEIDtect: towards distributed elastic intrusion detection. In *Proceedings of the 2014 ACM SIGCOMM workshop on Distributed cloud computing*. ACM, 17–24.

[55] Neil Spring, Ratul Mahajan, and David Wetherall. 2002. Measuring ISP topologies with Rocketfuel. *ACM SIGCOMM Computer Communication Review* 32, 4 (2002), 133–145.

[56] Jeffrey S Vitter. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)* 11, 1 (1985), 37–57.

[57] Lili Yang and George Michailidis. 2007. Sampled based estimation of network traffic flow characteristics. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*. IEEE, 1775–1783.