A Two-stage Heavy Hitter Detection System Based on CPU Spikes at Cloud-Scale Gateways

Jianyuan Lu*, Tian Pan†, Shan He*, Mao Miao*, Guangzhe Zhou*, Yining Qi‡, Biao Lyu*, Shunmin Zhu*
*Alibaba Group
†State Key Laboratory of Networking and Switching Technology, BUPT, Beijing, China
‡State Key Laboratory of Industrial Control Technology, Zhejiang University, China

Abstract*—The cloud network provides sharing resources for tens of thousands of tenants to achieve economics of scale. However, heavy hitters engendered by a single tenant will probably interfere with the processing of the cloud gateways, undermining the predictable performance expected by other cloud tenants. To prevent it, heavy hitter detection becomes a key concern at the performance-critical cloud gateways but faces the dilemma between fine granularity and low overhead. In this work, we present *CloudSentry*, a scalable two-stage heavy hitter detection system dedicated to multi-tenant cloud gateways against such a dilemma. CloudSentry contains a lightweight coarse-grained detection running continuously to localize infrequent CPU spikes. Then it invokes a fine-grained detection to precisely dump and analyze the potential heavy-hitter packets at the CPU spikes. After that, a more comprehensive analysis is conducted to associate heavy hitters with the cloud service scenarios and invoke a corresponding backpressure procedure. CloudSentry significantly reduces computation, memory and storage overhead compared with existing approaches. Additionally, it has been deployed world-wide in Alibaba Cloud for over eleven months, with rich deployment experiences. In a gateway cluster under an average traffic throughput of 251Gbps, CloudSentry consumes only a fraction of 2%-5% CPU utilization with 8KB run-time memory, producing only 10MB heavy hitter logs during one month.

I. Introduction

Today's cloud service vendors [1]–[3] provide a wide range of public cloud services across the world, such as computing cloud service, storage cloud service and network cloud service. Based on the diverse cloud services, cloud users (*aka*, tenants) can flexibly and cost-effectively deploy and customize their own applications, focusing more on the business logic, without worrying too much about the underlying infrastructure management. One of the most significant features of the cloud service is the *sharing of resources* to achieve economies of scale, which can further be classified into different sharing models. For example, one network device can be shared by multiple tenants, usually through oversubscribing Virtual Machines (VMs) based on a hypervisor [4]. The same network device can also be shared by many services or network functions [5]. At a closer look, one network device can also be shared by massive concurrent flows, generated from multiple services launched by tenants [6]. Generally, tenants customize their own applications in the cloud through *cloud service gateways* [7]. Under the varying sharing models, contention between different tenants, services or flows will arise at the gateways if there is no guaranteed resource isolation mechanism. According to our experience of cloud gateway management, the traffic bursts or heavy-hitter flows are triggered by a few tenants in most cases. It will probably affect the forwarding performance of other tenants, since even a single tenant's traffic bursts may starve gateway's network resource while the strict resource isolation at the network level has not been readily deployed [4]. Therefore, for public clouds, the detection of heavy hitters and the appropriate response measures to the detected heavy hitters are critical to the stable performance of cloud services.

For public cloud services, heavy hitter detection needs to address the following three challenges.

- Low overhead. The performance overhead of heavy hitter detection reflects in three aspects: computation, memory and storage. First, lots of heavy hitter detection approaches are conducted on the data plane [8]–[12] for rapid traffic anomaly detection. Such a detection should be designed as lightweight as possible to avoid interfering with the normal traffic forwarding. Second, modern Data Center(DC) switches generally equip with very compact memory (SRAM and TCAM) to accommodate fast forwarding tables [5]. The size of data structures for heavy hitter detection should be constrained to fit in the limited remaining memory. Last but not least, storage is also an important metric once overlooked. A real cloud infrastructure needs to preserve system logs about the detected heavy hitters for a long time (*e.g.*, 30 days) for retrieval and analysis of historical problems. Therefore, heavy hitter storage should better be succinct enough to save operation and maintenance expenses.
- Both incast and elephant-flow detection. Incast [13] is an aggregation of multiple concurrent mice flows which together pose great traffic processing pressure to the system. While an elephant flow is a single flow with persistent traffic burst. In cloud gateways, we find both traffic patterns of heavy hitters prevail and cause severe damages to the predictable performance of cloud tenants.
- World-wide scalability and deployability. Since cloud infrastructures for public cloud services are built globally with a great many data centers distributed at regions and zones, heavy hitter detection approaches should have horizontal scalability and rapid deployability.

Although extensive research has been conducted for heavy hitter detection, few existing approaches achieve low overhead at computation, memory and storage simultaneously. For example, NetFlow [14], sFlow [15] have low computation overhead but at the cost of considerable memory consumption. Sketch-based approaches [10]–[12], [16], [17] are memory-efficient with succinct probabilistic data structures but it is hard to enumerate or perform statistics about the heavy hitters since the flow identifiers are not explicitly preserved. Approaches like Flowradar [18] dump the detected heavy hitters every 10ms, engendering enormous storage overhead for the log system of globally distributed cloud environment. Besides, existing approaches are designed to detect heavy hitters caused either by incast [13], [19] or by elephant flows [8], [20] and very few approaches can detect both at the same time. For scalability and deployability, most approaches focus on single-point heavy hitter detection. A few approaches claim to conduct network-wide detection [21], [22] but still are evaluated inside a small local network. Heavy hitter detection research at cloud-scale environment dealing with high-throughput production traffic is still a missing piece.

At cloud scale, conducting heavy hitter detection at production traffic continuously in 7*24 hours (24 hours a day, 7 days a week) need massive amount of computation, memory and storage resources. Our measurement of cloud gateway traffic indicates that for most of the time, the traffic is steady with minor packet drops, thanks to the horizontal expansion of the processing capacity in the cloud. That is to say, fine-grained detection at production traffic for all the time wastes a large amount resources, which however, is done by most state-of-the-art approaches. Some sampling-based detection approaches [14], [15] can reduce the performance overhead to some extent, but cannot precisely localize the heavy hitters.

In this work, we propose *CloudSentry*, a two-stage heavy hitter detection system based on CPU spikes for cloud-scale multi-tenant gateways. Unlike the existing approaches detecting heavy hitters in 7*24 hours, CloudSentry follows a two-stage heavy hitter detection architecture that contains a lightweight coarse-grained detection plus an event-based fine-grained detection. The coarse-grained detection monitors CPU utilization as the indicator for 7*24 hours, which faithfully reflects the traffic load changes on the gateways in real time. Once CPU utilization spikes persist for some time, the fine-grained heavy hitter detection will be triggered to dump the packets and then conduct further analysis. Currently, CloudSentry is implemented and deployed at software cloud gateway clusters. We believe, the event-based heavy hitter detection paradigm can also be extended to hardware-based systems.

Our major contributions are summarized as follows:

- We propose CloudSentry, a two-stage heavy hitter detection approaches based on CPU spikes for cloud-scale multi-tenant gateways. Such an event-triggered, on-demand monitoring system is lightweight and accurate enough to be deployed in globally distributed cloud environment. Under the meticulous guard of CloudSentry at the cloud gateways, predictable performance of cloud tenants becomes more tractable than ever.
- Different from the previous works that focus on finding out heavy hitters under a single definition of either incast or elephant flows, CloudSentry is versatilely designed to detect both incast and elephant flows at the same time.
- With extensive real cloud traffic measurement, we discover the "dominant-tenant effect" that most of the simultaneous heavy hitters are correlated and generated by one dominant tenant. That is to say, such a dominant tenant will probably invoke several elephant flows or thousands of mice flows (the incast case) in a very short time. Accordingly, CloudSentry leverages such observations for rapid heavy hitter filtering.
- CloudSentry has been steadily working in Alibaba Cloud for over eleven months, from which the rich experiences are learned and shared in the paper.

II. Background and Motivations

A. Cloud Service Gateways

Multi-tenant cloud service gateways. Virtual cloud network services offer opportunities for users to customize their own private networks in the cloud through multi-tenant cloud gateways [7]. Scaling with growing cloud traffic, these gateways need to handle millions of internal tunnels, providing functions like forwarding, encapsulation/decapsulation, Quality of Service (QoS) provisioning and security protection [6], [23]. Since a cloud gateway is generally shared by multiple tenants with massive concurrent flows, sudden arrival and persistence of abnormal heavy-hitter flows will potentially drain the available CPU and memory resources of the gateway, leading to starvation of other processing tasks in peak hours and affecting the latency-sensitive traffic forwarding of the remaining tenants. Therefore, cost-effective heavy hitter detection and then targeted rate limiting are crucial for building resilient cloud gateways.

Hardware and software gateways. Cloud service gateways can be classified into hardware and software gateways. The hardware gateways are generally built with fixed-function switches [24], programmable switches [25] or smartNICs [26], which address the forwarding capacity problem, *e.g.*, P4-based Tofino switch chip can provide Tbps-level forwarding throughput and μs-level forwarding latency. However, constrained by the hardware resources (*e.g.*, limited TCAM for storing flow items) and fixed configuration, the hardware gateways lack scalability to maintain huge concurrent sessions and flexibility to provide diverse service capabilities. To this end, software gateways [27], [28] are deployed to address the flexibility problem. Software gateways are based on x86 bare-metal servers or virtualization techniques such as VM [29] and docker [30]. Software gateways are highly flexible to implement diverse applications such as DPI and encryption. The defect of software gateways is the low processing capability, especially for a single CPU core, which is vulnerable to persistent traffic pressure (packets of one heavy-hitter flow will be hashed into the same CPU core for in-order processing). In this work, we focus on improving the performance stability of software gateways to steadily serve concurrent tenants.

B. Traffic Characteristics in Cloud Service Gateways

We measure the real traffic characteristics in Alibaba Cloud software gateways. The results show that network traffic is steady for most of the time and traffic bursts do happen, exerting a big pressure to gateways, but at a very low frequency.

Specifically, Fig. 1 shows the network traffic and packet drops on NICs in one Alibaba Cloud gateway cluster for three weeks. We can find that the traffic volume changes periodically (day and night) and steadily in general without frequent persistent bursts. The packet drops are much lower compared with the traffic volume by (at least) four orders of magnitude. Furthermore, the packet drops are evenly distributed, the long spikes only occur in a few moments, *e.g.*, in the 16th days. The spikes of packet drops essentially reflect the bursts of traffic and the corresponding pressures to the gateways.

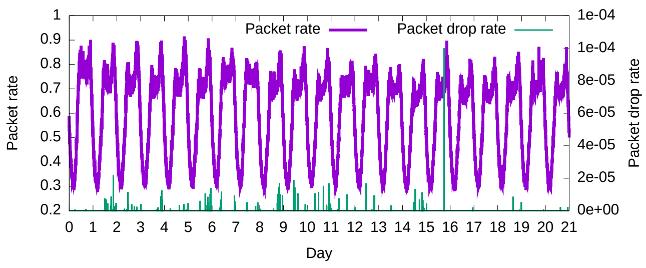

Fig. 1. Network traffic and packet drops on NICs in a typical Alibaba Cloud software gateway cluster (The traffic rate has been normalized, start date: 2020/06/14 00:00:00, end date: 2020/07/05 00:00:00)

Other gateway clusters show similar network traffic and packet drop pattern. It can be inferred that there is no packet drop for most of the time. Further analysis shows the packet drop probability follows an obvious heavy-tail distribution, which means persistent heavy-hitter flows that cause severe packet loss at gateway NICs are very rare.

Since persistent heavy-hitter flows occur infrequently, if we could detect these abnormal flows on demand rather than in a 7*24 manner, the computation and memory overhead for heavy hitter detection can be drastically eliminated.

C. Motivations

According to the above measurement results, we show how CloudSentry detects the heavy-hitter flows and improves the vulnerability of cloud gateways. CloudSentry follows a two-stage heavy hitter detection architecture that contains a lightweight coarse-grained detection plus an event-based fine-grained detection. The coarse-grained detection runs for 7*24 hours, aiming to localize sudden changes of steady traffic. Specifically, we select CPU utilization as the indicator, which is easy to acquire but faithfully reflects the traffic load changes on the gateways in real time. Once CPU utilization spikes persist for a predefined period of time, we further start the fine-grained heavy hitter detection to dump the packets, upload the packets, analyze the results, associate the root causes and make corresponding backpressure measures for rate limiting.

The CPU utilization monitoring in coarse-grained detection stage is conducted at the control plane in a fairly slow pace with zero disturbance to the high-performance data plane forwarding. The fine-grained detection is conducted at the data plane but in a very low frequency triggered by CPU spike events which are generated by the coarse-grained detection. Such a two-stage detection mechanism eliminates the data plane interference and reduces the computation and memory overhead as much as possible.

Compared with general heavy hitter detection approaches, CloudSentry conducts targeted heavy hitter detection. That is to say, with performance scalability taken into consideration, our system only focuses on detecting and rate limiting the heavy-hitter flows that cause forwarding performance degradation, which is directly indicated by CPU utilization spikes.

III. System Design

In this section, we describe the basic architecture of CloudSentry. CloudSentry is composed of two parts: coarse-grained detection (§III-A) and fine-grained detection (§III-B). As CloudSentry is dedicated to the software gateway system, it measures the traffic load changes via the CPU spikes, which is further regarded as the evidence of potential heavy hitters. Once a CPU spike is detected, it triggers a more detailed heavy hitter detection to find out the candidate heavy hitters as well as the root causes behind.

A. Coarse-grained Detection

Coarse-grained detection is shown in Fig. 2. The distributed deployed gateways will report their CPU utilization rates periodically to a storage system. The coarse-grained detection algorithm is very simple. It first reads these CPU utilization rates, if one of the CPUs exceeds a pre-defined threshold T_1, then it triggers the next stage process (fine-grained detection).

It is obvious that coarse-grained detection does not affect packet forwarding in the data plane, instead, it only measures the CPU utilization rate in the control plane. This design avoids invoking a large amount of unnecessary background measurement when cloud service traffic is in the steady period, reducing the impact on the data plane as much as possible.

Usually several gateways form a service cluster, providing a unified Virtual IP (VIP) for service access. Flows generated by the same service in one cluster are strongly correlated, which results in a strong correlation of heavy hitters in the same cluster. Therefore the detection (including fine-grained detection in next stage) is cluster-based, which tests a cluster's CPU utilization rates and triggers CPU spike events.

B. Fine-grained Detection

Fine-grained detection is shown in Fig. 3. It is triggered by the condition that CPU spikes occur in the coarse-grained detection. Fine-grained detection is divided into three procedures: *packet dump*, *top-flow analysis*, and *comprehensive analysis*.

Packet dump. The CPU-spike usually is a transient event, which means most of the traffic bursts last for a very short time, *e.g.*, less than one second, which will not drastically affect the data plane performance. A few works try to solve

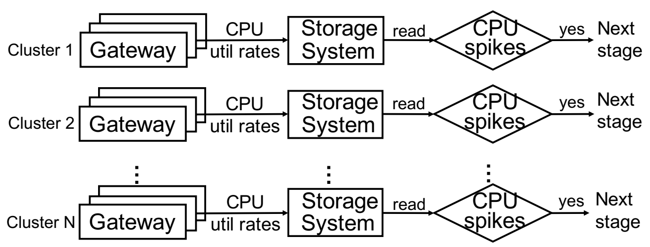

Fig. 2. First Stage: Coarse-grained Detection

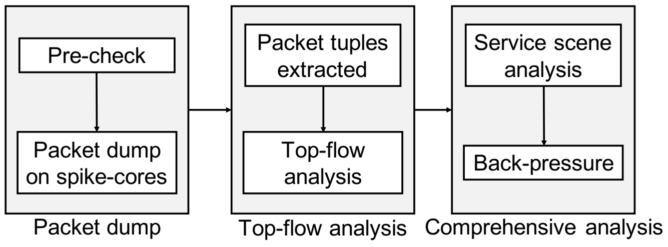

Fig. 3. Second Stage: Fine-grained Detection

micro-burst in data centers [31], [32], while we concentrate on persistent traffic bursts. To eliminate frequent interruption caused by the micro-burst effect, we perform a pre-check process before packet dump on gateways. If the pre-check confirms the current CPU utilization rate is lower than a threshold T_2, the detection will skip this pass of packet dump procedure. Only when the current CPU utilization exceeds T_2, the packet dump will be performed. Packet dump is executed on the specified CPU core which has a spike, others without spikes will not be affected. A sampling method is used in packet dump to reduce the performance interference of packet dump to the data plane. We also design an algorithm for packet dump to further protect packet forwarding by limiting the maximum packet dump number in a short time interval. The packet dump details are introduced in §IV-B.

Top-flow analysis. Once packet dump ends, the detection begins to extract the candidate flow tuples (*i.e.*, potential heavy hitters), and then performs top-flow analysis. A tuple means a field in the packet header, *e.g.*, source IP address, destination IP address, *etc*. The extracted tuples will be stored in the storage system. The top-flow analysis algorithm reads these tuples and find out top flows which potentially cause CPU spikes. Flows are aggregated in different dimensions, *e.g.*, five-tuple flow <sIP,dIP,sPort,dPort,proto>, one-tuple flow <tunnel-id>. Multilevel top flows have different functions for further analysis with diverse services and complex forwarding paths. The top-flow analysis details are introduced in §IV-C.

Comprehensive analysis. The top-flow analysis only conducts flow-level detection. In the cloud environment, we further need to find out which tenant and which service cause a heavy-hitter impact. The tenant-level detection is easy to accomplish by aggregating the flow-level results with tenant identifiers. The service-level detection is a more complicated task. In addition to the need of more tuples than traditional heavy hitter detection, it also needs configuration information in the control plane. These analysis is further leveraged by *backpressure*, which is used to prevent gateways from experiencing persistent high processing delay and packet loss rate. Sometimes, heavy hitters last for a very long-time, *e.g.*, tens of minutes. These heavy hitters bring a huge decrease to processing performance of gateways. Therefore, once detected, we need to eliminate the heavy hitter's damage and protect the gateways. Backpressure is one solution in the cloud to limit the sending rate of a heavy hitter [4]. The limitation of backpressure is that it only works for heavy hitters from cloud controlled resources. The comprehensive analysis details are introduced in §IV-D.

IV. Implementations

In this section, we show the implementation details and deployment experience of CloudSentry in Alibaba Cloud.

A. CPU Utilization Collection and Check

We implement a CPU utilization collection process on each gateway. It reads the current CPU utilization rates every second and uploads them to a remote storage system through APIs. In case network jitters, the process first records the CPU rates to local disk, then uploads them to storage system. The storage system we employ is a public cloud storage service, named SLS [1], which provides high-throughput and stable log reads and writes. The coarse-grained detection pulls the CPU utilization rates from SLS and checks whether a CPU exceeds a spike threshold. The spike threshold is usually set to be $T_1 = 90\%$, because when a CPU utilization reaches 90%, the corresponding gateway starts to drop packets with a higher probability. The CPU utilization collection and check runs for 7*24 hours. Both the CPU utilization collection (on gateways) and spike threshold check (on a centralized controller) are light-weight implementations. Such a manner does not affect the data plane and saves a large amount of processing ability when gateways are not in performance bottlenecks caused by heavy-hitter flows. Once a CPU spike is detected, it would trigger a packet dump stage (in fine-grained detection).

B. Packet Dump

In §III, we have mentioned to have a pre-check procedure in gateway servers before packet dump, with which small-burst can be filtered. In our implementation, the pre-check threshold is set to be $T_2 = 50\%$. It means if the current CPU utilization rate is higher than 50%, we will start the packet dump process. Otherwise, we skip it because the heavy hitter has gone. Note that $T_2 < T_1 = 90\%$. A smaller T_2 means more packets will be captured in case heavy-hitter flows be neglected. The online gateways usually have strict cluster capacity management. If the average utilization rate is close to 50%, the gateway cluster needs to be expanded. Therefore, the CPU utilization rate will be less than 50% under normal circumstances.

The gateways run at multi-core servers. According to the operator's experience, a gateway server only has one persistent

CPU spike at the same time in most cases. Therefore, the packet dump is executed on a specified core which triggered a spike in the coarse-grained detection. If more than one core has a spike, then in our current system the packet dump is executed sequentially on these cores.

In order to prevent dumping too many packets from affecting the forwarding in data plane, we implement a rate limiting algorithm in packet dump process, shown in Algorithm 1. This algorithm has two purposes. Firstly, restricting the maximum packet dump number, R, in one second. If the target packet dump number x exceeds R, then it takes $\lceil \frac{x}{R} \rceil$ seconds to complete the packet dump. Secondly, sampling the packets evenly in a stream of traffic packets. Note that new_tokens is proportional to the time $interval$ in two consecutive dump packets, therefore the sampling is time evenly distributed. For example, if the rate limiting threshold $R = 100$, and we want to capture 500 packets, then it dumps a packet every 0.01 seconds, and needs 5 seconds to complete the packet dump process.

Algorithm 1: Rate limiting algorithm for packet dump

```
Input: rate limiting threshold $R$, rate limiting record
       $r$, machine frequency $F$
Result: $Accept$ or $Drop$ this packet
1  $cycles\_now \leftarrow$ get_cycles_now();
2  $cycles\_last \leftarrow r.cycles\_last$ ;
3  $interval \leftarrow cycles\_now - cycles\_last$;
4  $new\_tokens \leftarrow \lfloor (interval * R)/F \rfloor$;
5  if $new\_tokens > 0$ then
6      $r.tokens \leftarrow r.tokens + new\_tokens$ ;
7      $r.cycles\_last \leftarrow cycles\_now$ ;
8      if $r.tokens > R$ then
9          $r.tokens \leftarrow R$ ;
10 if $r.tokens > 0$ then
11     $r.tokens \leftarrow r.tokens - 1$ ;
12     return $Accept$ ;
13 return $Drop$;
```

C. Top-flow Analysis

We need to extract the candidate tuples of the dumped packets for further analysis. Cloud operators usually run an overlay network to isolate traffic from different tenants. The packet format for overlay network in Alibaba Cloud is Vxlan [33]. In order to do a thorough analysis, we extract the Vxlan tunnel-id, inner and outer five-tuple, basic packet information, shown in Table I. The *inner-6tuple* and *outer-6tuple* are marked by *, which means that they are logically tuples and do not consume additional storage space.

We can see that *tunnel-id* plays an important role in the top-flow analysis, because most of the concurrent heavy hitters are caused by one tenant (tunnel-id denotes a tenant). We call this phenomenon “dominant-tenant effect”. A dominant-tenant is likely to send out one elephant flow, or multiple small flows belonged to a same service. A single elephant flow can only be sent to a single CPU (avoiding the out-of-order effect). Even though multiple flows are likely hashed to different CPUs by multilevel ECMP paths, the flows with the same service in a tenant may be directed to one CPU due to special overlay forwarding rules[1]. Therefore, if we can find out a dominant-tenant, then we can narrow down the heavy hitter detection range greatly.

Based on the extracted tuples, top-flow analysis algorithm is trying to find out the top flows defined by different combination of tuples, as shown in Algorithm 2. The basic idea is to find out flows that have large z-scores. The definition of z-score is:

$$z\text{-}score = \frac{x - \mu}{\sigma} \tag{1}$$

where x is a variable, μ,σ are the mean, standard deviation of a variable set. The z-score means a deviation from average, and the deviation threshold Z is set to be 1 in our implementation. The algorithm finds out three types of heavy hitters, *i.e.*, tunnel-id, inner-6tuple, outer-6tuple. These heavy-hitter types will be used in comprehensive analysis.

TABLE I
TUPLES EXTRACTED

inner-5tuple	<sIP,dIP,sPort,dPort,proto> from Vxlan inner layer
outer-5tuple	<sIP,dIP,sPort,dPort,proto> from Vxlan outer layer
tunnel-id	tunnel id from Vxlan tunnel layer
pkt_info	packet information, *e.g.*, length, timestamp, location
*inner-6tuple	<tunnel-id, inner-5tuple>
*outer-6tuple	<tunnel-id, outer-5tuple>

Algorithm 2: Top-flow analysis algorithm

```
Input: tunnel-id List $tList$, inner-6tuple List $iList$,
       outer-6tuple List $oList$, z-score threshold $Z$
Result: top tunnel-id List $tTopFlow$, top inner-6tuple
        List $iTopFlow$, top outer-ttuple List
        $oTopFlow$
1  Function findTopFlow(List $l$)
2      $z\text{-}score \leftarrow$ get_zscore($l$) ;
3      $topList$ = [] ;
4      for $i \leftarrow 1$ to $l.length()$ do
5          if $z - score[i] > Z$ then
6              $topList.add(l[i])$ ;
7      return $topList$ ;
8  $tTopFlow \leftarrow$ findTopFlow($tList$) ;
9  $iTopFlow \leftarrow$ findTopFlow($iList$) ;
10 $oTopFlow \leftarrow$ findTopFlow($oList$) ;
```

D. Comprehensive Analysis

In addition to the basic heavy hitter detection, we also do further comprehensive analysis. On the one hand, we associate

[1]These rules are designed for purposes like security, deep packet inspection, traffic shaping, *etc*.

the heavy hitters with the cloud service scenarios, deriving a conclusion that which tenant and which service result in the CPU spikes. Due to the space limitations, we do not show the specific algorithms. The basic idea is that locating the tenant using top tunnel-id, narrowing down the service scope using top inner-6tuple, and locating a specific service using top outer-6tuple. We mainly focus on cloud network services, such as *Internet Service, Cross-region Service, or Hybrid Cloud Service*. From the perspective of cloud gateways, these services (including heavy hitters) have directions, *from-cloud* or *to-cloud*. While *from-cloud* means the traffic is coming from a cloud source, such as a hypervisor, or a cloud service, *to-cloud* is the opposite.

On the other hand, we have implemented a backpressure recommendation algorithm. Lacking traffic scheduling mechanism in different forwarding cores, the software gateways usually suffer from a weak single-core processing capability. Once a heavy hitter happens at a forwarding core, it drastically decreases the Service Level Agreements (SLAs) of this core, incurring increased forwarding delay and packet loss rate. Conducting a rate limiting at the gateway NICs is one solution to relieve the heavy-hitter impact. However, the NICs lack tenant isolation mechanism, which means a rate limiting in NICs may harm the normal traffic of other tenants. Therefore, we take a backpressure method, which limits the heavy hitters at the source. The backpressure algorithm is simple, it halves the forwarding rate of the heavy-hitter sources iteratively until the CPU spikes disappear. Note that we only draw a backpressure decision, not implement the decision automatically. The actual backpressure actions should be confirmed by operators in practice. Obviously, the backpressure is only effective when the heavy-hitter source is in the cloud. If the heavy hitters come from non-cloud sources, such as the Internet, or the tenant data centers, then we need to consider other methods to lessen heavy-hitter damage. For example, we can offload the heavy hitters to high-speed hardware gateways.

E. Deployment in Cloud-scale System

Major cloud providers have tens of regions across the world, with each region containing 5-20 DCs [34]. In our implementation, each DC contains several gateway clusters. Hence, how to deploy a heavy hitter detection in large-scale cloud environment should be considered. The simplest extension is to deploy a collection of heavy hitter detection for each cluster. However, it is very difficult to operate and maintain such a system. Actually, CloudSentry is deployed in a centralized analysis controller, just with a single server. The reason for a centralized deployment is that CloudSentry filters out redundant traffic with best effort for heavy hitter detection.

While heavy hitters may happen in different regions at the same time, a sequential processing model could incur Head-of-Line (HOL) effect. To solve this problem, we run a detection thread for each region. Note that we do not run a detection thread for each cluster, because the CPU spikes happen with a small probability in a practical gateway cluster. We can prove that the sum of a finite number of *i.i.d* Poisson variables is still a Poisson variable. Therefore, according to the Poisson theory, in a short time interval Δt, only one cluster with spikes has the probability $P(n=1)=\lambda\Delta t+o(\Delta t)$, where λ is a constant, and more than one cluster has CPU spikes has the probability $P(n>1)=o(\Delta t)$.

CloudSentry has been deployed in Alibaba Cloud for over six months. It helps the operators identify heavy hitters easily and fast, it also helps to backpressure several persistent heavy hitters triggered by cloud sources.

V. Evaluation

A. Setup

In our evaluation, we select six gateway clusters in Alibaba Cloud to evaluate our system. The basic information and traffic characteristics are shown in Table II. The gateways are implemented on x86-based servers, which run DPDK-based (a kernel bypass technique [35]) forwarding program. Each server is configured with 32-core CPUs (26 cores for packet processing, 6 cores for maintenance tasks like communication with controller and CPU data collection) and four 40Gbps NICs. These clusters are distributed in different regions globally. Each cluster has 6-16 gateways, according to the current traffic rate and traffic volume prediction in the future. The default measurement duration is for one month, with starting date: 2020/06/22 00:00:00, ending date: 2020/07/22 00:00:00.

We implement the heavy hitter detection algorithm on a centralized controller. Both coarse-grained detection (except CPU utilization collection) and fine-grained detection are executed in controller. The controller runs on a server with the same configuration as gateways. In order to improve the parallel detection ability of different regions, we start a detection thread for each region (the selected gateway clusters are in different regions). The SLS log system [1] are used to store CPU utilization rates and several intermediate results.

TABLE II
GATEWAY CLUSTER INFORMATION

Gateway Cluster	**A**	**B**	**C**	**D**	**E**	**F**
Max Mpps	64.2	55.8	126.1	117.9	92.1	101.4
Avg Mpps	15.0	25.9	54.7	60.1	49.8	40.1
Max Gbps	494.4	181.6	670.4	520.0	629.6	393.6
Avg Gbps	74.4	86.4	211.2	251.2	251.2	157.2
# of Gateways	16	8	16	16	8	6

B. CPU Spikes Measurement

We measure the CPU utilization rate of six gateway clusters for one month. Fig. 4 shows the cumulative distribution function (CDF) of their CPU utilization rate. Generally, the average CPU utilization rate of the cloud gateways are relatively low. For example, the average CPU utilization of cluster-A is 4%; the average CPU utilization of cluster-B, cluster-C and cluster-D are around 20%; cluster-E and cluster-F have the highest CPU utilization of around 33%. Furthermore, the CDF curves of the CPU utilization rate of gateway clusters have very long tails, which means CPU's peak utilization occurs infrequently. Such long-tail property motivates us to use CPU spikes to

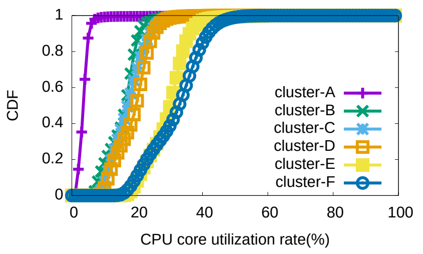

Fig. 4. CDF of CPU utilization rate in six clusters.

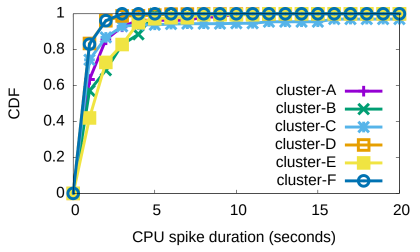

Fig. 5. CDF of CPU persistent high utilization ($> 90\%$) in six clusters.

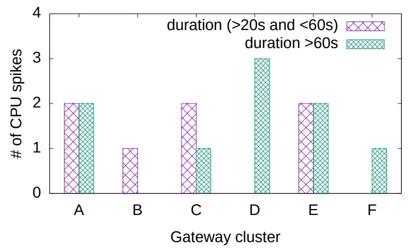

Fig. 6. Long duration of CPU persistent high utilization ($> 90\%$) in six clusters.

trigger the dump and analysis of heavy-hitter flows in a cost-effective way.

Fig. 5 shows the CDF of CPU persistent high utilization ($> 90\%$). It can be inferred from Fig. 5 that one single CPU spike (lasting for less than or equal to one second) occurs much more frequently than CPU spikes lasting for more than one second. Specifically, more than 40% of CPU spikes last for less than or equal to one second; more than 68% of CPU spikes last for less than or equal to two seconds. In the six gateway clusters, most of CPU's high utilization will not persist for a very long time. The CDF curves also have very long tails. Fig. 6 shows the count of long duration (>20s) of CPU persistent high utilization. All the six clusters have less than five times of long duration. It can be concluded that the long duration occurs with a low probability in cloud services. However, even with low probability, once it happened, it will deteriorate the Service Level Agreement(SLA) of thousands of tenants. In the following experiments, we will evaluate a backpressure procedure to eliminate the harm of long duration (>60s) of CPU persistent high utilization Actually, a single CPU spike will not drastically affect the traffic processing of concurrent flows. Even the CPU's peak utilization causes some packet losses, they will be re-transmitted by the end-host TCP stack. Accordingly, in our system design, we conduct a pre-check process to reduce the impact of one single CPU spike in fine-grained detection. However, persistent high CPU utilization will undermine end user's network experience and our system will detect the CPU spikes persist for more than one second.

Fig. 7 shows the additional CPU consumption in control plane to collect the gateway's several basic stats, including the CPU utilization rate. We can see that the additional CPU consumption is around 1%, which shows the coarse-grained detection of CloudSentry is a light-weight monitoring task for the CPU spikes.

C. Packet Dump Performance

Packet dump is an event-triggered action, the execution times are proportional to the CPU spikes. Because a single CPU spike happens more frequently than persistent CPU spikes, we design a pre-check process to filter the single CPU spike. Fig. 8 shows the packet-dump actions with and without pre-check. We can draw two conclusions from this figure. First, packet dump is an infrequent action, less than 300 packet-dump actions in six clusters during one month. Second, the pre-check filters out 80% packet-dump actions at most (in cluster-A). The design philosophy of CloudSentry is to decrease the ineffective measurement to the maximum extent. Fig. 9 shows the comparison between the total data packets (traffic through gateways) and the dumped packets (for heavy hitter detection). We can see that CloudSentry only needs thousands of packets to detect the heavy hitters (for one month), ten orders of magnitude lower than the original data packets. Other approaches can only achieve at most three or four orders of magnitude reduction.

D. Performance Overhead Comparison

We compare our approach (CloudSentry) with other heavy hitter detection approaches like NetFlow [14], HashParallel [8], CM-Sketch [10], NitroSketch [9] in the aspect of hash computation, memory and storage overhead.

Fig. 11 shows the hash computation overhead per device per second in the six clusters under the five approaches. Among the five approaches, NetFlow leverages one hash function to index the flow table of sampled flow items. The hash computation will be invoked once an incoming packet is sampled with a probability of 1/1000. Hence, the hash computation overhead is proportional to the packet arrival rate as well as the sampling rate. HashParallel uses six hash functions in parallel to maintain a flow cache containing the most frequently accessed flow items. The six hash functions are invoked on per-packet basis. Therefore, HashParallel is the most computation-intensive. CM-Sketch achieves the optimal performance with five hash functions when the parameter δ is set to 0.05. Under such configuration, five hash functions will be invoked on the arrival of each packet. NitroSketch applies sampling on traffic and conducts one hash calculation every 20 packets on average. Therefore, NitroSketch is less computation-intensive compared with HashParallel and CM-Sketch. As a contrast, our approach does not rely on the flow table to maintain stateful counters to identify the heavy hitters thus does not need any hash computation. The event-based detection is stateless with minor computation overhead.

Fig. 10 shows the data structure memory overhead in a single device under the five approaches. NetFlow needs to maintain a flow table of about 80MB with sampled flow items and their counters. HashParallel needs to maintain a much smaller flow cache of around 80KB with most frequently

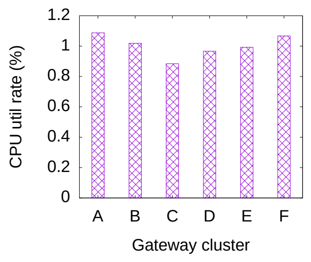

Fig. 7. Additional CPU consumption in control plane to collect gateway information in different clusters.

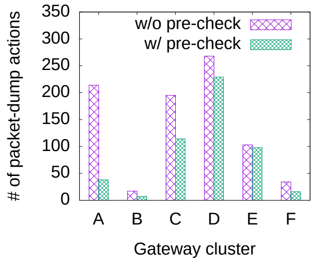

Fig. 8. The comparison of packet dump actions between w/ and w/o pre-check for one month in different clusters.

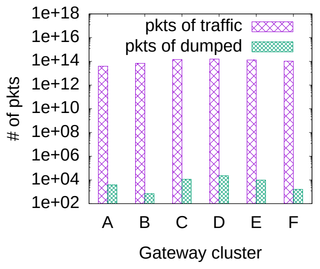

Fig. 9. The comparison between traffic and dumped pkts for one month in different clusters. (Y-axis is in logscale)

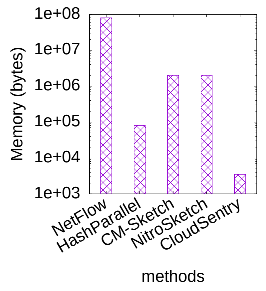

Fig. 10. Memory overhead in a single device. (Y-axis is in logscale)

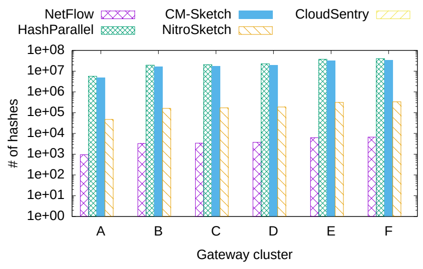

Fig. 11. Hash computation overhead per device per second in different clusters. (Y-axis is in logscale)

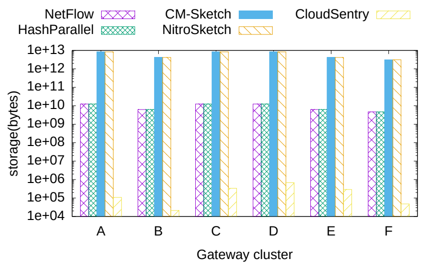

Fig. 12. Storage overhead in different clusters for one month. (Y-axis is in logscale)

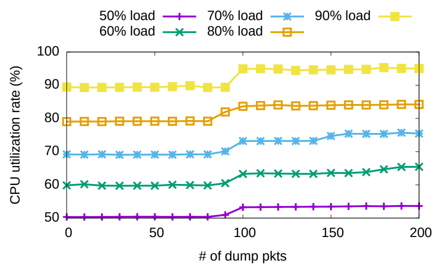

Fig. 13. Dumping packets when CPU utilization rate $<= 90\%$.

accessed flow items and their counters. CM-Sketch and NitroSketch has the similar sketch data structures consuming the similar amount of memory of around 2MB. The data structure size depends on the error rate. While our approach needs to store a list of only 80-100 packets each time triggered by the CPU high utilization. Under the same configuration with that in Fig. 11, our approach has the lowest memory occupation among the five approaches.

In Fig. 12, we measure the storage consumption in the six clusters for one month under the five approaches. Here, data storage is allocated to periodically record the instant heavy hitters once in every 10s. The heavy-hitter logs are maintained for at least one month in cloud service providers for on-demand retrieval, analysis and statistics. The storage will consume huge amount of disks which come with additional cost for acquisition and maintenance. For example, NetFlow and HashParallel dump the top 100 heavy hitters with a storage overhead of 10GB on average per month. While CM-Sketch and NitroSketch have to dump the entire sketch data structures to the disks since the sketches maintain no flow identifiers and we have to record the entire data structure for flow item membership query. They consume around 10TB each month. By contrast, our approach consumes very little storage overhead of around 100KB in a month since what we have to do is to dump 80-100 packets once in every CPU spike event. To summarize, our approach radically reduces the expense for exception logging in cloud-scale service gateways.

E. CPU Contention During Packet Dumping

When triggered by the event of CPU spikes, the packet dump operation will further consume additional CPU cycles in data plane. We evaluate the data plane CPU utilization change in Fig. 13 and Fig. 14, to check whether dumping packets will drastically affect the CPU utilization in data plane, which will further affect the packet forwarding latency. Furthermore, our system will rate limit the number of sampled packets to 100 since 100 packets is good enough for fine-grained heavy hitter detection in our cloud environment.

Fig. 13 shows the impact of packet dump on CPU utilization rate when the CPU utilization $<= 90\%$. We measure with different traffic load levels from 50% load to 90% load. It is indicated that the increase of dumped packets per second will not drastically affect the CPU utilization, which means the design and implementation of the packet dump mechanism of our system is lightweight and sustainable for handling huge amount of concurrent flows at cloud gateways. However, when the packet dump speed hits the rate limit threshold, the CPU utilization has an obvious jump. According to our debugging and estimation, it is the rate limiting logic itself consumes a fraction of CPU processing power, because rate limiting is a per-packet operation executed in high frequency in data plane.

Fig. 14 shows the impact of packet dump on CPU utilization rate when the CPU utilization $= 100\%$. Likewise, we can draw the similar conclusion with the previous Fig. 13. The y-axis of Fig. 14 shows the traffic processing ability after normalization. It is indicated that before rate limiting is invoked, the traffic processing rate is at the peak and rather steady. The execution of rate limiting will consume a fraction of CPU cycles and lower the traffic processing throughput accordingly.

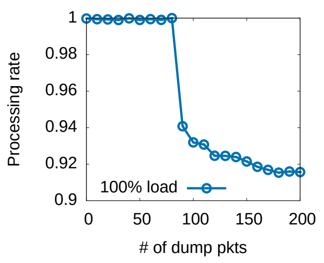

Fig. 14. Dumping packets when CPU utilization rate = 100%. (The processing rate has been normalized.)

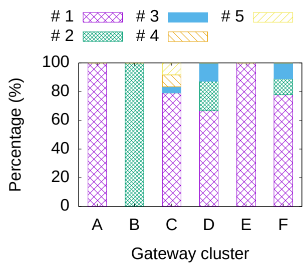

Fig. 15. Overloaded CPU core #'s distribution when CPU utilization exceeds 90% in different clusters.

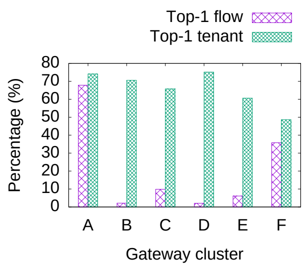

Fig. 16. The proportion of Top-1 flow and Top-1 tenant on average in different clusters.

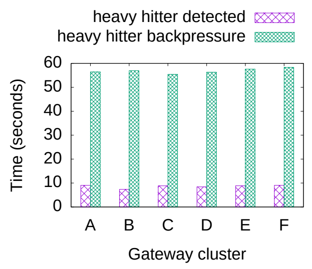

Fig. 17. Time consumed for heavy hitter detected and start to backpressure in different clusters.

F. Dominant-Tenant Effect

Fig. 15 shows the number of overloaded CPU cores distribution with CPU utilization exceeds 90% in six gateway clusters for one month. For cluster-A and cluster-E, all the overloaded CPU cases affect only 1 CPU core. For cluster-B, all the overloaded CPU cases affect 2 CPU cores. For cluster-C, cluster-D and cluster-F, due to high-throughput incoming traffic and traffic load balancing between CPU cores, more than one CPU core is affected in some overloaded CPU cases. According to the above measurement, we can infer that the number of overloaded CPU cores also follows the long-tail distribution that one CPU core overloaded occupies most of the cases.

We take a deeper look into the dataset for finding the root cause of the long-tail property and count the proportional top-1 flow and top-1 tenant in different clusters during CPU overloading in Fig. 16. Here, we find an interesting observation about the root cause of CPU overloading. We name it "dominant-tenant effect" because we find that compared with heavy-hitter flows, the heavy-hitter tenant is much more like to become the root cause of CPU overloading. In Fig. 16, the top-1 tenant occupies higher percentage of CPU overloading than the top-1 flow in all clusters, especially in cluster-B, C, D, E. That is to say, it is more easily to identify the heavy-hitter traffic using the tenant IDs rather than using the traditional 5-tuple flow IDs. The reason behind is that the flows generated from the same tenant are highly correlated and the tenant ID is more likely to aggregate the heavy-hitter flows. While the flows between tenants are totally independent. When identify the heavy-hitter tenant, we can directly rate limit the outlier traffic from that tenant.

G. Heavy Hitter Detection and Backpressure Response Time

When the CPU overloading event occurs, our system needs to verify the persistance of the CPU overloading for at least 2 seconds with distributed messaging and processing before the heavy-hitter confirmation. After that, the system will prepare the background resource to analyze and start to backpressure the outlier flows. In both stages of CloudSentry, the algorithms are simple and should be implemented in a short time, *e.g.*, several seconds. However, in our real deployment, the

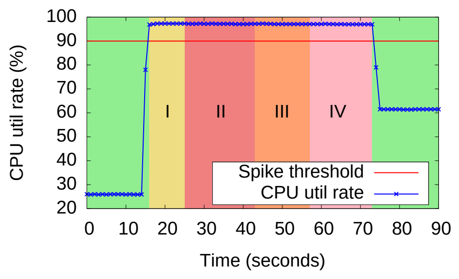

Fig. 18. An experiment of heavy hitter detection in test environment. I: Heavy hitter detected(First stage). II: Packet dump(Second stage). III: Top-flow analysis(Second stage). IV: Comprehensive analysis(Second stage)

detection and backpressure consumes longer time than naive expectation.

Fig. 17 shows the time consumed on average for heavy hitter detection (First Stage) and the corresponding backpressure response (at the end of Second Stage) in six online gateway clusters. The time for heavy-hitter detection is generally below 10 seconds while it takes nearly 1 minute to start to backpressure the heavy-hitter flows. The longer time is because a number of cloud components are employed, such as log storage service and database service. Using these standard cloud components have two advantages. First, we can speed up the development and deployment of our system. Second, we can increase the stability of our system by leveraging the fault tolerance mechanism of cloud components. An optimization is easy to employ for a more time-sensitive heavy hitter detection by cutting off these cloud components.

Fig. 18 shows a time line of a heavy-hitter detection experiment. We use one VM to send background traffic and heavy-hitter traffic to a gateway. All the traffic are received and processed by one core of the gateway. The heavy hitter detection and backpressure process takes 58 seconds, and the first stage, packet dump, top-flow analysis, comprehensive analysis takes 9, 18, 14, 17 seconds respectively. The background traffic consumes 26% CPU utilization. When the heavy hitter occurs, the CPU utilization reaches (approximate) 97%. After the backpressure is implemented, the CPU utilization drops to (approximate) 60%, by halving the heavy-hitter traffic rate. Note that our system only rate limit the very long duration persistent large flows to protect the QoS of other normal flows passing through the service gateways.

VI. Experiences Learned

In this section, we discuss the deployment experiences of CloudSentry in nearly one year.

Low overhead and robustness. The heavy-hitter detection should be low overhead and robustness in online cloud network environment. First, the resources in data plane (*e.g.*, memory, CPU, bandwidth) are dedicated to forward packets, leaving a small fraction of resources to do additional tasks (such as heavy-hitter detection). Therefore, the heavy-hitter detection should be low overhead in data plane, avoiding interfering packet forwarding. Second, it can not be assumed that the resources for data processing (*e.g.*, storage, database, stream computing) are unlimited. Cloud providers have plenty of big-data processing systems. Intuition tells us that we can use big data resources at will. However, experiences told us that it is a big waste for storing and processing big data. Not only waste a lot of budget, but also waste precious time for localizing network anomalies. Third, the heavy-hitter detection should be robustness. It is used to protect the network from impact by critical persistent burst. Therefore, the detection should keep working at some extreme conditions. In our system, we employ several cloud components to build the heavy-hitter detection. Though the detection system is not very time-sensitive, it is very robust to complex network conditions. Some methods [9], [36]–[38] are designed to achieve high accuracy at the cost of CPU, memory, or storage overhead. Therefore, they are hard to deploy in a large-scale production cloud network.

Dominant-tenant effect for CPU spikes. According to our evaluation (in Fig. 16), most CPU spikes are caused by one dominant tenant, both in incast and elephant-flow scenarios. It implies that the traffic among multiple tenants is totally independent. Although the resource of the cloud network is shared, behaviors of different tenants are isolated in most cases. CPU spikes caused by one tenant will not trigger heavy hitters from other tenants. Therefore, finding the dominant tenant is very important in cloud heavy-hitter detection. We achieve this idea by aggregating the captured traffic at the tunnel-id granularity. Once the dominant-tenant is detected, we will protect our cloud gateways by backpressure the dominant-tenant's traffic.

Impact of persistent heavy hitter is larger. Persistent heavy hitters can impact more tenants, incurring more packet losses and more economic losses, compared to transient heavy hitters (*aka*, micro-bursts). Most tenants are insensitive to micro-bursts because their applications can tolerate small network jitters (*e.g.*, fast retransmission of TCP stack). Additionally, micro-bursts can be solved by some fault tolerance mechanisms, *e.g.*, by congestion control [39] and traffic shaping [40]. In contrast, persistent heavy hitters are much more annoying. Within our operational experiences, one persistent heavy-hitter event may cause thousands of tenants suffering the degradation of network performance.

VII. Related Work

Sampling-based heavy hitter detection. A straightforward way for heavy hitter detection is using port mirroring or traffic splitter to collect a complete copy of traffic for remote identification [41]. For cloud gateway traffic of huge volume, the line-rate traffic gathering and identification can be costly. NetFlow [14] and sFlow [15] take an alternative approach by sending a sampled subset of network traffic to the remote collector for analysis. Although packet sampling greatly reduces the data collection and packet processing overhead, NetFlow's aggressively low sampling rate in practice [42] affects the heavy hitter detection accuracy. As a comparison, our approach also analyzes a small subset of network traffic with minor performance overhead. However, the analysis is triggered on demand by high CPU utilization with more accuracy rather than aimless probabilistic sampling.

Counter-based heavy hitter detection. Counter-based heavy hitter detection [36], [43]–[46] maintains a table of flows and corresponding counters for heavy hitter measurement. Generally, maintaining a complete table for all the flows incurs huge memory overhead on high throughput links. To overcome the limitation, these approaches only allocate a bounded-size flow cache for the largest flows with a trade-off between memory consumption and measurement accuracy, where more memory promotes the accuracy. The flow table is updated on a per-packet frequency and each update will further produce multiple memory accesses for flow table scanning or minimum item locating [45]. As a comparison, our approach is much less computation-intensive and there is no need to explicitly maintain the flow cache due to the hierarchy-based detection mechanism according to CPU utilization.

Sketch-based heavy hitter detection. Sketch-based heavy hitter detection [10]–[12], [16], [17] maintains implicitly shared counters between flows and does not explicitly store the flow identifers. Such slim data structures greatly reduce the memory space overhead and make it possible to measure all the concurrent flows. However, these approaches also have known limitations. First, sketch-based approaches are even more computation-intensive compared with counter-based approaches because multiple hashing calculations are invoked on the arrival of each packet. Second, the probabilistic data structures also expose trade-offs between space and accuracy. One cannot achieve high accuracy and low memory cost at the same time. Last but not least, these approaches do not track the flow identifers, making it difficult to extract the keys and the corresponding counters from the sketch. Therefore, it is hard to enumerate or perform statistics about the heavy hitters. Our approach addresses the above limitations with small log size.

VIII. Conclusion

In this work, to ensure the high availability of multi-tenant cloud service gateways, we propose a two-stage heavy-hitter detection approach CloudSentry. CloudSentry is triggered by gateway CPU spikes in coarse-grained stage, then identify and rate limit the outliers in fine-grained stage. Different from existing heavy-hitter detection approaches based on per-packet tracking using either explicitly maintained or implicitly shared counters, our approach starts fine-grained detection only when high CPU utilization occurs and persists for a predefined period of time. Our approach is cost-effective and nearly stateless, which eliminates the massive memory footprints and CPU cycles once needed for heavy hitter detection on millions of concurrent flows at the cloud gateways. CloudSentry has been deployed in production for nearly one year, protect cloud providers from being impacted by persistent heavy hitters.

References

[1] "Alibaba cloud," https://www.alibabacloud.com/.

[2] "Microsoft azure," https://azure.microsoft.com/.

[3] "Amazon web services (aws)," https://aws.amazon.com/.

[4] P. Kumar, N. Dukkipati, N. Lewis, Y. Cui, Y. Wang, C. Li, V. Valancius, J. Adriaens, S. Gribble, N. Foster *et al.*, "Picnic: predictable virtualized nic," in *Proceedings of ACM SIGCOMM*, 2019.

[5] K. Qian, S. Ma, M. Miao, J. Lu, T. Zhang, P. Wang, C. Sun, and F. Ren, "Flexgate: High-performance heterogeneous gateway in data centers," in *Proceedings of APNet*, 2019.

[6] M. Zhang, J. Bi, K. Gao, Y. Qiao, G. Li, X. Kong, Z. Li, and H. Hu, "Tripod: Towards a scalable, efficient and resilient cloud gateway," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 570–585, 2019.

[7] J. Son, Y. Xiong, K. Tan, P. Wang, Z. Gan, and S. Moon, "Protego: Cloud-scale multitenant ipsec gateway," in *Proceedings of USENIX NSDI*, 2017.

[8] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of ACM SOSR*, 2017.

[9] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "Nitrosketch: Robust and general sketch-based monitoring in software switches," in *Proceedings of ACM SIGCOMM*, 2019.

[10] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[11] M. Chen, S. Chen, and Z. Cai, "Counter tree: A scalable counter architecture for per-flow traffic measurement," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1249–1262, 2016.

[12] T. Li, S. Chen, and Y. Ling, "Per-flow traffic measurement through randomized counter sharing," *IEEE/ACM Transactions on Networking*, vol. 20, no. 5, pp. 1622–1634, 2012.

[13] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding tcp incast throughput collapse in datacenter networks," in *Proceedings of ACM WREN*, 2009.

[14] B.-Y. Choi and S. Bhattacharyya, "Observations on cisco sampled netflow," *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 3, pp. 18–23, 2005.

[15] P. Phaal, S. Panchen, and N. McKee, "Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks," 2001.

[16] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proceedings of ICALP*, 2002.

[17] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proceedings of ACM SIGCOMM*, 2002.

[18] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *Proceedings of USENIX NSDI*, 2016.

[19] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "Ictcp: Incast congestion control for tcp in data-center networks," *IEEE/ACM transactions on networking*, vol. 21, no. 2, pp. 345–358, 2012.

[20] R. B. Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Designing heavy-hitter detection algorithms for programmable switches," *IEEE/ACM Transactions on Networking*, 2020.

[21] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-wide heavy hitter detection with commodity switches," in *Proceedings of ACM SOSR*, 2018.

[22] D. Ding, M. Savi, G. Antichi, and D. Siracusa, "An incrementally-deployable p4-enabled architecture for network-wide heavy-hitter detection," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 75–88, 2020.

[23] M. T. Arashloo, P. Shirshov, R. Gandhi, G. Lu, L. Yuan, and J. Rexford, "A scalable vpn gateway for multi-tenant cloud services," *ACM SIGCOMM CCR*, vol. 48, no. 1, pp. 49–55, 2018.

[24] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM CCR*, vol. 44, no. 4, pp. 27–38, 2014.

[25] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of ACM SIGCOMM*, 2017.

[26] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure accelerated networking: Smartnics in the public cloud," in *Proceedings of USENIX NSDI*, 2018.

[27] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *Proceedings of USENIX NSDI*, 2018.

[28] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch," in *Proceedings of USENIX NSDI*, 2015.

[29] S. K. Garg, A. N. Toosi, S. K. Gopalaiyengar, and R. Buyya, "Sla-based virtual machine management for heterogeneous workloads in a cloud datacenter," *Journal of Network and Computer Applications*, vol. 45, pp. 108–120, 2014.

[30] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[31] D. Shan and F. Ren, "Improving ecn marking scheme with micro-burst traffic in data center networks," in *Proceedings of IEEE INFOCOM*, 2017.

[32] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo, "Micro-burst in data centers: Observations, analysis, and mitigations," in *Proceedings of IEEE ICNP*, 2018.

[33] M. Mahalingam, D. G. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks." *RFC*, vol. 7348, 2014.

[34] V. Dukic, G. Khanna, C. Gkantsidis, T. Karagiannis, F. Parmigiani, A. Singla, M. Filer, J. L. Cox, A. Ptasznik, N. Harland *et al.*, "Beyond the mega-data center: networking multi-data center regions," in *Proceedings of ACM SIGCOMM*, 2020.

[35] D. Intel, "Data plane development kit," 2014.

[36] G. Cormode and M. Hadjieleftheriou, "Methods for finding frequent items in data streams," *The VLDB Journal*, vol. 19, no. 1, pp. 3–20, 2010.

[37] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proceedings of ACM SIGCOMM*, 2018.

[38] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, and X. Li, "Heavykeeper: An accurate algorithm for finding top-k elephant flows," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, 2019.

[39] G. Kim and W. Lee, "Absorbing microbursts without headroom for data center networks," *IEEE Communications Letters*, vol. 23, no. 5, pp. 806–809, 2019.

[40] A. Goswami, K. K. Pattanaik, A. Bharadwaj, and S. Bharti, "Loss rate control mechanism for fan-in-burst traffic in data center network," *Procedia Computer Science*, vol. 32, pp. 125–132, 2014.

[41] A. Callado, C. Kamienski, G. Szabó, B. P. Gero, J. Kelner, S. Fernandes, and D. Sadok, "A survey on internet traffic identification," *IEEE communications surveys & tutorials*, vol. 11, no. 3, pp. 37–52, 2009.

[42] V. Carela-Español, P. Barlet-Ros, A. Cabellos-Aparicio, and J. Solé-Pareta, "Analysis of the impact of sampling on netflow traffic classification," *Computer Networks*, vol. 55, no. 5, pp. 1083–1099, 2011.

[43] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proceedings of VLDB*, 2002.

[44] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Transactions on Database Systems (TODS)*, vol. 28, no. 1, pp. 51–55, 2003.

[45] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proceedings of ICDT*, 2005.

[46] R. B. Basat, G. Einziger, R. Friedman, and Y. Kassner, "Randomized admission policy for efficient top-k and frequency estimation," in *Proceedings of IEEE INFOCOM*, 2017.