Anomaly Detection for Black Box Services in Edge Clouds Using Packet Size Distribution

Marcel Wallschläger, Anton Gulenko, Florian Schmidt, Alexander Acker, Odej Kao
Complex and Distributed IT-Systems
{firstname}.{lastname}@tu-berlin.de

***Abstract*—Future services in fields like autonomous driving and virtual reality rely on cloud computing resources located at the edge of internet service provider(ISP) networks. Instead of deploying many service-specific monitoring and reliability platforms, a centralized monitoring solution can reduce the usage of the already sparse edge cloud resources. The ISP can offer such a service using the black box monitoring approach presented in this paper. Current cloud providers already collect data about customer services for cloud performance and cloud reliability. We propose to extend current monitoring solutions for virtual machines by real-time analysis of network packet headers. In particular, we use the packet size distribution and the TCP connection time to infer the operational state of the service. We conduct an evaluation of the presented approach using a content delivery system which is set into different load and anomaly states. The random forest algorithm trained to differentiate normal from abnormal service states based on the collected data resulted in an accuracy of 94%. The overhead of collecting the data on a commodity hardware hypervisor using eBPF is about 3% CPU at 10GB/s.**

***Keywords*—edge cloud; cloud networks; monitoring; network function virtualization**

I. Introduction

The future mobile network standard 5G is getting closer to market, and with the release of the new broadband technology, many technologies that are currently limited to research applications, will be available for industrial and private users. This includes technologies from fields like the Internet of Things (IoT), autonomous driving, Industry 4.0 and virtual reality (VR). A related phenomenon which is of special interest for internet services providers (ISP) is the need for compute resources located closer to customer applications. For an autonomously driving car it might not be acceptable to have a backend server located in the cloud as the connection between the car and the server would exhibit an unacceptably high latency. These problems are tackled by edge computing.

Edge computing provides elastic computing resources similar to a cloud, but closer to the customer applications. Such edge clouds could be located at broadband base stations or access networks of ISPs. An access network today is a small data center, where the different internet access technologies like ISDN, DSL, vDSL, broadband etc. are converted to IP traffic by vendor specific appliances. With the rising interest in Network Function Virtualization (NFV) in the telco domain, these appliances are replaced by a combination of commodity hardware and open source software.[1].

In the near future companies offering virtual reality applications might want to have parts of for their server infrastructure run with very low network latency. Low latency is important for virtual reality applications as lagging movement of the virtual surrounding frequently leads to headaches and nausea for users. To overcome this problem the VR company can place parts of their server infrastructure into the access networks of the ISPs. However, due to the favorable positioning, resources in edge clouds are costly and limited in their capacity. Therefore, many companies in need of edge resources will not be able to deploy their entire chain of monitoring and reliability components as they would do in a regular public cloud. This enables the ISP to offer a new class of services for monitoring and detection of faults and anomalies. Of course such services still have to comply with the Service-Level-Agreements (SLA) between the ISP and their customers. The SLAs prohibit the ISP from directly accessing their customers computing resources and stored data, which means that the mentioned monitoring services must treat the customer software as black boxes.

This paper proposes a novel approach to infer the operational state of virtualized services without access to service specific data and without directly communicating with the service. The proposed monitoring techniques need to provide enough information to gain knowledge about the current status of a *compute unit* that hosts a customer service. Such a compute unit could be a docker container or a virtual machine (VM), depending on the deployment model of the edge cloud. The core of the proposed approach is to collect statistics from TCP/IP headers of outgoing and incoming network traffic of a monitored service. The resulting metrics include packet size distributions, statistics about TCP-connection durations and other TCP-header related metrics.

The rest of this paper is organized as follows. Section II summarizes different sources of data that can be used to infer the operational state of virtualized services. Section III describes the proposed approach in detail, while section IV presents an evaluation of the expressiveness of the collected metrics and the introduced resource overhead. Finally, section

[1]https://portal.etsi.org/nfv/nfv_white_paper.pdf [Accessed: 01-Jun-2018]

V concludes and proposed concepts for further development of the presented approach.

II. Monitoring Black Box Services

The technological stack involved in virtualizing services is highly complex and consists of a multitude of heterogeneous components. Even without direct access to service specific runtime information, many management components are involved in executing and connecting the services and can therefore be queried for information about their operational state.

The most basic source of information is the Infrastructure as a Service (IaaS) platform used to host and manage all virtualized services. The cloud platform contains a repository of static information about all virtual machines [1]. Available metrics include the resources allocated for a given VM, networking information like IP addresses or routing tables, the physical location of the compute resources, and finally network access rules. This information can be analyzed to create a model of the service dependency graph, enriched with meta information about all contained VMs and virtual network elements. As this data is mostly static, however, it contains little knowledge about the runtime state of the hosted service. While high-level configuration issues can be detected and automatically mitigated, actual runtime performance anomalies like overloads or memory leaks must be detected by different means.

The component closest to a virtual machine is the hypervisor. It can deliver various resource and runtime state metrics like consumption of CPU, memory or network resources or the state of virtual machines. KVM, for example, can provide such metrics in a very high sampling rate [2] [3], but other virtualization engines like Xen have similar interfaces. Many performance anomalies like overload situations, memory leaks or irregular usage of certain resources, can be detected in such metrics [4]. Some services, however, do not exhibit a consistent resource usage for different operational states. A misconfigured web server, for example, might behave normally from a resource point of view, but reply with 403 or 500 status codes for requests that are usually successful.

Services that are based on a request-reply model usually manifest abnormal states in unexpected communication patterns. By observing the network traffic of services, anomalies can be detected in the distribution of different protocol specific counters like status codes of HTTP or SIP replies. This approach is based on deep packet inspection (DPI) [5], which requires that the SLAs of the cloud provider allow such an analysis. Furthermore, this approach is limited to a fixed number of supported network protocols, requires unencrypted traffic and usually special hardware for inspecting network packets at line speed.

These problems can be avoided by observing the network traffic on lower levels of the OSI layer model. One approach is to maintain statistical distributions of sizes and properties of Internet Protocol (IP) packets. This approach is widely used in a number of research and application fields of computer science.

Intrusion detection in computer networks is often based on packet size distribution [6] [7]. The most common approach is to maintain a pattern database for different types of intrusions. All historically or empirically known attack patterns are stored in the pattern database and observed network traffic is continuously compared against stored patterns. Contrary to intrusion detection, the approach presented in this paper does not aim to detect malicious actions, but rather anomaly situations that originate in software bugs, misconfigurations, overload situations and other unintended scenarios. However, the use of packet size distributions is common for both approaches. Denial of Service (DOS) have been shown to manifest in changes of packet size distributions as well [8]. [9], [10], [11] show that TCP/IP header statistics also suite for anomaly detection. The authors show an approach for detecting anomalies on the network layer.

III. Approach

The goal of the approach presented in this paper is to collect information from possibly encrypted black box service network traffic without violating data protection SLAs. Additional requirements include handling of encrypted traffic and to use less resources than deep packet inspection. Only using IP packet headers as seed for additional network metrics is more computational lightweight than parsing the entire packet payload. The most important question is whether the packet headers on the lower protocol layers contain enough information for service layer anomaly detection. Previous work suggests that packet header statistics represent enough information for detecting network layer anomalies [12]. We apply a similar methodology to the field of anomaly detection on the service layer.

The presented data collection approach is based on observing every network packet going into or sent from an observed virtual machine. Since all network traffic of the virtual machine goes through the associated tap interface provided by the hypervisor, the necessary packets can simply by sniffed from that virtual network interface. If a real-time data collection is not required, the captured packets can simply be written to a PCAP file for offline analysis. However, anomaly detection usually has the requirement to run in real-time. Therefore the captured packets must be analyzed online by injecting data collection code close to the virtual switching unit (often Open vSwitch) or by attaching an eBPF program [13] to the tap-interface of the virtual machine. Figure 1 shows a common network stack of a cloud operating system such as OpenStack. Virtual machine instances are black box customer instances with virtual network interfaces. Below, the `tap*` interfaces illustrate how the virtual machine is connected to the virtual switching domain. This example contains Open vSwitch, as it is the de facto standard virtual switch for OpenStack and NVF related use-cases. After traversing the security groups, which implement the firewall functionality of OpenStack, the traffic of the VM through a number of virtual network elements configured inside of Open vSwitch. The lowest layer is finally

connected to a physical network interface on the hypervisor node.

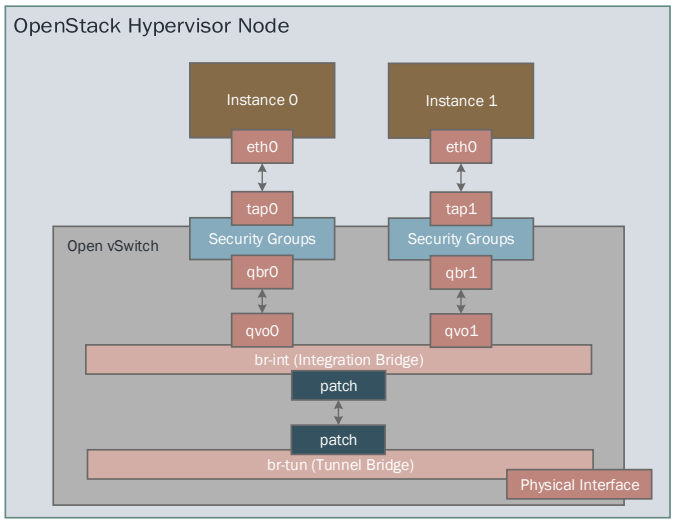

Fig. 1. Common virtual network stack in a cloud environment.

Two underlying traffic properties are extracted from the captured packets as the basis for all collected metrics. The first one is the duration of the TCP-Connections established with the observed VM, the second one is the size of each packet in bytes. All metrics are sampled and reported regularly in a configurable time interval. Regardless of the chosen sampling interval, all packet headers must be examined to obtain the proposed metrics. Therefore, the sampling interval can be chosen arbitrarily and the performance overhead of a high sampling frequency only manifests in the mechanism used to transport the collected data, like writing it to a file or sending it over the network.

Table I shows the complete list of the collected metrics. The metrics *opened_connections*, *stalled_connections*, *closed_connections* and *ongoing_connections* are implemented by observing the TCP handshake for opened connections and the TCP close sequence for closing connections. Additionally, a threshold can be defined to close stalled connections. Stalled connections are defined as connections which have not yet been closed by a TCP handshake, but also have not produced any packets within a fair amount of time. *bytes_in* and *bytes_out* are the summation of all packets going into the VM or coming from the VM. These trivial metrics can also be obtained from other sources mentioned in section II. They are still included in the metric set of this approach, because these basic metrics have a high impact on the prediction power for service related anomalies. Since collecting these metrics does not introduce any additional overhead, they make this approach usable on its own without relying on other data sources. Finally, it could be interesting to compare the sampled values with the values reported by the virtualization engine and the virtual network layer.

The metrics prefixed with *pkt_in_* and *pkt_out_* are based on a histogram statistic of the incoming and outgoing traffic of the VM. Packets of different sizes are counted in a configurable number of buckets. For example a packet sent to the VM with the packet size of 767 bytes, would increase the counter *pkt_in_700-800* by one. Table I shows the buckets of the common MTU of 1500 bytes and a bucket size of 100 bytes. Due to the TCP Segmentation Offloading mechanism, packets with a greater size than the MTU can arrive at the Linux kernel. Therefore, this approach assumes that each packet that is bigger than MTU, has a size of 1500 bytes. In other words, if a packet with a size of 4005 bytes is encountered, the counter of the 1400–1500 byte bucket is increased by two and the 1000–1100 byte bucket by one.

TABLE I
SYSTEM RESOURCES SAMPLED BY THE DATA COLLECTION SERVICE

Metric name	**Description**
opened_connections	Number of opened connections
closed_connections	Number of closed connections
stalled_connections	Number of stalled connections
ongoing_connections	Number of ongoing connections
avg_duration	Average duration of all connections
bytes_in	Overall bytes sent to the VM
bytes_out	Overall bytes sent by the VM
avg_bytes_in	Average size of packets sent to the VM
avg_bytes_out	Average size of packets sent by the VM
	Packets sent to VM with size...
pkt_in_0-100	0 – 100 bytes
pkt_in_100-200	100 – 200 bytes
pkt_in_200-300	200 – 300 bytes
pkt_in_300-400	300 – 400 bytes
pkt_in_400-500	400 – 500 bytes
pkt_in_500-600	500 – 600 bytes
pkt_in_600-700	600 – 700 bytes
pkt_in_700-800	700 – 800 bytes
pkt_in_800-900	800 – 900 bytes
pkt_in_900-1000	900 – 1000 bytes
pkt_in_1000-1100	1000 – 1100 bytes
pkt_in_1100-1200	1100 – 1200 bytes
pkt_in_1200-1300	1200 – 1300 bytes
pkt_in_1300-1400	1300 – 1400 bytes
pkt_in_1400-1500	1400 – 1500 bytes
	Packets sent by VM with size...
pkt_out_0-100	0 – 100 bytes
pkt_out_100-200	100 – 200 bytes
pkt_out_200-300	200 – 300 bytes
pkt_out_300-400	300 – 400 bytes
pkt_out_400-500	400 – 500 bytes
pkt_out_500-600	500 – 600 bytes
pkt_out_600-700	600 – 700 bytes
pkt_out_700-800	700 – 800 bytes
pkt_out_800-900	800 – 900 bytes
pkt_out_900-1000	900 – 1000 bytes
pkt_out_1000-1100	1000 – 1100 bytes
pkt_out_1100-1200	1100 – 1200 bytes
pkt_out_1200-1300	1200 – 1300 bytes
pkt_out_1300-1400	1300 – 1400 bytes
pkt_out_1400-1500	1400 – 1500 bytes

IV. EVALUATION

A. Experimental Setup

An evaluation of the information content of the described approach was conducted on a private cloud testbed especially deployed for that purpose. The service layer use case was a content delivery system with about 6000 items served over the HTTP protocol. This load generation service was running

inside an instance of `apache2`, the widely used Linux web server.

The virtual machine hosting this web server had 2GB of memory and 2 CPU cores. The overall content available through this content delivery system was about 1850MB. Using the built-in caching mechanism, the web server kept all the content in memory. This allows the web server to answer requests faster and with a lower CPU and disk usage overhead. This web server VM simulates a black box service running in an edge cloud close to the customer.

Two client virtual machines were deployed to generate load on the web server. Each of the load generation machines was running 80 load generator instances. Each of these load generator instances had access to a full list of items available through the content delivery system. From this list they randomly selected one item to request from the web server at a time. After one item was finished, the next item was selected and requested. This simple load generation mechanism was used as simulation of a wider range of traffic patterns that can be encountered in edge clouds.

During the evaluation experiments, the average throughput on the network interface of the web server was around 220Mbit/s. The resulting traffic was captured and later analyzed and evaluated offline. For the purpose of traffic capture, the `tcpdump` tool was used on the hypervisor hosting the web server VM.

To evaluate the informational content of the extracted metrics the evaluation VM was put into different states. Most of the states simulate anomaly behavior inside the black box edge service, in this case the web server. Two others simulate network anomalies on other layers within our edge cloud. The following list summarizes all states used for this evaluation:

- **normal:** The service is operating normally.
- **miss-cont-5:** 5% of the content is deleted, clients still try to request data that cannot be delivered by the content delivery system. Web server response: HTTP 404.
- **miss-cont-10:** Like the previous anomaly, but the percentage of missing content is increased to 10%.
- **stress-mem:** An additional process is executed, allocating and holding large amounts of memory. This forces `apache2` to use swap. The response time will increase significantly.
- **stress-cpu:** An additional process is executed with an excessively high CPU utilization. This should not affect the web server much, because all the content is already loaded into memory.
- **packet-loss:** The *iptables* mechanism is used of the hypervisor to randomly drop 5% of the packets towards and from the virtual machine. This will increase the amount of retransmissions within the network, which are not measured directly.
- **inc-latency:** The *tcfilter* mechanism is used to increase the latency towards and from the virtual machine by 15ms. This decreases the number of requests per second and represents a network layer anomaly.

During the evaluation experiments, each of these states was active for 60 minutes. After collecting the data, the metrics were extracted for every 200ms of stored data.

B. Information Content

Figure 2 shows box plots of all metrics related to connection timing for the listed states. Outliers are excluded from the plots in order to increase the readability. Each x-axis shows the states, while the y-axis of (a), (b) and (c) shows the number of connections opened, closed and ongoing. In (d) the y-axis shows the average connection duration in seconds.

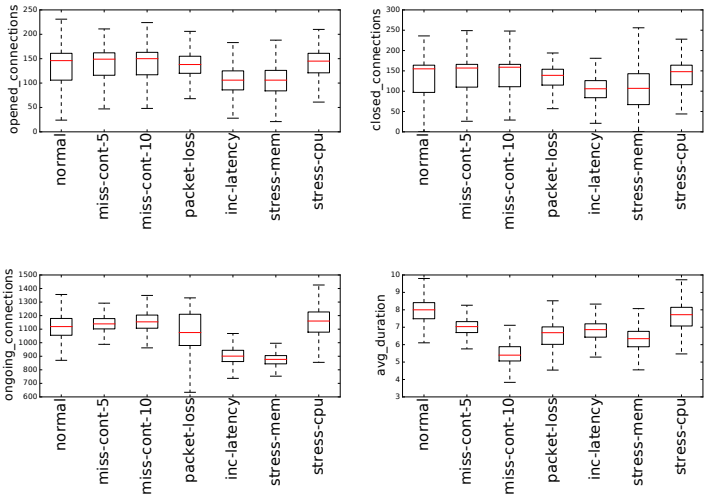

Fig. 2. (a) Opened connections; (b) Closed connections; (c) Ongoing connections; (d) Average duration of ongoing connections

The plots for opened, closed and ongoing connections show similar overall patterns. In these plots, the states *stress-cpu* and *miss-cont-5* and *miss-cont-10* show little deviation from the *normal* state. The *stress-cpu* does not affect the service quality, because the content is already stored in memory. Even the missing content states do not interfere with the amount of established connections, because of the small size of the served content items. On the other hand, the states *stress-mem* and *inc-latency* do reduce the amount of opened connections by around 30% in average. As *stress-mem* starves the web server of access to the memory, the web server takes longer to answer requests. *packet-loss* causes a slight drop in established connections compared to the *normal* state, because retransmissions on the transport layer cause a higher connection time in average. The effects of *stress-mem* and *inc-latency* manifest more severely in the number of ongoing connections.

The average duration of a connection is presented in Figure 2 (d). Again, the *stress-cpu* state does not deviate much from the *normal* state, while the other states generally exhibit shorter connection times. The default keep-alive timeout used by the `apache2` web server, which was also used for this evaluation, is 5 seconds[2]. For the missing content states

[2]http://httpd.apache.org/docs/2.2/mod/core.html [Accessed: 01-Jun-2018]

we assume that sending a HTTP 404 response will make `apache2` automatically close the connection.

Overall, the metrics about connection times clearly differentiate between most anomaly states. The states *stress-mem* and *inc-latency* can be especially well differentiated.

Figure 3 shows box plots of four additional selected metrics. The plot (a) shows the average byte size sent by the virtual machine for each of the defined states. Plot (b) (c) and (d) show the number of packets in the buckets of 500–600 byte, 1000–1100 byte and 1300–1400 byte, respectively for the different states. Outliers are again omitted from the box plots in order to increase readability.

Since the content items provided by the `apache2` web server are text pages and images, the average packet size shown in Figure 3 (a) is quite small. Again, the *stress-cpu* state of this plot shows a similar pattern to the *normal* state. This shows also that the *stress-cpu* anomaly state has no influence on the packet size. The *miss-cont-5* and *miss-cont-10* anomalies reduce the average packet size significantly. The packet loss, on the other hand, causes an increased packet size.

In Figure 3 (b) we observe a significant increase of small packets sized between 500 and 600 byte for the anomalies that delete parts of the web server content. None of the other states show any significant changes compared to the *normal* state. Similarly, 3 (c) shows that two of the anomaly states have a significant impact on the number of packets sized between 1000 and 1100 byte. The *packet-loss* state increases the number of packets in this bucket, while *inc-latency* reduces the number of these packets to close to zero. This clear distinction can be used to detect and identify these states in a live system. Finally, Figure 3 (d) shows the packet size bucket of 1300–1400 byte. This bucket shows less clear differences between the different anomaly states. The most significant changes for this bucket are the lower number of packets for the *miss-cont-5*, *miss-cont-10* and *packet-loss* anomalies. What is also to consider here is the higher number of total packets when compared to the other buckets.

The visualization and analysis presented so far shows that different operational states of black box services influence the packet size distribution in an unpredictable manner. Therefore, a machine learning approach can be leveraged to distinguish the different anomaly states.

C. Prediction Accuracy

A deeper analysis of the information content is provided by training and evaluating a machine learning algorithm model with the collected data. The underlying assumption is: the more expressive the data, the greater the accuracy of the resulting model. Since the goal is not to evaluate the learning algorithm, the absolute accuracy numbers are not as important as the comparison between the different states. For this purpose, we use the random forest model algorithm [14]. Random forest works by constructing multiple decision trees and using the average prediction result of all sub models as the overall result. This approach corrects for decision trees' habit of overfitting to their training set. Before training the random forest model with it, all data was shuffled and then separated into separate test and training data sets. We used 80% of the data as training data and 20% as testing data and repeated this procedure ten times.

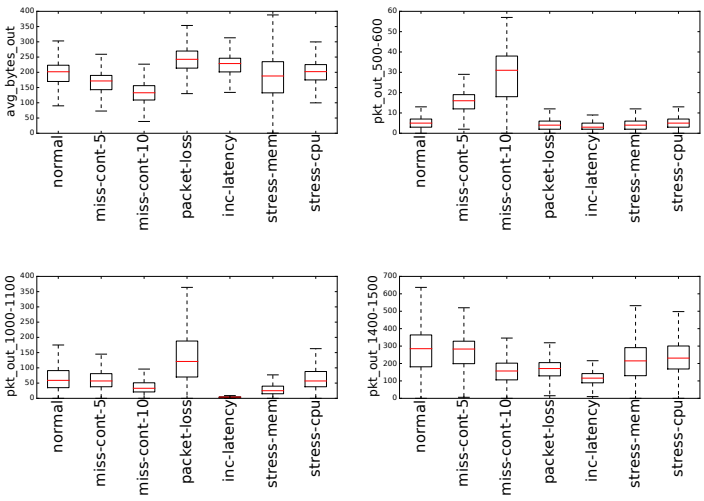

Fig. 3. (a) Average size of packets sent by VM; (b) Bucket 500–600 bytes sent by VM; (c) Bucket 1000–1100 bytes sent by VM; (d) Bucket 1400–1500 bytes sent by VM

TABLE II
PREDICTION RESULTS USING THE RANDOM FOREST ALGORITHM

state name	accuracy
normal	85.7%
missing content 5%	90.9%
missing content 10%	98.3%
stress memory	94.8%
stress CPU	39.6%
packet loss 5%	92.7%
increased latency	98.2%
average	85.8%

Table II shows the results as the average accuracy values of the ten iterations. As discussed in the previous section, some anomaly states produce significant differences in the collected metrics. Especially *inc-latency*, *miss-cont-10* and *stress-mem* have a very high accuracy above 94%. A slightly lower accuracy is achieved for the *packet-loss* and *miss-cont-5* anomalies. The worst prediction accuracy was observed for the *normal* and *stress-cpu* states. These two states showed very similar patterns in all analyzed metrics. As the *stress-cpu* anomaly does not actually affect the service quality, a second evaluation was performed using the same approach as described above, but also treating all samples from the *stress-cpu* state as *normal* samples. With this modification, the average accuracy increased to 93.4%. The prediction results could be further improved by using more advanced prediction models or by combining the metrics presented in this paper with other metrics discussed in section II.

D. Resource Overhead

As resources in an edge cloud are limited, the resource overhead of a monitoring solution is of big significance for real world applications. To evaluate the resource overhead of the presented approach, an online solution was implemented with the extended Berkeley Packet Filter (eBPF). The eBPF Linux kernel mechanism allows users to attach C code to specific kernel functions or network operations [13]. The `bcc` Python bindings for eBPF were used for this evaluation prototype. *BPF_PROG_TYPE_SCHED_CLS* were used to collect the data, Linux `tc filters` were used for ingress and egress separation and *BPF_MAP_TYPE_HASH* were used for communication between kernel space and user space.

The CPU overhead of the prototypical implementation was measured in a small test scenario. This scenario includes two hosts, one running an `iperf3` server, the other one an `iperf3` client and the eBPF monitoring program. `Iperf3` is a networking benchmark tool that can be configured to establish constant network traffic between the server and the client. For this experiment, `iperf3` was configured to use ether 100Mbit/s, 1Gbit/s or 10Gbit/s, each configuration lasting for 100 seconds. The Maximum Transmission Unit (MTU) in the evaluation network was set to 1500 byte. The tool `perf` was used to measure the overhead of the eBPF program. `perf` can output the CPU utilization of all BPF programs under the name *__bpf_prog_run*. To get adequate results only for the implemented eBPF program, the overhead of the *__bpf_prog_run* value was measured with and without it. The baseline CPU utilization of the other BPF programs was subtracted afterwards.

Figure 4 shows the resulting CPU overhead for the different throughput configurations. The CPU utilization starts at 0.15% for 100Mbit/s, grows to 0.9% at 1Gbit/s, and finally reaches 2.9% at 10Gbit/s. These measurements show that the overhead grows sub-linearly with increasing network throughput. A limitation of this evaluation is that the `iperf3` tool only creates and maintains one TCP connection for the entire evaluation duration. In a productive environment like a web server, the number of connections is usually significantly higher. The additional overhead for each new connection is the creation of a data structure for maintaining the counters, and adding this structure to an eBPF map.

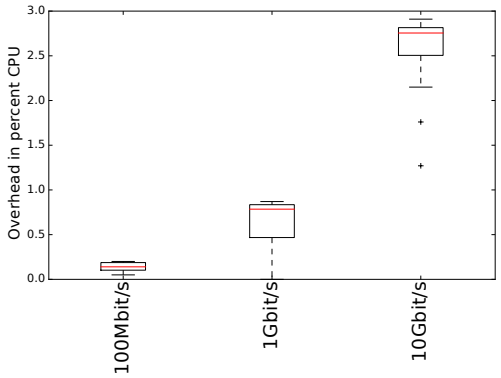

Fig. 4. CPU overhead of the eBPF implementation for different throughputs

V. Conclusion

This paper presents a method for monitoring black box services without access to service specific data or protocols. Capturing network packets sent between a virtual machine and the network allows to construct and maintain statistics about packet sizes and connection times. An evaluation of the information content showed that different anomaly states of a web server clearly manifest in the proposed metrics and a random forest model is able to differentiate the different anomalies with an accuracy of 94%. The computational overhead of the proposed approach was measured at less than 3% of a single commodity hardware core for 10Gbit/s throughput. These evaluations show that the presented approach is suitable to detect abnormal behavior of black box services in real-time, which can be leveraged to offer Anomaly-Detection-as-a-Service for edge cloud applications.

References

[1] J. M. A. Calero and J. G. Aguado, "Comparative analysis of architectures for monitoring cloud computing infrastructures," *Future Generation Computer Systems*, vol. 47, pp. 16–30, 2015.

[2] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, pp. 225–230, 2007.

[3] J. Sugerman, G. Venkitachalam, and B.-H. Lim, "Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor.," in *USENIX Annual Technical Conference*, pp. 1–14, 2001.

[4] A. Gulenko, M. Wallschläger, F. Schmidt, O. Kao, and F. Liu, "Evaluating machine learning algorithms for anomaly detection in clouds," in *IEEE International Conference on Big Data*, pp. 2716–2721, 2016.

[5] M. Wallschläger, A. Gulenko, F. Schmidt, O. Kao, and F. Liu, "Automated anomaly detection in virtualized services using deep packet inspection," *Procedia Computer Science*, vol. 110, pp. 510–515, 2017.

[6] C. Krügel, T. Toth, and E. Kirda, "Service specific anomaly detection for network intrusion detection," in *ACM symposium on Applied computing*, pp. 201–208, 2002.

[7] K. R. Rao, S. K. Battula, and T. L. S. R. Krishna, "A smart heuristic scanner for an intrusion detection system using two-stage machine learning techniques," *International Journal of Advanced Intelligence Paradigms*, vol. 9, no. 5-6, pp. 519–529, 2017.

[8] P. Du and S. Abe, "Detecting dos attacks using packet size distribution," in *Bio-Inspired Models of Network, Information and Computing Systems*, pp. 93–96, IEEE, 2007.

[9] M. V. Mahoney, "Network traffic anomaly detection based on packet bytes," in *Proceedings of the 2003 ACM symposium on Applied computing*, pp. 346–350, ACM, 2003.

[10] N. Duffield, P. Haffner, B. Krishnamurthy, and H. A. Ringberg, "Systems and methods for rule-based anomaly detection on ip network flow," June 13 2017.

[11] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data plane performance diagnosis of tcp," in *Proceedings of the Symposium on SDN Research*, pp. 61–74, ACM, 2017.

[12] N. Duffield, C. Lund, and M. Thorup, "Properties and prediction of flow statistics from sampled packet streams," in *ACM SIGCOMM Workshop on Internet measurement*, pp. 159–171, 2002.

[13] L. Torvalds, "Linux socket filtering aka berkeley packet filter (bpf)," tech. rep., Linux Foundation, 2005.

[14] T. K. Ho, "Random decision forests," in *International Conference on Document analysis and recognition*, vol. 1, pp. 278–282, IEEE, 1995.