Data Log Management for Cyber-Security Programmability of Cloud Services and Applications

Abstract

In last years, the security appliance is becoming a more important and critical challenge considering the growing complexity and diversification of cyber-attacks. The current solutions are often too cumbersome to be run in virtual services and Internet of Things (IoT) devices. Therefore, it is necessary to evolve to a more cooperative models, which collect security-related data from a large set of heterogeneous sources for centralized analysis and correlation.

In this paper, we outline a flexible abstraction layer for access to security context. It is conceived to program and gather data from lightweight inspection and enforcement hooks deployed in cloud applications and IoT devices. We provide a description of its implementation, by reviewing the main software components and their role.

Finally, we test this abstraction layer with a performance evaluation of a Proof of Concept (PoC) implementation with the aim to evaluate the effectiveness to collect data / logs from virtual services and IoT to enable a centralized security analysis.

Keywords

Data Inspection, Log Management, Cyber-Security, Programmability, Cloud

ACM Reference Format:

. 2019. Data Log Management for Cyber-Security Programmability of Cloud Services and Applications. In *CYSARM '19: Workshop on Cyber-Security Arms Race, November 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 6 pages. https://doi.org/https://doi.org/10.1145/3338511.3357351

1 Introduction

Agility and cost-effectiveness in building and operating Information and Communication Technologies (ICT) services are enabled by virtualization in the cloud paradigm. However, unlike current legacy deployments, they pose additional security concerns [11, 12].

Physical and virtual services usually resemble the same development structure. In the Infrastructure-as-a-Service (IaaS) model, a common practice is to deploy each software application in a different virtualization environment, which may be a virtual machine or a software container, and then interconnecting them through virtual network links. This way, the failure of a single virtual machine does not necessary affect the whole service; applications may be easily packaged and delivered as cloud images. [1]

The substantial limitations of security mechanisms in the virtualization infrastructure such as distributed firewalling, micro-segmentation, and security groups [2–4]; the difficulty to coordinate them in cross-cloud deployments; and the typical diffidence in trusting security services provided by third parties favoured an increasing trend to insert legacy security appliances in the topology of virtual services. Per contra, this approach has several issues: *i*) own inspection hooks for each appliance; *ii*) detection requires large amount of computing resources due to the ever-growing number and complexity of protocols and applications; *iii*) difficult to balance the need for pervasive protection with performance of the whole graph; and *iv*) complex security appliances not immune to bugs and vulnerabilities. The first drawback may result in unnecessary duplication of operation: same packets may be processed by different appliances for retrieving similar information. The second one may significantly increase the cost of running the graph. Finally, the last ones eventually increase the overall attack surface of the deployed service.

Considering these aspect, new architectural paradigms are required to build situational awareness for virtual services. This way, it will possible to overcome the above limitations by combining fine-grained and precise information with efficient processing, elasticity with robustness, autonomy with interactivity [12]. A transition from stand-alone security appliances to more cooperative models is, therefore, necessary. For cooperative model, we mean a centralized architecture where security information, data, and events are collected from multiple sources within a given domain for common analysis and correlation. This is a common trend today for all major vendors of cyber-security applications, which are increasingly developing Security Events and Information Management and Security Analytics software for the enterprise, leveraging machine learning and other artificial intelligence techniques for data correlation and identification of attacks. They are usually designed as integration tools of existing security applications and require to run heavyweight processes on each host; hence, they are not suitable for virtual services. In addition, a centralize architecture improves the detection rate while decreasing the overhead on each terminal [10]. On the other hand, security management of service graphs is a very challenging task, since the context continuously changes. Integrating security appliances in service graph design is not the best solution, since it usually requires manual operations; instead, security should be described at a very *abstract* level, by defining policies and constraints that describe *what* is required rather than *how* to implement it.

The general architecture of a novel framework proposed for AddreSsing ThReats for virtualIzeD services (ASTRID) [6] shifts security appliances away from service graph design. In ASTRID, security properties of each graph component as well as the whole service are defined by proper models and policies, which are then used at deployment time to properly configure the execution environment. The developer specifies security requirements and policies for the protection of the graph, without the need for deep technical understanding of the underlying technology. The underlying concept is the de-coupling of inspection tasks to be integrated into the different forms of virtualization boxes – as Virtual Machine (VM) or containers – from a logically, centralized and shared detection logic to be kept outside the graph, as schematically shown in Fig. **??**. In particular, the proposed architecture covers the following aspects: *i*) Increase society's resilience to advanced cyber-security threats. Full control of the underlying packet forwarding policies, hence providing better control and recovery in case of attack; *ii*) Progress in technologies and processes needed to improve organisations' capabilities to detect and respond to advance attacks. Exploit advanced programmability features in virtualised environments, bringing the possibility to duplicate compromised services or functions, to isolate the attack in a fake environment, to restart the service in a safer environment, and more; *iii*) Security control and intrusion prevention systems become more efficient and adapted to new and dynamic environments. ASTRID leverages data plane technologies for fast and efficient monitoring and inspection of packets and software, removing the need for deploying many overwhelming virtual security appliances throughout the service graph; and *iv*) Portability of the security logic. Every orchestration engine has

[1] A cloud image is a bootable software image that already contains a fully functional operating system and some software applications.

CYSARM '19, November 15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6840-7/19/11…$15.00
https://doi.org/https://doi.org/10.1145/3338511.3357351

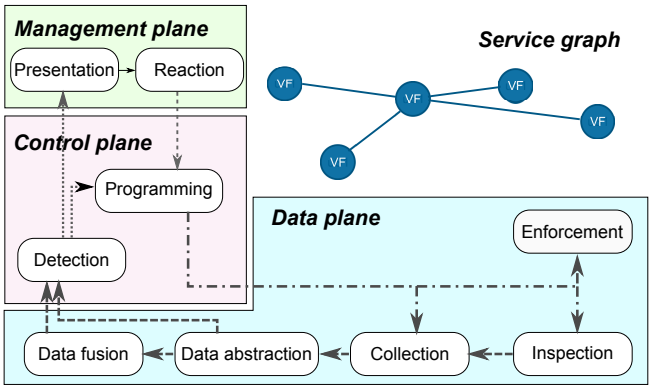


Figure 1: Multi-layer security architecture.

its own graph models and packaging format (e.g., OpenBaton, [2] TeNOR, [3] Arcadia, [4] Juju, [5] OpenStack Heat [6]). If applications are deployed inside the service graphs, different versions must be built and maintained, which complicates the distribution of updates and security patches.

In this paper, we describe the definition of an abstraction layer to provide the detection logic with uniform and *bi-directional* access to heterogeneous security context of virtualized services. The novelty of our work consists in abstracting lightweight *programmable* hooks in the kernel or system libraries, without the need to deploy complex and cumbersome security appliances inside VMs or as separate components in the overall service graph. The ability to program both the collection of security context and the configuration of enforcement rules (which is the mean for *bi-directional* access) is just a major improvement over the number of log [7] collection tools already available as commercial or open-source implementations.

The rest of the paper is organized as follows. We describe the overall ASTRID architecture in Section 2. We then elaborate on the concept of abstraction layer and its architectural design in Section 3, while we discuss the current implementation in Section 4 with a exhaustive description of the chosen technologies. Then, in Section 5, we provide a functional validation and extensive performance evaluation of a Proof of Concept implementation, including integration with local monitoring/enforcement agents. Finally, we give our conclusion in Section 6.

2 The ASTRID architecture

Fig. 1 show the three complementary planes in which ASTRID multi-layer architecture is organized. Although our architecture is not directly related to network operators, we applies network terminologies. ASTRID is a multi-tier architecture, where a common, programmable, and pervasive *data plane* feeds a powerful set of multi-vendor detection and analysis algorithms (*business logic*). On the one hand, the challenge is to assemble a wide knowledge over multiple sites by real-time collection of massive events from a multiplicity of capillary sources, while maintaining essential properties such as forwarding speed, scalability, autonomy, usability, fault tolerance, resistance to compromises, and responsiveness. On the other hand, the ambition is to support better and more reliable situational awareness by inter- and intra-domain data correlation in both space and time, in order to timely detect and respond even the more sophisticated multi-vector and interdisciplinary cyber-attacks.

The *data plane* is the only part of the architecture that is deployed in the virtualization environment. It collects the security *context*, i.e., a knowledge base including events, logs, measures that can be useful for detection of known attacks or identification of new threats.

One of the main advantages of a common control plane is the availability of data from different subsystems (disk, network, memory, I/O), instead of relying on a single source of information as is the common practice nowadays. Since the collection of data from multiple sources may easily result in excessive network overhead, it is important to shape the inspection, monitoring, and collection processes to the actual need. The data plane must therefore support re-configuration of individual components and programming of their virtualization environments, to change the reporting behaviour, including parameters that are characteristics of each app (logs, events), network traffic, system calls, Remote Procedure Call (RPC) toward remote applications. Programming also include the capability to offload lightweight aggregation and processing tasks to each virtual environment, hence reducing bandwidth requirements and latency.

The data plane is responsible for enforcing security policies, including packet filtering, access control, and re-configuration of the execution environment. A fundamental property for the data plane is programmability, that is the capability to shape the deep of inspection according to the current need, in both spacial and temporal dimensions, so to effectively balance granularity of information with overhead.

The flexibility in programming the execution environment is expected to potentially lead to a large heterogeneity in the kind and verbosity of data collected. For example, some virtual functions may report detailed packet statistics, whereas other functions might only report application logs. In addition, the frequency and granularity of reporting may differ for each execution environments. Correlation of data in the time and space dimensions will naturally lead to concurrent requests of the same kind of information for different time instants and functions. Finally, the last requirement is the ability to perform quick look-ups and queries, also including some forms of data fusion. That would allow clients to define the structure of the data required, and exactly the same structure of the data is returned from the server, therefore preventing excessively large amounts of data from being returned. This could turn very useful during investigation, when the ability to understand the evolving situation and to identify the attack requires to retrieve and correlate data beyond typical query patterns.

The *control plane* is a logically and centralized collections of algorithms for detection of attacks and identification of new threats. Every algorithm retrieves the data it needs from the common data plane. This represents one the main innovation behind the proposed framework: indeed, every algorithm has complete visibility on the overall system, removing the need to have local agents deployed in each virtual function, [8] which often perform the same or very similar inspection operations. The control plane should also include programming capabilities to configure and offload local processing tasks to the data plane, so to effectively balance the depth of inspection with the generated overhead.

Beyond the mere (re-)implementation of legacy appliances for performance and efficiency matters, the ASTRID approach is specifically conceived to pave the road for a new generation of detection intelligence, arguably by combining detection methodologies (rules-based, machine learning) with big data techniques; the purpose is to locate vulnerabilities in the graph and its components, to identify possible threats, and to timely detect on-going attacks.The combined analysis of security logs, events, and network traffic from multiple intertwined domains can greatly enhance the detection capability, especially in case of large multi-vector attacks. In this respect, the application of machine learning and artificial intelligence would be useful to inspect and correlate the large amount of data,

[2] https://openbaton.github.io.
[3] https://github.com/T-NOVA/TeNOR.
[4] http://arcadia-framework.eu.
[5] https://jaas.ai.
[6] https://wiki.openstack.org/wiki/Heat.
[7] In this paper, we refer to the terms *data* and *log*, interchangeably.
[8] In this paper, we refer to the terms *virtual function*, and *virtual service* interchangeably.

events, and measures that have to be analysed for reliable detection and identification of even complex multi-vector attacks.

The control plane basically corresponds to the "detection logic" depicted in Fig. **??**. It might look like we are anyway inserting additional virtual functions in the service graph. Indeed, we point out that this component should not necessarily run as an additional virtual function, but a dedicated infrastructure is perhaps the best choice for security and efficiency reasons (this is roughly comparable with cloud scrubbing centres used to mitigate DDoS attacks). For example, the same control plane may be shared by multiple graphs, with the possibility to combine and correlate contextual information from them, which further improves timely detection of new attacks.

The *management plane* is conceived to keep humans in the loop. It notifies detected attacks and anomalies, allowing access to the full context in case the human expertise is needed to complement artificial intelligence in the inspection process. The management plane supports quick and effective remediation actions, by the definition of high-level policies that are then translated in specific data plane configurations from the control plane. The management plane also seamlessly integrates with orchestration tools which are expected to be widely used for automating deployment and life-cycle operations of virtual services [7, 9, 13].

3 An abstraction layer for the data plane

The main purpose for an abstraction layer is to provide uniform access to the underlying data plane capabilities. According to the general description in Section 2, the data plane is made of heterogeneous inspection, measurements, and enforcement hooks, which are implemented in the virtualization environment.

These hooks include logging and event reporting developed by programmers into their software, as well as monitoring and inspection capabilities built in the kernel and system libraries that inspect network traffic and system calls. They are *programmable* because they can be configured at run-time, hence shaping the system behaviour according to the evolving context. This means that packet filters, types and frequency of event reporting, and verbosity of logging are selectively and locally adjusted to retrieve the exact amount of knowledge, without overwhelming the whole system with unnecessary information. The purpose is to get more details for critical or vulnerable components when anomalies are detected that may indicate an attack, or when a warning is issued by cyber-security teams about new threats and vulnerabilities just discovered. This approach allows lightweight operation with low overhead when the risk is low, even with parallel discovery and mitigation, while switching to deeper inspection and larger event correlation in case of anomalies and suspicious activities. This allows to scale with the system complexity, even for the largest services.

There are two main aspects to be covered by the abstraction layer: i) hiding the technological heterogeneity of the monitoring hooks; and ii) abstracting the whole service graph and the capabilities of each node.

Fig. 2 shows a schematic view of the envisioned abstraction. Locally, within each virtualization box, a Local Security Agent (LSA) provides a common interface to different hooks. Then, the whole graph topology is abstracted as a hub-and-spokes graph. In this model, each node represents a virtual function and each link a communication path. Satellites of nodes are security properties; they include both monitoring/inspection capabilities (what can be collected, measured, and retrieved) and relative data (metrics, events, logs). Similarly, links have properties too (though not explicitly shown in the picture), related to the usage of encryption mechanisms and utilization metrics. This abstraction, effectively decouples the detection logic from the distributed data plane: a common language can be used to query security-related attributes and to re-program inspection and enforcement tasks, without the need to use different interfaces and heterogeneous semantics.

To provide composite metrics, data fusion is also envisioned as part of the overall abstraction framework. Pre-processing and aggregation of elementary data can be accomplished by the same query, hence optimizing look ups in the abstraction model. The abstraction layer also includes storage capabilities, so to provide both real-time and historical information for both on-line and off-line analyses. In this abstraction, the overall topology and security capabilities are set by the orchestrator, whereas security data are fed by LSAs.

4 Implementation

As described in a previous section, the data plane is the part of the architecture responsible for difference actions: i) collecting the security context (in terms of events, logs, measures, etc.), and ii) enforcing security policies (in terms of packets filtering, access control, re-configuration of the execution environment, etc.).

The collection of the security context is mediated by an abstraction layer for retrieving data and programming the monitoring tasks. Considering the description of the suitable technologies to implement the whole data plane in [5], we selected the Elasticsearch [9] – Logstash [10] – Kibana [11] (ELK) stack provided by Elastic. [12]

Generally, applications – in our case, (virtual) services – generates logs and data because they serve as a mirror to their state and health. With the voluminous amount of generated logs, it becomes imperative to have a system that can analyse this data and present a singular view of the application/service. When the application is deployed in a distributed environment, maintaining and retrieving the data can be challenging. Searching for an error across several virtual services and through a large number of data files is extremely difficult.

Centralized logging provided by the ELK stack is a step in this direction. It allows searching through all data at a central place. It is a versatile collection of open-source software tools that are implemented based on a distributed log collector approach that makes gathering insights from data easier. In a nutshell, the ELK stack consists of three core projects: i) *Elasticsearch* as a search and analytics engine, ii) *Logstash* for data processing and transformation pipeline, and iii) *Kibana* a web UI to visualize data. Together, they form the acronym ELK. Afterwards, Elastic launched a fourth project called Beats (lightweight and single-purpose data shippers) and decided to rename the combination of all projects to simply Elastic Stack. To learn more about the history behind it, a nice explanation can be found in [1].

In addition to the ELK stack, we selected Apache Kafka. [13] It is publish-subscribe messaging rethought as a distributed commit log. Kafka was created at LinkedIn [14] to handle large volumes of event data [8]. Like many other message brokers, it deals with publisher-consumer and queue semantics by grouping data into topics.

The Elastic Stack and Apache Kafka share a tight-knit relationship in the log/event processing realm. A number of companies use Kafka as a transport layer for storing and processing large volumes of data. In many deployments we've seen in the field, Kafka plays an important role of staging data before making its way into Elasticsearch for fast search and analytical capabilities.

4.1 Logstash Overview

Logstash supports a variety of inputs that pull in events from a multitude of common sources, all at the same time. Easily ingest from your logs, metrics, web applications, data stores, and various Amazon Web Service

[9] https://www.elastic.co/products/elasticsearch
[10] https://www.elastic.co/products/logstash
[11] https://www.elastic.co/products/kibana
[12] https://www.elastic.co
[13] http://kafka.apache.org.
[14] https://www.linkedin.com.

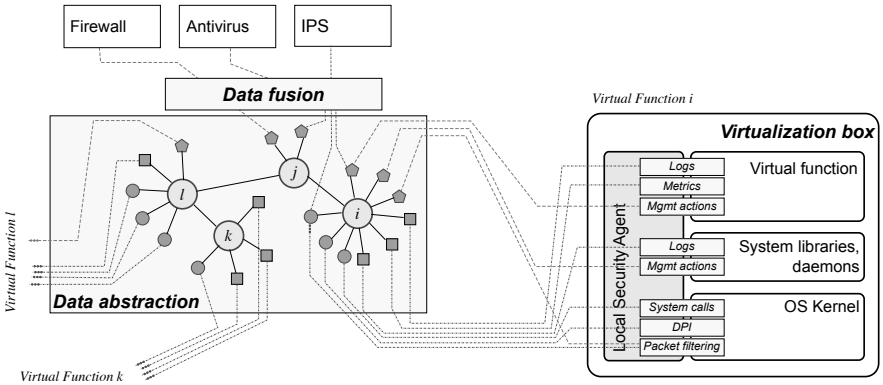


Figure 2: The data plane collects and abstract the security context for the whole virtual graph.

(AWS) services, all in continuous, streaming fashion. The filter are Logstash plug-ins that can read events, parse them, structure data, and then transform them into an easy to analyse format. Common filters are: i) *grok* to derive structure from unstructured data, ii) *geoip* to decipher geo-coordinates from Internet Protocol (IP) addresses, iii) *fingerprint* to anonymize Personally Identifiable Information (PII) data, exclude sensitive fields, and iv) *date* to identify timestamp of the log.

4.2 Elasticsearch Overview

Elasticsearch is a distributed, RESTful search and analytics engine. It allows to store, search, and analyse big volumes of data quickly and in near real time. A *cluster* is a collection of one or more nodes (i.e. servers) that together holds the entire data and provides indexing and search capabilities across all nodes. A *node* is a single server that stores data and participates in the cluster's indexing and searching capabilities. A *document* is a basic unit of information that can be indexed. For example, it can be possible to have a document for a single customer and another document for a single product. Instead, an *index* is a collection of documents that have somewhat similar characteristics. For example, an index for customer data, another one for a product catalogue. The *type* used to be a logical category/partition of indices allowing to store different types of documents in the same index, i.e. one type for users and another for blog posts.

4.3 Kibana overview

Kibana provides data visualization capability. It has a browser-based User Interface (UI). It is the analytics and visualization components of the Elastic Stack. Kibana provides various kinds of charts like histograms, line graphs, and pie charts. It integrates easily with Elasticsearch. It has easy-to-share reports as Portable Document Formats (PDFs), Comma-separated valuess (CSVs), embed links, etc.

4.4 Architecture

Fig. 3 show the architecture of the PoC implementation. Different kinds of data are generated like system log files, database log files, logs generated by message queues, and other middle-wares. Those data are collected by Beats installed on all virtual functions (services). The Beats send the logs to a local instance of Logstash at fixed interval. Then, Logstash after some light data processing, send the processed output to the Context Broker (CB) that is the centralized node where the data are collected and saved for centralized analysis and correlation. Inside the CB, Kafka sends the data to a local instance of Logstash. After the processing, Logstash sends out the data to the Elasticsearch that will then index and store the data. Finally, Kibana provides a visual interface for searching and analysing the data.

5 Performance Evaluation

In this section, we provide a functional validation and extensive performance evaluation of a PoC implementation, including integration with local

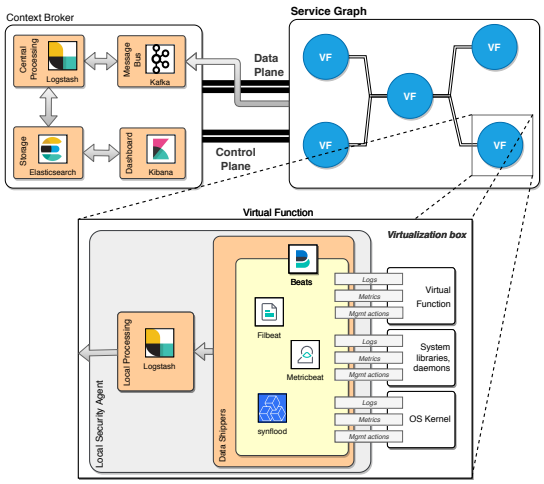


Figure 3: Architecture of the PoC implementation.

monitoring/enforcement agents. We validate the effectiveness to collect data from local agents in order to apply centralized security analysis and appropriate actions to solve cyber-issues.

In Sub-section 5.1, we describe the test-bed for PoC evaluation and its relative configuration. Instead, Sub-section 5.2 shows the results of the performed tests.

5.1 Test-bed

Fig. 4 shows the test-bed for the PoC evaluation of the proposed abstraction layer. For the performance evaluation, we consider 3 different virtual services: i) Apache Hyper-Text Transfer Protocol (HTTP) server, [15] ii) MySQL database server, [16] and iii) mini_httpd server [17] with a PolyCube [18] synflood app to detect if the system is under attack.

To collect the data from the Apache HTTP and MySQL service we use the filebeat and Metricbeat agent, respectively. Instead, to interact with the Polycube framework and to collect the data from the synflood app, we implemented a custom Beats called *Polycubebeat*. In this ways, we provide a simple evaluation of the custom modularity of the proposed layer and, at the same time, we guarantee an uniform format of the collected data.

The specifications of the test-bed are shown in Table 1. All the virtual functions and the CB are set as VMs. The processor of the underlying physical machines are: Intel Xeon E5-4610 with 4 CentralProcessingUnits(CPUs)

[15] http;//http.apache.org.
[16] http://www.mysql.com.
[17] http://https://acme.com/software/mini_httpd.
[18] https://github.com/polycube-network/polycube.

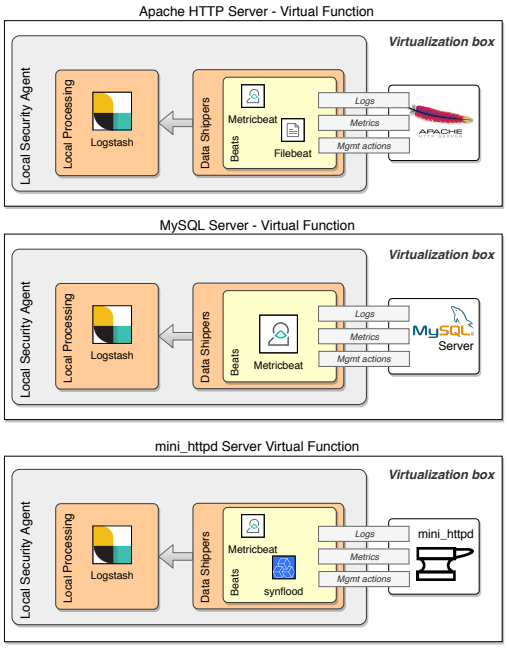


Figure 4: Test-bed for the PoC evaluation.

and 8 cores each ones with the hyper-threading [19] enabled. Instead, the available Random Access Memory (RAM) is equal to 128 GB.

Table 1: Test-bed specification for the PoC evaluation.

Apache	
CPU	1 virtual – core 2295 GHz
RAM	1 GB
OS	Linux 4.9.0 – Debian 4.9 (64 bit)
MySQL	
CPU	1 virtual – core 2295 GHz
RAM	1 GB
OS	Linux 4.9.0 – Debian 4.9 (64 bit)
mini_httpd	
CPU	2 virtual – core 2295 GHz
RAM	2 GB
OS	Linux 4.15.0 – Ubuntu 18.04.2 LTS (64 bit)
Context Broker	
CPU	4 virtual – cores 2295 GHz
RAM	4 GB
OS	Linux 4.9.0 – Debian 4.9 (64 bit)

The evaluation consists of different tests varying the following parameters: i) the average number of request per second α, and ii) the data collection period β. The α parameter allows to evaluate the scalability of the proposed layer and how it response to different level of burst of data. For the evaluation, we consider the following values: 10 requests/s, 100 requests/s, 500 requests/s and 1000 requests/s Instead, the β parameter sets the polling period of the Beats to collect the data from the applications. For this parameter, we consider the following values: 1 s, 10 s, 500 s and 1000 s. We repeated the experiments for a sufficient number of times to get small errors, representing the 95 % Confidence Intervals (CIs). The measured CIs have not been reproduced in the graphs since all of them are small and clutter the figures. During the experiments, we measured the following average statistics: workload in term of CPU utilization (γ), latency (η) and jitter (ι). The measures are performed for each blocks involved in the log collection (as shown in Figs. 3 and 4).

[19] A feature of certain Intel chips that makes one physical CPU appear as two logical CPUs.

5.2 Numerical Results

Fig. 5 shows the results in the CB and in the virtual functions of the CPU workloads during the performance evaluation. The amount CPU required to get the data from the virtual functions mainly depends in the average number of events per seconds. The polling internal does not substantially leverage the performance.

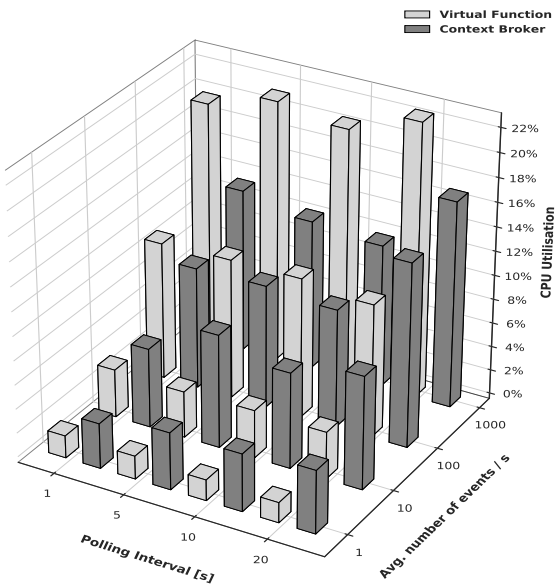


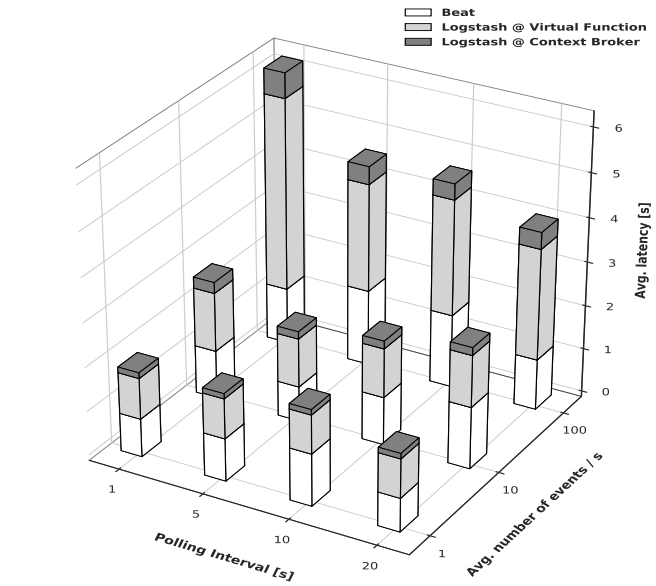
Figure 5: CPU utilization in the virtual functions and in the CB during the performance evaluation.

The results from the point of views of the network are shown in Fig. 6. The latency and the jitter are low when the number of events per second are less than 1000. Instead, with 1000 events per second, the beat in the virtual functions take a long time to get the data. In more details, when the polling internal is set to 20 s, the beat is not able to catch all the data during the performance evaluation. This means that, with this number of events, the virtual machines make the most of their performance. Currently, we are working to overcome these issues.

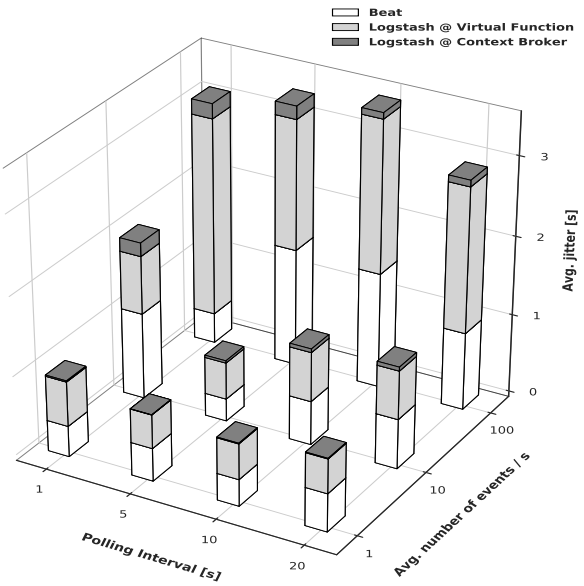
6 Conclusion

In this paper, we have outlined the main features and the preliminary design of an abstraction layer that provide bi-directional access to an heterogeneous set of information and sources. This approach makes large data sets available for application of machine learning and other artificial intelligence mechanisms, which are currently the main research frontier for a new generation of threat detection algorithms. Differently from existing approaches, our target is to expose programmable features of the execution environment, which can be used to program local inspection and monitoring tasks.

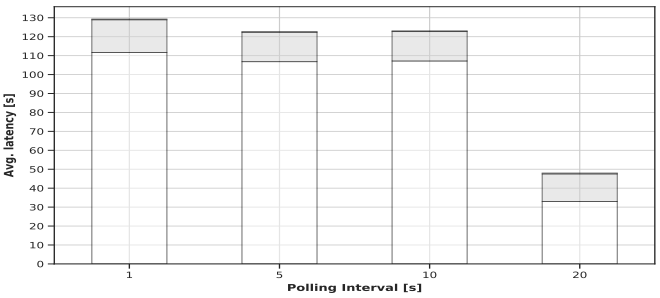
We describe in details the architecture based on ELK stack integrated with Kafka message broker and how can satisfy the requirement to collect log for cyber-security analysis. We provide a functional validation and extensive performance evaluation of a PoC implementation, including integration with local monitoring/enforcement agents. The results shows that, considering the capacity, the architecture is able to collect data without delay when maximum resources are not used. At the limit of the used resources (when the number of events per second is equal to 1000, and independently of the polling interval value), the various beats in the virtual



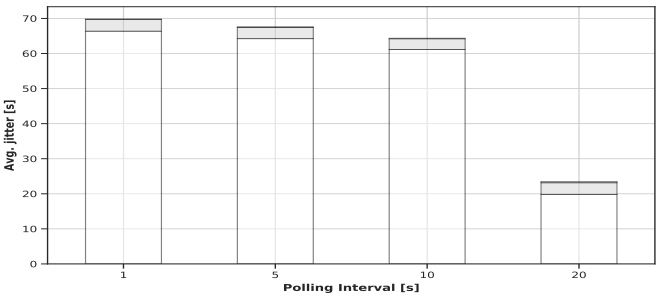
(a) Latency – Cases with $\alpha < 1000$.



(b) Jitter – Cases with $\alpha < 1000$.



(c) Latency – Case with $\alpha = 1000$.



(d) Jitter – Cases with $\alpha = 1000$.

Figure 6: Average latency and jitter computed when the beat gets the data, when Logstash in the virtual function receive the data, and when the data arrives at Logstash in the CB.

functions are not able to collect the data without introducing significant delays.

As future works, we will provide the Application Program Interfaces (APIs) to obtain the data from the CB and to read/set the agent's status in each virtual functions.

Acknowledgments

This work was partially supported by the European Commission under the projects ASTRID (contract 786922) and GUARD (contract 833456).

References

[1] [n.d.]. ELK Stack. https://www.elastic.co/elk-stack
[2] [n.d.]. Vmware vcloud air. http://vcloud.vmware.com
[3] 2017. Security groups – OpenStack Networking guide. OpenStack documentation. http://docs.ocselected.org/openstack-manuals/kilo/networking-guide/content/section_securitygroups.html
[4] 2017. Security Groups for Your VPC. AWS documentation. http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_SecurityGroups.html
[5] R. Bolla, A. Carrega, and M. Repetto. 2019. An abstraction layer for cybersecurity context. In *International Conference on Computing networking and communications (ICNC)*. 214–218. https://doi.org/10.1109/ICCNC.2019.8685665
[6] S. Covaci, M. Repetto, and F. Risso. 2018. A New Paradigm to Address Threats for Virtualized Services. In *Proc. of the 42nd IEEE Ann. computer software and applications Conference (COMPSAC)*, Vol. 02. 689–694. https://doi.org/10.1109/COMPSAC.2018.10320
[7] Jokin Garay, Jon Matias, Juanjo Unzilla, and Eduardo Jacob. 2016. Service description in the NFV revolution: Trends, challenges and a way forward. *IEEE Commun. Mag.* 54, 3 (March 2016), 68–74. https://doi.org/10.1109/MCOM.2016.7432174
[8] Joel Koshy. 2016. Kafka Ecosystem at LinkedIn. https://engineering.linkedin.com/blog/2016/04/kafka-ecosystem-at-linkedin
[9] Ahmed M. Medhat, Tarik Taleb, Asma Elmangoush, Giuseppe A. Carella, Stefan Covaci, and Thomas Magedanz. 2017. Service Function Chaining in Next Generation Networks: State of the Art and Research Challenges. *IEEE Commun. Mag.* 55, 2 (Feb. 2017), 216–223. https://doi.org/10.1109/MCOM.2016.1600219RP
[10] Jon Oberheide, Evan Cooke, and Farnam Jahanian. 2008. CloudAV: N-version antivirus in the network cloud. In *Proceedings of the 17th conference on Security symposium (SS'08)*. San Jose, CA – USA, 91–106.
[11] Gábor Pék, Levente Buttyán, and Boldizsár Bencsáth. 2013. A survey of security issues in hardware virtualization. *Comput. Surveys* 45, 3 (June 2013), 40:2–40:34.
[12] R. Rapuzzi and M. Repetto. 2018. Building situational awareness for network threats in fog/edge computing: Emerging paradigms beyond the security perimeter model. *Future Generation Computer Systems* (Aug. 2018), 235–249. https://doi.org/10.1016/j.future.2018.04.007
[13] J. Wettinger, U. Breitenbücher, and F. Leymann. 2014. Standards-Based DevOps Automation and Integration Using Topology and Orchestration Specification for Cloud Applications (TOSCA). In *Proc. of the 7th IEEE/ACM International Conference on utility and cloud Computing (UCC)*. IEEE, London, England, U.K., 59–68.