An architecture to manage security services for cloud applications

M. Repetto, A. Carrega
S2N Lab, CNIT
Genoa, Italy
Email: {alessandro.carrega,matteo.repetto}@cnit.it

G. Lamanna
Infocom Srl
Genoa, Italy
Email: guerino.lamanna@infocomgenova.it

***Abstract*—The uptake of virtualization and cloud technologies has pushed novel development and operation models for the software, bringing more agility and automation. Unfortunately, cyber-security paradigms have not evolved at the same pace and are not yet able to effectively tackle the progressive disappearing of a sharp security perimeter.**

In this paper, we describe a novel cyber-security architecture for cloud-based distributed applications and network services. We propose a security orchestrator that controls pervasive, lightweight, and programmable security hooks embedded in the virtual functions that compose the cloud application, pursuing better visibility and more automation in this domain. Our approach improves existing management practice for service orchestration, by decoupling the management of the business logic from that of security. We also describe the current implementation stage for a programmable monitoring, inspection, and enforcement framework, which represents the ground technology for the realization of the whole architecture.

I. Introduction

Today, many organizations are progressively moving their ICT processes in (private and public) clouds, motivated by the increasing agility to deploy and withdraw their services without caring about the physical infrastructure that hosts them.

One of the main operational benefits brought by the cloud paradigm is full control of the virtualization infrastructure through software APIs. The fully-digital management workflow allows large automation in deployment and lifecycle operations. Orchestration software replaces humans for repetitive tasks, such as provisioning of virtual resources as well as installation and configuration of applications, driven by descriptive models based on imperative or declarative languages that capture the topology and behavior of the service.

The combination of dynamic provisioning and configuration management software allows to change the service at runtime, so to react to the evolving context, hence providing elasticity, redundancy, and resilience. Most important, such changes can be automated, by defining suitable management policies in the service template. However, the ability to adapt to the evolving context makes the service topology partially unpredictable at design-time, which may be a major issue for security.

The final publication is available at IEEE Xplore via https://doi.org/10.1109/CCCS.2019.8888061.

In this paper, we review the two main paradigms (i.e., infrastructure-centric and service-centric) that have been used so far for implementing security services for cloud applications. Motivated by the lack of common protocols, APIs, and security models, which hinder cross-cloud interoperability when implementing security services, we describe an architecture that improves the visibility over cloud services and automates reaction. Our approach is based on the definition of security hooks embedded in the virtual functions, and external detection and processing logic. The main objective is the independence between service orchestration, which manages the business logic, and security orchestration, which directly controls all security-related components. The distinctive characteristic of our approach is the integration with programmable technologies, so to easily support new inspection and monitoring tasks at run-time.

The paper is organized as follows. Section II briefly reviews the concept of software orchestration. Section III lists alternative paradigms for implementing security frameworks for cloud-based services, and highlights the main novelty of our approach. We describe the architecture for a holistic cyber-security framework for cloud services in Section IV, and we describe the part of this framework that we are implementing in Section V. Preliminary results from experimentation are shown in Section VI. Finally, we give our conclusion in Section VIII.

II. Software orchestration

The ability to easily adapt applications to the evolving context calls for a transition from "imperative" (i.e., procedural languages) to declarative models, an on-going process both for cloud applications [1], [2] and network function virtualization [3]. A declarative model defines the application as a logical topology (the service "graph") of elementary (virtual) functions, together with a set of rules and constraints for deployment and operation. A service is usually provided as a sort of template, which has to be instantiated and initialized at run-time by "Service Providers" through an orchestration process. The set of information that describes how to instantiate, configure, and manage the service is denoted as metadata. It includes the name and version of the software, vendor, description (including licensing and usage terms), entry points, deployment constraints, and management hooks (for instance, to start, stop, reload, or reset the service, to collect measurements, data, events, log).

The elementary components of any service topology are (virtual) functions. Virtual functions are developed by programmers (hereby indicated as "software developers") and delivered in different forms: (compressed) archive, packages including dependencies and configuration scripts, bootable images.

Virtual functions should be enriched with metadata as well to drive automatic deployment and orchestration. Metadata typically includes the name of the component (i.e., trademark and vendor), its description (including licensing and usage terms), provided functionality (e.g., web server, database, DNS, EPC, eNodeB, RAS), required services (e.g., database, authentication server), deployment constraints (e.g., number of cores, CPU speed, RAM, disk space, network bandwidth, hardware acceleration), measured performance metrics (e.g., packet latency and throughput, dropped packets, packet statistics), and management hooks (for instance, to start, stop, reload, or reset the function). This information is used by orchestration tools to provision the proper set of resources, set up and configure the execution environment, and perform life-cycle management actions (e.g., scale the function upon indication from the orchestrator, recover from failure).

Starting from the declarative service template, orchestration is responsible to start the provisioning process for virtual resources, deploy and configure the software, start all functions and execute any lifecycle management operation. The whole process may be totally automated, or there may remain a number of functions that should be carried out by human staff. Typical orchestration tools monitor at run-time the execution of the graph, by collecting measurements about used resources (CPU, RAM, disk, network), workload, performance (processed requests, latency) as defined in the service/function metadata. This data is then used to trigger lifecycle management actions, according to the policies defined by the service designer or the service provider. Example of well-known software orchestration tools are Juju, Kubernetes, OpenBaton, OpenMANO.

III. Security paradigms for distributed cloud applications

From a purely architectural perspective, we can identify legacy security appliances as an infrastructure-centric paradigm, because they have been traditionally designed to protect the physical infrastructure, not the services implemented on top of it. Though many appliances have been re-designed and re-implemented in the hypervisor layer to cope with virtualization and multi-tenancy, this approach still creates management issues for services that are deployed over heterogeneous infrastructures, though the IETF I2NSF working group is already tackling this challenge [4].

The progressive dichotomization between the software and the underlying hardware brought by the adoption of virtualization and cloud paradigms has boosted a transition from infrastructure-centric to service-centric architectures, which are more suitable to be integrated in the overall service orchestration process. When the Infrastructure-as-a-Service paradigm is used, virtual instances of security appliances are "plugged" into service graphs, leveraging the large correspondence with physical infrastructures that is present in this model. Each tenant retains full control and responsibility of security management for its own graphs, without the need to rely on and trust external services. In some way, this represents a sort of virtual perimeter model, though the isolation from external threats is not comparable with a physical infrastructure. We argue that application to other cloud models is not straightforward, especially when some software components are shared among multiple tenants (i.e., Service-as-a-Service model).

Following the ever-more agility and adaptability to the evolving context at both the infrastructure and service layer, the basic principle behind our approach is a service-centric framework, with security capabilities embedded into each software element, and orchestrated by a common security manager that (logically) centralizes the detection processes.

IV. An orchestration framework for security services

Existing service-centric approaches for cyber-security of cloud applications/network services make use of a common software orchestrator for both service management and security operation, with the risk of raising conflicts between different operators. The main objective of our work is the definition of a security orchestration framework that is independent from the management of the business logic: our design goal is therefore a stand-alone security orchestrator that interacts with existing software orchestration tools, by means of their APIs (see Fig. 1). The two orchestrators are therefore responsible for implementing operation at different logical layers: control (security orchestrator) and management (service orchestrator).

According to the main concept introduced in the previous Section, our approach breaks up cyber-security appliances into two parts: monitoring, inspection, and enforcement tasks (embedded in the same service graph) and detection, analysis, and reaction logic (which are part of the orchestrator itself). The colored elements in Fig. 1 are the components introduced by our architecture; grey elements represent virtualization and management components that are already present in existing software orchestration frameworks.

The service orchestrator retains the exclusive control over the overall service topology and its components; it is therefore responsible for provisioning and deprovisioning virtual resources, deploying and updating software, managing life-cycle events. The security orchestrator controls specific security components placed in the service graph (pictorially depicted as orange ellipses in Fig. 1), but it relies on the service orchestrator to deploy them and to make any changes in the service topology. Examples of such security components include:

- kernel features, integrated in the operating system for monitoring network traffic and system calls;

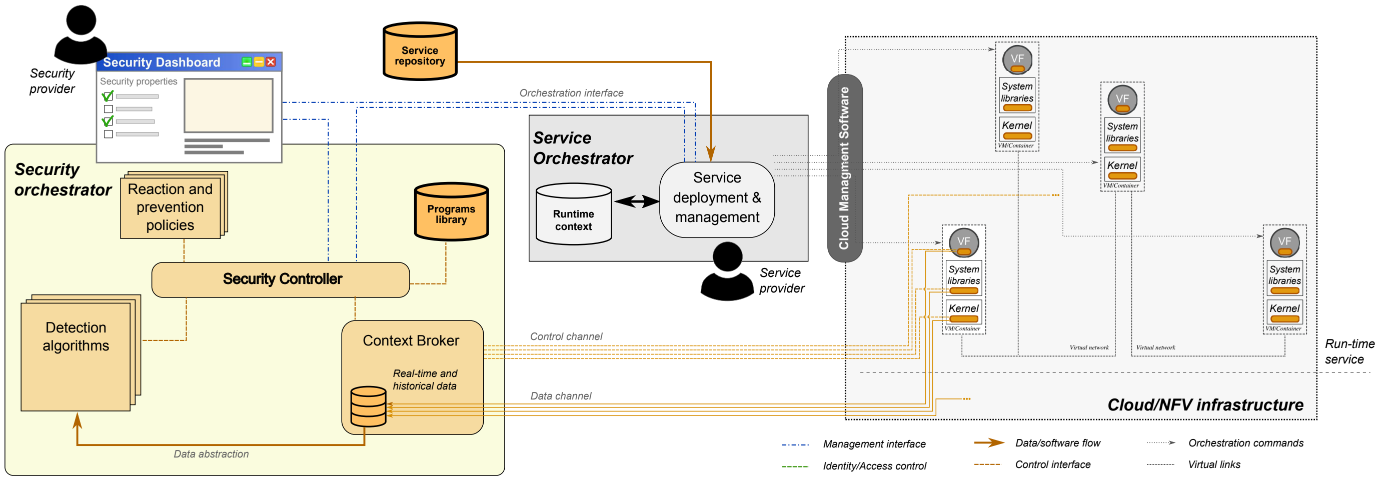

Fig. 1: Architecture for secure orchestration of cloud-based services.

- log agents in user-space that collect logs from libraries, daemons, and even the kernel, so their scope extends to the whole stack;
- function-specific monitoring information that is conceived to drive orchestration actions, but might also be used for detection purposes.

Security components for monitoring and inspection are expected to be integrated in the service template at design time. At deployment time, any software orchestration tool that understands the specific template format takes care of installing all the libraries, proxies, and agents required by the security enrichment in the execution environment (indicated as orange ellipses in the pictorial representation of the service topology in the Cloud/NFV Infrastructure).

The security orchestrator acts as the mediator between the security context and the detection logic, aiming to automate as much as possible reaction and mitigation. The role of its main components and their interactions are briefly described in the following Sections.

A. *Security Controller*

The Security Controller represents the most valuable part of the security orchestrator, conceived to automate as much as possible the behavior of the whole framework. It positions between the reaction and mitigation policies and the context, and orchestrates security functionalities.

Overall, the Security Controller will work according to an Event-Condition-Action (ECA) pattern. *Events* are triggered by detection or management entities. Detection events include the indication of on-going attacks, compromised virtual functions, vulnerabilities found in software, protocols and configurations, behavioral anomalies. Management events include initialization triggers, start/stop actions, scaling, topology changes (insertion/removal/replacement of virtual functions), configuration changes, and so on. *Conditions* on the current context are evaluated by querying the Context Broker, which exposes the service topology, configuration, events, and measurements. *Actions* entail modifications of the security hooks (monitored data, frequency, granularity, filtering, marking, etc.), re-configuration of the detection algorithms, changes in the service graph. Commands may be sent directly to the responsible entity (service orchestrator, detection algorithm, Context Broker) or to the Security Dashboard.

ECA rules are expressed by *policies*, which represent the real "smartness" of the Security Controller. Though the ECA pattern is quite expressive, the long-term ambition is the incorporation of "intent" frameworks, which can derive the low-level behavior based on very high-level requirements and expectations from users.

The adoption of advanced reasoning models, even based on machine learning and other forms of artificial intelligence, is clearly a very promising yet challenging target to automate the system behavior. This would open the opportunity for dynamically adapting the response to new threat vectors. In this respect, the historical analysis and correlation of the events and conditions with the effects of the corresponding actions from existing policies or humans would provide useful hints to assess the effectiveness of the latter, so to identify and improve the best control strategies.

B. *Context Broker*

The first task for the Context Broker is to manage the heterogeneity of sources and protocols, which is reflected in different data and control interfaces. The Context Broker hides this heterogeneity and exposes a common data model to the other components in the security orchestrator, both in the data and control planes, for discovering, configuring, and accessing the security context available from the execution environment.

The Context Broker collects data from monitoring and inspection processes deployed in the execution environment (*data* channel in Fig. 1). The Context Broker hides the heterogeneity and asynchrony of the sources, organizes historical data, and provides simple querying and fusion capabilities in data access.

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 (i.e., those at the external boundary of the service), whereas other functions might only report application logs. In addition, the frequency and granularity of reporting may differ for each virtual function. The definition of a (security) context model is therefore necessary for detection algorithms to know what could be retrieved (i.e., capabilities) and what is currently available, how often, with each granularity (i.e., configuration). The Context Broker also offers a homogeneous control interface for configuring and programming different data sources, by implementing the specific protocols (*control* channel in Fig. 1). Given the very different semantics of the context data, the obvious choice is non-relation databases (NoSQL). This allows defining different records for different sources, but also poses the challenge to identify a limited set of formats, otherwise part of the data might not be usable by some detection algorithms.

C. *Programs library*

One of the main distinctive characteristics for future cyber-security frameworks will be programmability, that is the capability to shape the depth of inspection according to the current need, in both spatial and temporal dimensions, so to effectively balance granularity of information with overhead. This goes beyond mere re-configuration of individual components and their virtualization environments: programmability also includes the capability to offload lightweight aggregation and processing tasks to each virtual environment, hence reducing bandwidth requirements and latency.

A programmable run-time environment is able to run monitoring, inspection, classification, and aggregation tasks on demand. The long-term ambition would be the definition of dynamic code generation and run-time compiling, as part of the intent framework; the main technical challenge here is the definition of common actions at the policy level and their translation into configurations and code for the heterogeneous set of security hooks.

The current architecture is more pragmatic in this respect. Programmability is realized by selecting pre-defined programs and configuration files from an internal library. Different languages can be used by different hooks: ELF binaries, java bytecode, python scripts, P4/eBPF programs. Such programs are written and compiled offline, and then pushed in the repository by the Security Dashboard. They also include metadata for identification and description, so to be easily referred by the Security Controller. The scope of such programs may include monitoring, inspection, and enforcement actions; it is clearly limited by the instruction set of the execution virtual machine (if any).

From a security perspective, the current architecture assumes that the programs are safe. This is implicitly guarantee, for example, for the eBPF, where the code runs within an execution sandbox. In case of general-purpose languages, the correctness and safety of the source code might be verified by static source-code tools.

D. *Detection algorithms*

Detection algorithms process, analyze and correlate security-related data and events. They can be mapped to existing security appliances (DoS detection, IPS, IDS, antivirus, etc.). They are fed by the Context Broker, rather than implementing their own monitoring and inspection functions.

The availability of a broad real-time and historical security context from the whole graph, virtually including any type of data and events, paves the road for novel and more effective detection approaches, leveraging machine learning and other forms of artificial intelligence to detect anomalies. The availability of heterogeneous data from multiple sources theoretically allows the detection of any kind of threats and attacks, including the typical scope of host-based, network-based, and hybrid IDSes, and antiviruses. From a practical perspective, however, the real range of algorithms will be limited by the possibility to find an acceptable trade-off between the complexity to implement local inspection and the communication overhead.

From an architectural perspective, each algorithm will only be required to implement the interfaces towards the Context Broker (to retrieve real-time and historical data) and the Security Controller (to notify security events like threats and attacks). For existing tools, this could be achieved by developing plug-ins or adapters. The description of the generated security events may include an estimation of the accuracy of the detection, so to trigger the collection of more detailed information; alternatively, this information could be retrieved by evaluating specific conditions on the current security context.

The scope of the detection algorithms is mostly affected by the type of monitoring and inspection hooks in the execution environment. While the availability of measurements on network traffic is rather straightforward, the analysis of the software execution flow is more challenging. Events and logs generated by applications are usually enough for IPS/IDS, but the implementation of antivirus and other forms of (remote) software attestation usually requires deeper access to system calls, memory, registers, and instructions. The possibility to remotely collect this information is still an open issue that deserves further investigation [5].

E. *Reaction and prevention policies*

Policies are used to automate the response to expected events, avoiding whenever possible repetitive, manual, and error-prone operations done by humans. Conceptually, reaction and prevention policies do not implement inspection, detection or enforcement tasks, so they do not correspond to any existing security function (IDS/IPS, antivirus, VPN).

According to current definition of the Security Controller, policies should be expressed with the ECA pattern. The definition of an ECA policy requires at least 3 elements:

- an *Event* that defines when the policy is evaluated; the event may be triggered by the data plane (i.e., detection algorithms), the management plane (i.e., manual indications from the dashboard, notifications from the service orchestrator), or the control plane (i.e., a timer);

- a *Condition* that selects one among the possible execution paths; the condition typically considers context information as data source, date/time, user, past events, etc.;
- a list of *Actions* that respond, mitigate, or prevent attacks.

Actions might not be limited to simple commands, but can implement complex logics, also including some form of processing on the run-time context (e.g., to derive the firewall configuration for the running instance). They can be described by imperative languages, in the forms of scripts or programs.

The range of possible operations performed by policies includes enforcement actions, but also re-configuration and re-programming of the monitoring/inspection components in the execution environment. Enforcement and mitigation actions are mostly expected when the attack and/or threat and their sources are clearly identified and can be fought. Instead, re-configuration is necessary when there are only generic indications, and more detailed analysis could be useful to better focus the response.

A typical example is a volumetric DoS attack. To keep the processing and communication load minimal, the monitoring process may only compute rough network usage statistics every few minutes. This is enough to detect anomalies in the volume of traffic but does not give precise indication about the source and identification of malicious flows to stop. Re-configuring the local probes to compute per-flow statistics or more sophisticated analysis helps implement traffic scrubbing.

F. Communication channels

The interaction with the execution environment requires the setup of communication channels, for both control and data messages. These communication buses between virtual functions and the security orchestrator are secure channels deployed by the service orchestrator.

There are multiple implementation alternatives:

- *In-band.* The simplest approach would be to use virtual network(s) in the IaaS already used for communication between the virtual machines. This does not require any specific effort on the orchestration side, but the traffic for data collection may overwhelm the network and should be recognizable so as to not alter traffic statistics for detection purposes. In addition, though the usage of encryption is taken for grant, there are still security concerns in mixing the management and service data traffic.
- *Out-of-band in the service graph.* The instantiation of a dedicated virtual network for the control and data channel looks a better approach with minimal overhead on service management. In this case, the enrichment process must envision an additional virtual NIC for each VM and a service-wide flat network connecting all VMs and the security orchestrator.
- *Out-of-band in management interfaces.* The underpinning assumption for automatic life-cycle management through service orchestration is i) the access to monitoring and context information about the execution of virtual functions, and ii) the possibility to interact with virtual functions to trigger management scripts. That means a management channel must be available outside the execution environment. An example of management channel is the Ve-Vnfm interface in the ETSI MANO architecture [6], which may correspond to the control network of OpenStack or a physical network for Docker containers.

G. Security Dashboard

An interface is envisioned in the proposed architecture to manage the security orchestrator. It can be used to select specific software analysis, to visualize anomalies and security events and to pinpoint them in the graph topology, to set run time security policies, and to perform manual reaction. With respect to the last two options, we point out that security policies are the best way to respond to well-known threats, for which there are already established practice and consolidated methodologies for mitigation or protection. However, the identification of new threats and the elaboration of novel countermeasures require direct step-by-step control over the on-going system behavior. The dashboard interacts with the orchestration system to give security provider full control over the graph in case of need.

H. Example of workflow

We give a concrete example of how our security framework is expected to behave in case of DoS service.

Detection of volumetric DoS is typically based on analytics on the network traffic. Since deep inspection of the traffic leads to high computational loads and latency, an initialization policy only requires statistics about the aggregate network traffic that enters the service, which may be collected by standard measurements reported by the kernel. The same policy also initializes an algorithm for network analytics and sets the alert thresholds.

Upon detection of an anomaly in the traffic profile, an event is triggered and the Security Controller invokes the corresponding DoS policy. The policy now requires finer-grained statistics; the Security Controller selects a filtering program from the repository of programmable components for packet classification, installs and configures it. The policy also requires the detection algorithms to work with the broader context information now available. Network analytics may then confirm the attack or classify the anomaly as a fortuitous event.

Before taking the decision about how to react, the mitigation policy may evaluate some conditions to check if either the suspicious flow comes from an expected user of the service, or has been previously blacklisted or whitelisted, or is acceptable based on previously recorded time series. The actions to be implemented (e.g., dropping all packets, dropping selected packets, redirecting suspicious flows towards external DoS mitigation hardware/software, stop the service, move part or the whole service to a different infrastructure) is therefore notified to the Security Controller, which again translates them in a set of commands for the external service orchestrator, and/or enforcement configurations and programs to be installed in the execution environment.

V. Programmable monitoring and inspection

One of the most challenging and innovative aspects of the security orchestration architecture discussed in Section IV is programmable inspection and monitoring of virtual functions. Existing tools and frameworks are mostly conceived to work statically, i.e., they always collect the same kind of information, with minimal or no possibility to change the configuration at run-time.

Our solution enhances existing frameworks with an additional dimension of programmability. We started from the Elastic Stack framework[1], a collection of open-source projects for data acquisition, processing, and storage. It was originally composed of Elasticsearch, Logstash, and Kibana (for which it was formerly known as ELK) and is now evolving to include additional options.

Fig. 2 shows our implementation of the monitoring and inspection framework, including both a data plane and a control plane. The left side of the picture entails the components that are deployed in the execution environment by service orchestration; the components in the right side are part of the Context Broker.

Logstash is a server-side data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then sends it to a remote "stash" like Elasticsearch. Data is gathered by a family of lightweight, single-purpose data shippers called *beats*. Several beats are already available to grab log files from popular applications (FileBeat), to read metrics from the kernel (MetricBeat), and even to perform measurements on network traffic (PacketBeat).

Our extension includes a new beat, named PolyBeat, which reads measurements, statistics, and events generated by eBPF programs. The enhanced Berkeley Packet Filter (eBPF) is an efficient, safe, and programmable event processing framework implemented in the Linux kernel, suitable for both packet inspection and software analysis. It can be used both for monitoring, inspection, and enforcement tasks.

eBPF programs can be easily injected at run-time, but we need a control plane to support this feature. We adopted polycube[2], a framework that provides fast and lightweight network functions such as bridges, routers, firewalls, and others. Polycube is made of control functions (*cubes*) implemented in high-level languages, and eBPF programs as data plane. We started with a simple program and limited statistics, but there will be virtually no limit to the classification, measurement, inspection, and enforcement filters that can be implemented and dynamically orchestrated by the Security Controller. An additional cube is currently under implementation to control other beats, so to change their configuration at run-time (this feature is not available in Logstash). A further beat will be implemented to load processing tasks in Logstash. Polycube offers a REST API for remote control, which in our framework implements the control channel envisioned by the overall architecture.

[1]The Elastic Stack. URL: https://www.elastic.co/elk-stack.

[2]Polycube.network. URL: https://github.com/polycube-network/polycube.

The data channel is implemented by Kafka[3], a message bus system for building real-time data pipelines and streaming apps. Though LogStash could directly feed the Elasticsearch database, we opt for this additional component to steer measurements towards detection applications, so to reduce the latency for real-time processing.

Elasticsearch is a NoSQL search and analytics engine, a perfect choice for storing unstructured data and performing queries on graph-based topologies. Kibana is currently used to visualize data with charts and graphs in Elasticsearch, but will be replaced at a later stage with a more powerful GUI (the Security Dashboard) with specific functions to display the service graph and manage security features.

VI. Experimental evaluation

We carried out preliminary evaluation in a small set up, which included most of the monitoring agents described in Sec. V. We considered three virtual functions: an Apache web server (monitored by FileBeat), a MySQL database server (monitored by MetricBeat), and a mini_httpd web server (monitored by PolyBeat); we also deployed a Context Broker that collects all data. Our testbed includes therefore logs, application status, and network measures.

Our purpose was to evaluate the overhead introduced by the distributed monitoring framework. We considered both CPU usage by security agents and network traffic generated to collect the security context. We investigated how variable workload and frequency of sampling impact these parameters. Specifically, we generated 1, 10, 100, and 1000 requests per second to emulate different workload levels for Apache and MySQL; the same number of requests were used to emulate a Denial-of-Service (DoS) attack to mini_httpd. The sampling period was varied from 1 to 20 seconds.

Fig. 3 shows the impact of the workload and polling period on CPU usage. For Apache, a larger number of requests implies more logs, hence more CPU is used to read the data and send them to the Context Broker. Shorter polling periods create more data when the data has fixed size (e.g., in case of status and network measurements for MetricBeat and PolyBeat); the impact on CPU usage is very small, as shown for MySQL and mini_httpd.

Similar considerations hold for bandwidth usage. Fig. 4 compares the total amount of traffic generated by each virtual function with the amount of data sent to the Context Broker. Also in this case the larger number of logs generated by Apache has a huge impact on resource usage by FileBeat; however, they still represent a small fraction with respect to the overall traffic. The same fraction is even smaller for MetricBeat and PolyBeat, where the size of the collected data is fixed. The length of the polling interval has negligible impact in this case.

[3]Apache Kafka. URL: https://kafka.apache.org/.

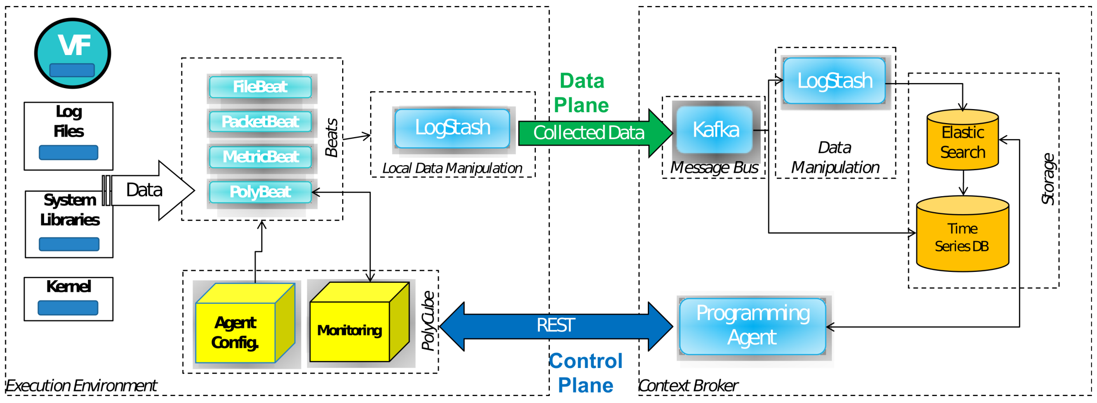

Fig. 2: Implementation of the context fabric for collecting and processing the security context.

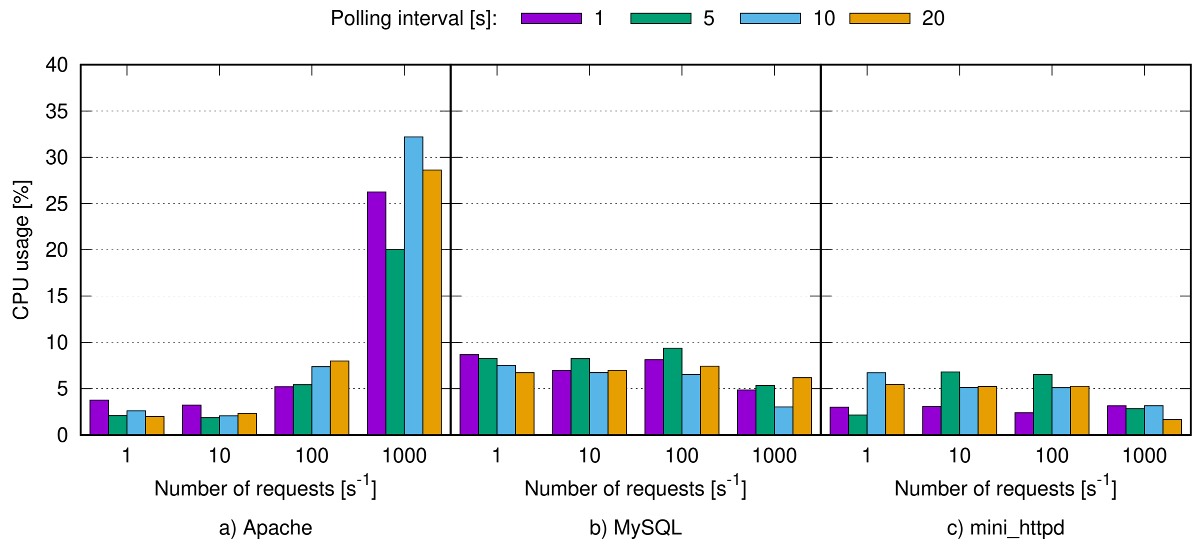

Fig. 3: CPU usage by the different monitoring agents.

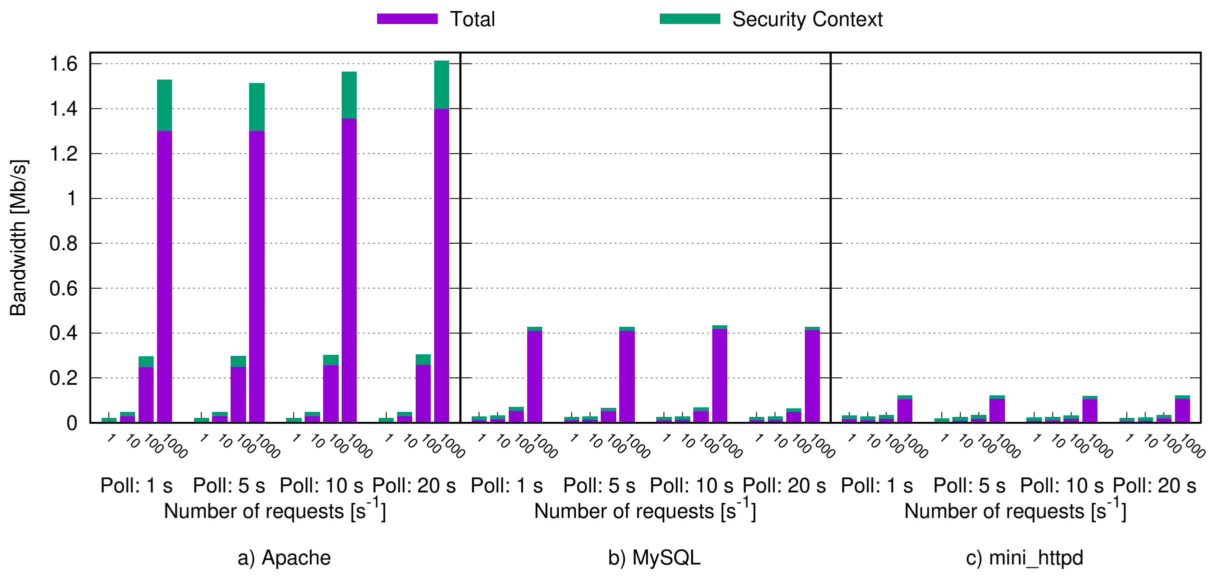

Fig. 4: Bandwidth usage for all traffic (“Total”) and the monitoring flow to the Context Broker (“Security Context”).

VII. Related work

Virtualization and cloud technologies are eroding ever more the concept of security perimeter, with the primary consequence that legacy security appliances cannot anymore applied according to existing models [7]. Accordingly, attempts can be found in the literature that aim at improving security in distributed, multi-domain and multi-tenancy systems. The work [8] is among the first proposals of a distributed antivirus (CloudAV). The usage of multiple detection engines in parallel increases the likelihood of detection and enable correlation between analysis from different engines, but also raises false positives compared to a 1-version engine.

An agent-based framework for virtualised services is proposed in [9]. The approach is based on Static and Mobile Agents (SA/MA). The former are inspection tools statically deployed in each VM; they are responsible to detect suspicious conditions and report to a central IDS Control Centre (IDS CC). The latter are detection agents for specific attacks, that are sent to VM to investigate in detail suspected conditions. The work in [10] proposes a distributed IDS for the Grid. This approach is based on standalone IDSes that share the security context for detection. They consider both messages exchanged by the grid nodes and logs from the grid middleware. Detection is based on behavioral anomalies (through neural networks) and knowledge (rule-based).

The goal of the proposal in [11] is to design a collaborative framework among IDPS deployed in different domains of a cloud environment (host, network, VM). The framework shares information at both the local (in clusters of similar hosts) and global level (among clusters), so that attacks and security events can be correlated to identify complex patterns. The duplication at the local and global level introduces redundancy and also communication overhead.

More recently, some works on countermeasures against cyber attacks in distributed systems are surveyed in [12]. They present a high degree of scalability, in terms of computational efforts, because of the architectures that distribute the computational tasks to the peripheral part of the network (i.e., the nodes), instead of concentrating them in a single central point. The weak point is that the works do not consider dynamic changes in the network parameters, so the countermeasures cannot dynamically change. Furthermore, the amount of information on the security state of the system, that strongly grows with the size of the system, makes the proposed solutions not suitable in real distributed environments.

VIII. Conclusion

In this paper we have described a framework for orchestrating security functions for virtual services, including both cloud applications and network services. We also carried out preliminary performance evaluation, in order to investigate the impact of the main parameters on resource usage.

Our future work will be the integration of the monitoring framework with security orchestration. This step will implement the necessary logic to automate the configuration of monitoring parameters at run-time, including the frequency of polling and the set of measurements to be collected. The second step will be the integration with detection and analysis algorithms, to demonstrate the benefits of a programmable context framework in terms of efficiency and effectiveness of detection.

Acknowledgment

This work was supported in part by the European Commission, under Grant Agreement no. 786922 and 833456.

References

[1] B. Karakostas, "Towards autonomic cloud configuration and deployment environments," in *International Conference on Cloud and Autonomic Computing (ICCAC)*, London, UK, Sep., 8th-12th 2014, pp. 93–96.

[2] J. Wettinger, U. Breitenbücher, and F. Leymann, "Standards-based DevOps automation and integration using TOSCA," in *IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)*, London, UK, Dec. 8–11, 2014, pp. 59–68.

[3] P. Bellavista, L. Foschini, R. Venanzi, and G. Carella, "Extensible orchestration of elastic IP multimedia subsystem as a service using Open Baton," in *5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, San Francisco, CA – USA, Apr., 6th-8th, 2017, pp. 88–95.

[4] S. Hares, D. Lopez, M. Zarny, C. Jacquenet, R. Kumar, and J. Jeong, "Interface to network security functions (I2NSF): Problem statement and use cases," IETF RFC 8192, July 2017. [Online]. Available: https://www.rfc-editor.org/rfc/pdfrfc/rfc8192.txt.pdf

[5] N. Koutroumpouchos, C. Ntantogian, S. A. Menesidou, K. Liang, P. Gouvas, C. Xenakis, and T. Giannetsos, "Secure edge computing with lightweight control-flow property-based attestation," in *1st International Workshop on Cyber-Security Threats, Trust and Privacy Management in Software-defined and Virtualized Infrastructures (SecSoft 2019)*, Paris, France, 2019, to appear.

[6] ETSI, "Network functions virtualisation (nfv); management and orchestration," ETSI GS NFV-MAN 001, December 2014, v1.1.1.

[7] R. Rapuzzi and M. Repetto, "Building situational awareness for network threats in fog/edge computing: Emerging paradigms beyond the security perimeter model," *Future Generation Computer Systems*, vol. 85, pp. 235–249, August 2018.

[8] F. J. Jon Oberheide, Evan Cooke, "Cloudav: N-version antivirus in the network cloud," in *Proceedings of the 17th conference on Security symposium (SS'08)*, San Jose, CA – USA, Jul. 28th – Aug. 1st, 2008, pp. 91–106.

[9] A. V. Dastjerdi, K. A. Bakar, and S. G. Hassan Tabatabaei, "Distributed intrusion detection in clouds using mobile agents," in *Third International Conference on Advanced Engineering Computing and Applications in Sciences*, Sliema, Malta, Oct. 11th–16th, 2009, pp. 175–180.

[10] K. Vieira, A. Schulter, C. Westphall, and C. Westphall, "Intrusion detection for grid and cloud computing," *IT Professional*, vol. 12, no. 4, pp. 38–43, Jul-Aug 2010.

[11] S. T. Zargar, H. Takabi, and J. B. Joshi, "DCDIDP: A distributed, collaborative, and data-driven intrusion detection and prevention framework for cloud computing environments," in *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, Pittsburgh, PA – USA, Oct. 15th–18th 2011, pp. 332–341.

[12] P. Nespoli, D. Papamartzivanos, F. G. Marmol, and G. Kambourakis, "Optimal Countermeasures Selection Against Cyber Attacks: A Comprehensive Survey on Reaction Frameworks," *IEEE Communications Surveys Tutorials*, vol. 20, no. 2, pp. 1361–1396, Secondquarter 2018.