Towards Effective Virtualization of Intrusion Detection Systems

Nuyun Zhang†, Hongda Li†, Hongxin Hu† and Younghee Park‡
†Clemson University ‡San Jose State University
{nuyun, hongdal, hongxih}@clemson.edu, younghee.park@sjsu.edu

ABSTRACT

Traditional Intrusion Detection Systems (IDSes) are generally implemented on vendor proprietary appliances or *middleboxes*, which usually lack a general programming interface, and their versatility and flexibility are also very poor. Emerging Network Function Virtualization (NFV) technology can virtualize IDSes and elastically scale them to deal with attack traffic variations. However, existing NFV solutions treat a virtualized IDS as a *monolithic* piece of software, which could lead to inflexibility and significant waste of resources. In this paper, we propose a novel approach to virtualize IDSes as microservices where the virtualized IDSes can be customized on demand, and the underlying microservices could be shared and scaled independently. We also conduct experiments, which demonstrate that virtualizing IDSes as microservices can gain greater flexibility and resource efficiency.

Keywords

Network Function Virtualization; Intrusion Detection Systems; Microservices

1. INTRODUCTION

Intrusion Detection System (IDS) is a critical network security function that is designed to monitor a network or system to detect malicious activities or security policy violations [16]. Recently, the network throughput has been dramatically increased to 100 Gbps for a number of networks [15]. Therefore, multi-thread approach has been adopted in IDSes to meet the high throughput requirement for detecting attacks [3]. Besides that, some IDSes, such as Bro [2], enable built-in capability to spread workload across IDS clusters [14]. However, despite their usefulness, both multi-thread and cluster solutions have some shortcomings. Especially, they are still inflexible to deal with attacks when a significant workload spike happens. For example, a massive attack like DDoS could bring the network traffic volume up to 0.5 TBps [6].

SDN-NFV Sec'17, March 22-24, 2017, Scottsdale, AZ, USA

DOI: http://dx.doi.org/10.1145/3040992.3041004

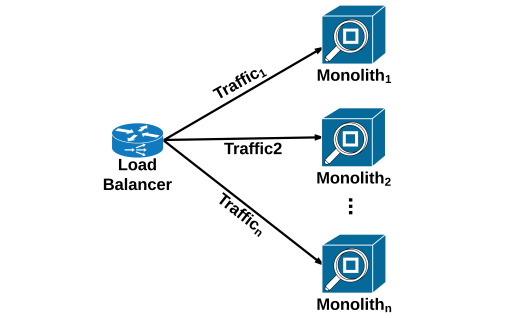
As an emerging networking paradigm, Network Function Virtualization (NFV) can be used to address those challenges. NFV turns network functions (NFs) into software-based and virtualized entities, which are deployed and run on industry standard servers. The virtualized NF (VNF) could be located in data centers, distributed network nodes, and cloud [11]. As a result, VNFs are low cost, flexible in deployment and management, and elastically scalable. Unfortunately, existing NFV solutions [8, 13] often treat a virtualized IDS as a *monolithic* piece of software running as a whole to detect attacks, where components within one virtualized IDS cannot be run independently. The nature of monoliths makes the distribution of a virtualized IDS into cloud-based systems with significant limitations such as inefficent use of resource and difficulty for sharing.

In this paper, we propose an approach to virtualize an IDS by decomposing its components into multiple different microservices [4], such as low-level connection parsing microservice and high-level attack detecting micorservice. Each micorservice can be instantiated and run independently, and provisioned with different numbers of virtual instances. Our approach provides a number of unique features for IDS virtualization: (1) resources can be fully utilized as instances of microservices are smaller; (2) customization of IDSes can be accomplished easily as components of IDSes are exposed and defined as micorservices; (3) different IDS components can scale individually according to their resource requirements; and (4) one micorservice could be shared by multiple other microservices at the run time. In general, microservices can achieve more flexibility and cost-effectiveness in term of IDS virtualization. To demonstrate the effectiveness of our approach, we conduct our experiments using Bro. Our experiments show that virtualizing IDSes as microservices can gain greater flexibility and resource efficiency.

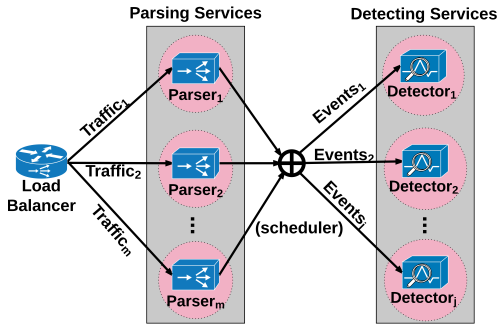
The rest of this paper is organized as follows. Section 2 describes our method of IDS virtualization as microservices. Section 3 presents our experiments based on the Bro system. Conclusion is drawn in Section 4.

2. OUR APPROACH

As the increasing concurrency and DevOps requirements of software rise, the microservices architecture has sprung up. Microservices are small services running in their own processes independently while communicating with each other through lightweight mechanisms [7]. The microservices architecture focuses on breaking an application into smaller and completely independent components, enabling each component to scale individually and be available all the time [10].



(a) IDS virtualization as monoliths



(b) IDS virtualization as microservices

Figure 1: IDS virtualization as monoliths and microservices

We utilize the microservices architecture to effectively virtulize IDS to meet high concurrency and on-demand provision requirements.

First of all, we argue that IDSes could be decomposed into small pieces. For example, as pointed out by [3], Bro-like IDSes can be abstracted into two parts: 1) low-level per-connection parsing, which enforces check-sum verification, stream reconstruction, protocol parsing, etc; and 2) high-level analysis—i.e, performing the detection tasks. The low-level parser captures the traffic data from the network and parses the traffic data in per-connection manner. Then, it generates a stream of events, which are delivered to the high-level analyzers for analysis. The two parts can run independently and in parallel, providing well-balanced workloads.

Second, we argue that virtulized IDSes can be run as microservices. In general, the low-level parser uses Libpcap to capture the packets and/or AF-Packet PF-Ring to achieve load balancing [9]. The high-level analyzer is normally IDS-specific. For example, Bro has its own interpreter to interpret security policies written in a specific scripting language. These software modules can execute independently and communicate via predefined messages. The execution of the two layers does not need to share resources, such as libraries, which makes it feasible to run virtulized IDSes as independent microservices.

2.1 Virtulizing IDSes as microservices

Based on our previous discussion, we can identify two microservices for building virtulized IDSes. The low-level per-connection parsing part can be defined as a `parsing` microservice; and the high-level analysis part can be defined as a `detecting` microservice. Figure 1(a) shows an IDS are traditionally virtulized as monoliths. Figure 1(b) illustrates how an IDS can be virtualized as two microservices, where both microservices can be further instantiated into a number of virtual instances according to the workload of microservices. The traffic data is split and delivered to each virtual instance of the `parsing` microservice, respectively. The `parsing` microservice is responsible for receiving traffic data from the network and process packets in per-connection granularity. The traffic data is discarded after being processed and only events generated by the `parsing` microservice are delivered to the `detecting` microservice. A scheduler is required to coordinate the communication between virtual instances of the `parsing` microservice and the `detecting` microservice, since the two microservices can be provisioned with different numbers of virtual instances. The `detecting` microservice then receives the events and performs analysis based on the predefined security policies.

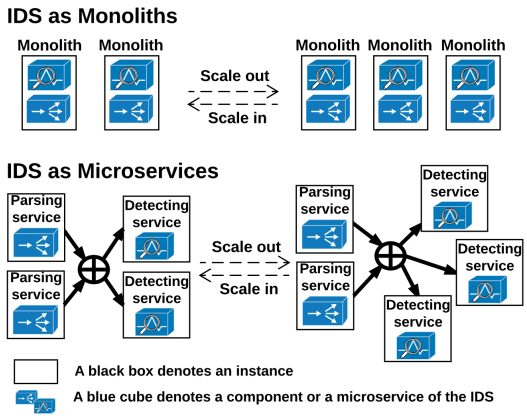
2.2 Benefits of IDS virtulization as microservices

As we can see from Figure 1, once an IDS is virtualized as microservices, its deployment becomes more *flexible*. Instead of finding a powerful instance to fit the whole IDS, the system could easily find a smaller instance to fit a microservice, which saves resources. Besides, we can identify three more benefits of our approach as follows:

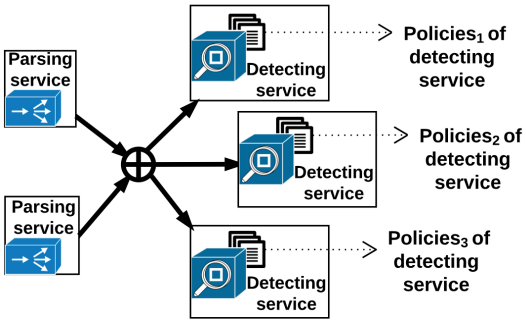
1) **Individually Scalability**. If an IDS is virtualized monolithically, all components in an IDS must be scaled out at the same time. For example, in the top part of Figure 2 (a), when two IDS instances scale out to three, the number of parsing components increases to three and the number of detecting components also increases to three. However, as we have discussed in Section 1, different parts of an IDS needs different amounts of resources. We assume the parsing service needs relatively less resources than the resources needed by the detecting service. When traffic increases, only the detecting service needs to scale out. The lower part of Figure 2 (a) illustrates an example where the number of parsing service instances remains two after scaling out but one more detecting service instance starts. We can see that the microservices are individually scalable. In this way, cost-efficiency and flexibility can be achieved.

2) **Customization**. As shown in Figure 2 (a), virtualizing an IDS monolithically makes all IDS instances the same. But using microservices, the IDS instances could be different. Figure 2 (b) illustrates the detecting service can be customized by setting different policies. The top instance has the detection service enabled to detect a certain set of attacks by $Policies_1$, the middle instance enforces $Policies_2$, and the bottom instance enforces $Policies_3$. In addition, virtualized IDS can choose different underlying microservices to perform a tailored intrusion detection function, which increases its flexibility.

3) **Shareability/reusability**. Figure 2 (b) illustrates the parsing service instances can be shared by different detecting service instances. The outputs of parsing services are a set of events, which can be used by different detection services for further analysis. In addition, if there are more than one network functions in the environment, a microservice can be also reused by different network functions. The shareable and reusable features of virtualizing IDSes as microservices can not only save resources but also make the system management and maintenance much easier.



(a) Individually scalable



(b) Customizable

Figure 2: Benefits of IDS virtualization as microservices

3. EXPERIMENTS

We have conducted experiments using Bro, an open source IDS platform to compare the resource usage of the two approaches (microservice versus monoliths) of IDS virtualization. The testbed of our experiments was constructed in CloudLab [1], which provisions a number of virtual machines. Each virtual machine ran a Bro IDS. The virtual machines were all allocated with 4 virtual CPU cores and 4GB memory.

The datasets for the experiments were *real* network traffic collected from Mid-Atlantic Collegiate Cyber Defense Competition (MACCDC) [12]. We did not use synthetic traffic because the resource usage and performance of IDS significantly depend on the volume and relevance (how relevant the traffic data is to the loaded policies) of the network traffic. In the experiments, we replayed the traffic traces from one virtual machine and distributed the traffic data to other virtual machines via Open vSwitch.

In the experiments, first we adopted emulation to assess the CPU usage of the IDS virtualized as microservices, since there are no existing IDS platforms that support microservice approach; then we assessed the CPU usage of Bro, a representative of monolithic IDS.

1) Microservices. In the microservices approach, we identified two microservices, the `detecting` microservice and the `parsing` microservice. Our goal was to assess the overall CPU usage of the two microservices. We emulated the `detecting` microservice and `parsing` microservice by using Bro IDS based on the model proposed by Dreger etal. [5]. The model pointed out that the CPU usage of a typical security policy is independent to other security policies, which provide a chance to decompose the CPU usage of Bro into orthogonal portions being assessed in isolation. The `detecting` microservice was instantiated as *detector* and the `parsing` microservice was instantiated as *parser*. Each *detector* was customized to only load a part of the policies. Multiple *detectors* shared one *parser*. Based on this model, we designed our emulation as follows:

- Starting a Bro instance without loading any security policies. This setup emulates the *parser*, since the Bro instance only performs low-level per-connection parsing in this situation [1]. We delivered all traffic data to the Bro instance and assessed the CPU usage (notated as c_0).
- Starting a Bro instance loading only a part of security policies (notated as D_1). This setup emulates the *parser* and $detector_1$. We delivered all traffic data to the Bro instance and assessed the CPU usage (notated as c_1).
- Starting a Bro instance loading a part of security policies (notated as D_2), such that D_1+D_2 is equivalent to the entire security policies of Bro. This setup emulates the *parser* and $detector_2$. Then we delivered all traffic data to the Bro instance and assessed the CPU usage (notated as c_2).

We explored three different methods to divide all the security policies in Bro into two complementary parts, D_1 and D_2, and assessed the resource usage for each partition method. The three partition methods are as follows:

1. Classifying by *http* and *non-http*, where *http* includes all security policies for the detections of attacks relevant to HTTP traffic, while *non-http* includes the remaining security policies.
2. Classifying by *protocols* and *non-protocols*, where *protocols* includes all security policies for all protocol-based detections supported by Bro, while *non-protocols* includes the remaining.
3. Classifying by *others* and *non-others*, where *others* includes all security policies for the detections of attacks across multiple traffic flows, such as port scan, file examination, Trojan detection, etc., while *non-others* includes the remaining.

Based on the IDS resource usage model proposed in [5], we derived: *(i)* the CPU usage of *parser* was estimated as c_0; *(ii)* the CPU usage of $detector_1$ was estimated as $c_1 - c_0$; and *(iii)* the CPU usage of $detector_2$ was estimated as $c_2 - c_0$. Finally, we summed up the usage of each microservice instance and estimated the overall CPU usage as $c_1 + c_2 - c_0$.

The CPU usage of the three different partition methods is shown in Figure 3. The *HTTP/non-HTTP*, *Protocols/non-Protocols* and *Others/non-Others* columns represent the CPU usage of the three partition methods, respectively. The overall CPU usage of each method consists of three components, the CPU usages of *parser*, $detector_1$ and $detector_2$. The three partition methods achieved similar overall CPU usage (24.9%, 26.1% and 26.2%) and reasonable load balance between two *detectors* in each method.

[1]This emulation consumes more resources than the parsing service in practice, since some fundamental policies have to be loaded. But this overhead does not invalidate our estimation.

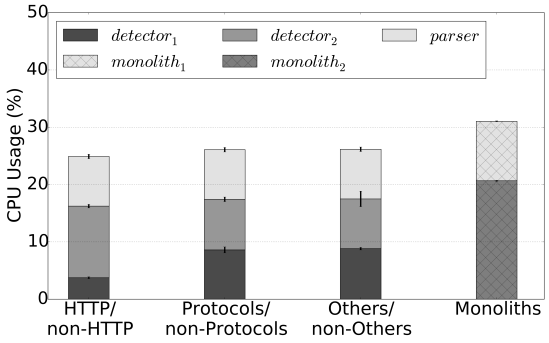


Figure 3: CPU usage of different microservice instances compared with that of monolithic instances. The *HTTP/non-HTTP*, *Protocols/non-Protocols* and *Others/non-Others* are the emulation results of three different methods to partition the detection policies. The *Monoliths* are the experimental results of the two monolithic Bro instances.

2) Monoliths. In this experiment, we split the traffic data into two portions based on IP addresses and delivered the two portions of traffic data to two monoliths (notated as $monolith_1$ and $monolith_2$), respectively. Then, we assessed the CPU usages of $monolith_1$ and $monolith_2$. Let c_4 and c_5 be the CPU usages of the two monoliths. Then we can derive that the overall CPU usage of $monolithic_1$ and $monolithic_2$ is $c_4 + c_5$.

The overall CPU usage of $monolithic_1$ and $monolithic_2$, is shown as the *Monoliths* column in Figure 3. We observed approximately **25%** extra overall CPU usage with monoliths approach than that with microservices approach in average. The extra CPU usage was induced by the *parser* replicas in the two monolithic instances, since the parsing service can be shared by all *detector*s and should be consolidated into one instance. Therefore, we conclude that the microservice approach achieves more resource efficiency due to scaling individually and microservice sharing.

4. CONCLUSION

We have proposed a new approach to virtualize IDSes, leveraging emerging microservices architecture. We have addressed three benefits using microservices for IDS virtualization: *individual scalability*, *customization* and *shareability*. Through experiments on Bro, we have demonstrated that IDS virtualization could gain more efficiency of resource usage by leveraging microservices.

Acknowledgment

This work was partially supported by grants from National Science Foundation (NSF-ACI-1642143, NSF-IIS-1527421, and NSF-CNS-1537924).

5. REFERENCES

[1] CloudLab. http://www.cloudlab.us/.

[2] Bro IDS. https://www.bro.org/, December 2016. [Online; accessed 16-Dec-2016].

[3] L. De Carli, R. Sommer, and S. Jha. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1378–1390. ACM, 2014.

[4] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036*, 2016.

[5] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Predicting the resource consumption of network intrusion detection systems. In *International Workshop on Recent Advances in Intrusion Detection*, pages 135–154. Springer, 2008.

[6] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and elastic ddos defense. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 817–832, 2015.

[7] M. Fowler". Microservices a definition of this new architectural term. http://www.martinfowler.com/articles/microservices.html, 2014. [Online; Published on 25 March, 2014].

[8] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. *ACM SIGCOMM Computer Communication Review*, 44(4):163–174, 2015.

[9] George Khalil. Open Source IDS High Performance Shootout. https://www.sans.org/reading-room/whitepapers/intrusion/open-source-ids-high-performance-shootout-35772, 2015. [Online; accepted 2-Feb-2015].

[10] Kim Clark. Microservices, SOA, and APIs: Friends or enemies? http://www.ibm.com/developerworks/websphere/library/techarticles/1601_clark-trs/1601_clark.html, 2016. [Online; Published on 21 Jan, 2016].

[11] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys & Tutorials*, 18(1):236–262, 2015.

[12] Netresec. MACCDC Dataset. http://www.netresec.com/?page=MACCDC, December 2016. [Online; accessed 16-Dec-2016].

[13] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 227–240, 2013.

[14] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware. In *Recent Advances in Intrusion Detection, 10th International Symposium, RAID 2007, Gold Goast, Australia, September 5-7, 2007, Proceedings*, pages 107–126, 2007.

[15] G. Wellbrock and T. J. Xia. How will optical transport deal with future network traffic growth? In *Optical Communication (ECOC), 2014 European Conference on*, pages 1–3. IEEE, 2014.

[16] Wikipedia. Intrusion detection system — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Intrusion_detection_system, 2016. [Online; accessed 16-Dec-2016].