2015 IEEE 7th International Conference on Cloud Computing Technology and Science

Security-as-a-Service for Microservices-Based Cloud Applications

Yuqiong Sun
Penn State University
Email: yus138@cse.psu.edu

Susanta Nanda
Symantec Research Labs
Email: susanta_nanda@symantec.com

Trent Jaeger
Penn State University
Email: tjaeger@cse.psu.edu

***Abstract*—Microservice architecture allows different parts of an application to be developed, deployed and scaled independently, therefore becoming a trend for developing cloud applications. However, it comes with challenging security issues. First, the network complexity introduced by the large number of microservices greatly increases the difficulty in monitoring the security of the entire application. Second, microservices are often designed to completely trust each other, therefore compromise of a single microservice may bring down the entire application. The problems are only exacerbated by the cloud, since applications no longer have complete control over their networks. In this paper, we propose a design for security-as-a-service for microservices-based cloud applications. By adding a new API primitive *FlowTap* for the network hypervisor, we build a flexible monitoring and policy enforcement infrastructure for network traffic to secure cloud applications. We demonstrate the effectiveness of our solution by deploying the Bro network monitor using FlowTap. Results show that our solution is flexible enough to support various kinds of monitoring scenarios and policies and it incurs minimal overhead (∼6%) for real world usage. As a result, cloud applications can leverage our solution to deploy network security monitors to flexibly detect and block threats both external and internal to their network.**

***Keywords*-microservices; network monitoring; security;**

I. Introduction

There are multiple trends that are forcing modern cloud applications to evolve. Users expect rich, interactive, and dynamic user experience on a wide variety of client devices. Applications must be highly scalable, highly available, and must run on cloud environments. Organizations want to roll out frequent updates – sometimes, even multiple times a day. Consequently, it is no longer adequate to develop monolithic web applications. The predominant way to address this problem today is to use an alternate architecture – known as *microservices architecture* [1] – that decomposes a monolithic application into a set of narrowly focussed, independently deployable services, called *microservices*. The popularity of this architecture is evident from the report by the popular jobs portal indeed.com [2] that the number of job openings on microservices-related technologies, such as JSON and REST, has grown more than 100 fold in the last six years, whereas jobs in similar technology areas like SOAP and XML have stayed nearly identical.

The microservices architecture, due to its new design paradigm, introduces two major security challenges. First, the microservices design creates many smaller applications interacting among themselves that results in complex network activity. This makes monitoring and securing networks for the overall application and individual microservices very challenging. Second, the trusted computing base (TCB) for cloud applications usually includes all of the component microservices and compromise of one could result in the entire application's compromise. It gets even more challenging in the public cloud environment where the application administrator has limited access to the application network.

One way to address these challenges is to leverage the software defined networking (SDN) capabilities provided by modern cloud networking and program the networks in a way that monitors the complex network interactions and enforces policies on them. For example, SDN provides the ability to scan through network packets at every forwarding element and control the forwarding as per the application requirements. This also allows to control the granularity of the network flows (while processing) and even changing them dynamically based on attack patterns and application behaviors. It also allows for passive monitoring (e.g. via copy-and-forward) and active rerouting (e.g. via changing the forwarding destination) that can be leveraged based on security requirements.

In this paper, we propose the design, development, and evaluation of a security-as-service infrastructure for microservices-based cloud applications. We propose a cloud-based network security framework that helps cloud application administrators/providers to construct a global view of their application, even when its components are distributed throughout the cloud. The framework also enables application providers/administrators to flexibly implement their own security control over its services, thus preventing a compromised service from compromising the rest of the application or the cloud platform/hypervisor. Our design is motivated by the micro-kernel design [3], where a security kernel monitors and mediates security critical operations performed by "server" (aka services) running atop, effectively removing the services from the TCB of the application.

II. Background and Problem

In this section, we illustrate microservice architecture with an example application—an online DVD rental store, and discuss security challenges of this design paradigm. Other

978-1-4673-9560-1/15 $31.00 © 2015 IEEE
DOI 10.1109/CloudCom.2015.93

applications will have different service decomposition and interaction model but the challenges are similar.

A. Mircoservice Architecture

Microservice architecture [4] decomposes an application into a set of narrowly focused and independently deployable services, a.k.a. microservices [1], which may communicate among themselves using lightweight mechanisms, such as REST APIs. Some examples of popular online services using this design include Netflix [5], Ebay [6], and Gilt [7]. In contrast, traditional internet applications are often designed using a three-tier model with a monolithic application implementing all of the business logic, as shown in Figure 1(a). In this example, all the logic for renting DVDs runs in a single process and is deployed as a single executable (i.e., a WAR file). For scalability, multiple application instances are deployed horizontally behind a load-balancer.

There are three problems with monolithic applications. First, different components of the application have different scaling requirements – for instance, creating new customers is less frequent than customers renting DVDs. However, scaling a monolithic application requires the entire application to be replicated, which requires greater resources. Second, monolithic architecture often means technology lock-in—it is difficult for application components to evolve separately and adopt new technologies (e.g., new databases, new programming languages). Moreover, a small change made to the application requires the entire application to be re-built. Finally, as the application becomes more complex, it is often difficult to separate out DevOps responsibilities, which results in slow development and deployment.

Microservice architecture addresses these problems by decomposing a complex monolithic application into a set of small and autonomous services that work together. In Figure 1(b), the DVD rental application is broken into many small and decoupled tasks, and each task is implemented by a small service. This decomposition allows different services to be built, deployed, managed, and scaled independently. During this decomposition, all the function calls across components are replaced by inter-service communications that are implemented via well-defined API interfaces, as illustrated in the Figure 1(b) using connections.

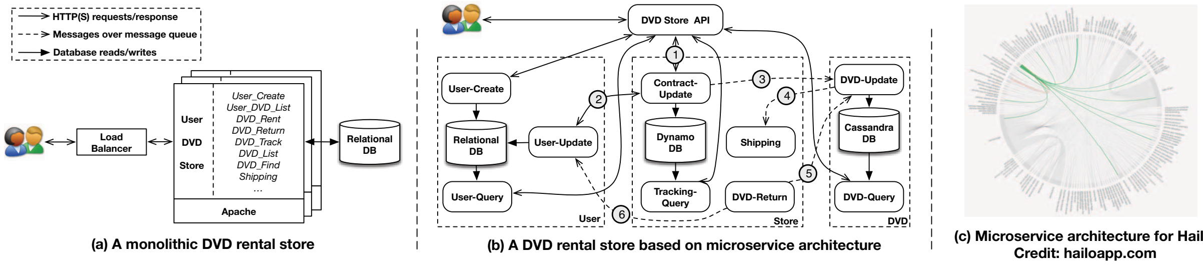
B. Security Issues

Microservice architecture does not make an application any simpler, it only distributes the application logic into multiple smaller components, resulting in a much more complex network interaction model between components [8], which is evident even in this simplified example in Figure 1(b). When a real world application is decomposed, it can easily create hundreds of microservices, as seen in the architecture overview of Hailo, an online cab reservation application, depicted in Figure 1(c). Around the circle are microservices and the lines are the communications between them. The security challenge brought by such network complexity is the ever-increasing difficulty in debugging, monitoring, auditing, and forensic analysis of the entire application. Since microservices are often deployed in a cloud that the application owners do not control, it is difficult for them to construct a global view of the entire application. Attackers can thus exploit this complexity to launch attacks against applications. Current cloud platforms lack a mechanism to assist application owners to effectively collect and monitor the interactions among distributed microservices in order to have a better visibility of the application.

Another security concern involves the trust among the distributed microservices. An individual microservice may be compromised and controlled by an adversary. For example, the adversary may exploit a vulnerability in a public facing microservice and escalate privilege on the VM that the microservice runs in. As another example, insiders may abuse their privileges to control some microservices. As a result, individual microservices may not be trustworthy. However, current applications often assume a TCB including all their microservices. Consequently, adversaries controlling one microservice may propagate their attacks through the connections among microservices and bring down the entire application. For example, in the DVD rental application, a compromised `Contract-Update` may send modified requests to `User-Update` to cause user account to be arbitrarily charged. A compromised `DVD-Update` service may consume and then delete messages on the queue without actually shipping out DVDs, causing a denial of service attack, and so on. As a real world example, a subdomain of Netflix was compromised, and from that domain, adversary can serve any content in the context of netflix.com. In addition, since Netflix allowed all users' cookies to be accessed from any subdomain, an adversary controlling a subdomain was able to tamper with authenticated Netflix subscribers and their data [9]. Current cloud platforms lack a mechanism for applications to monitor and enforce the connections among microservices to confine the trust placed on individual microservices, limiting the potential damage if any microservice gets compromised.

C. Problem Definition

Threat and Trust Model. In this work, we aim to assist cloud applications to monitor and enforce communication among its microservices. To do this, we assume that the cloud infrastructure (e.g., cloud controller, hypervisor, virtual router/switch) is not compromised. That is, we trust the cloud infrastructure to isolate VMs and we do not consider attacks launched against infrastructure services and components in cloud. In addition, we trust the security appliances and assume they will not be compromised by the data they monitor (e.g., we assume an IDS appliance may not be exploited by the network packets it monitors). However, we do not trust the application microservices or



(a) A monolithic DVD rental store

(b) A DVD rental store based on microservice architecture

(c) Microservice architecture for Hailo. Credit: hailoapp.com

Figure 1. Microservice architecture. Figure (b) shows a lifecycle of DVD rental. Step 1: customer sends a request to `Contract-Update` service to create a rental contract. Step 2: `Contract-Update` invokes a method of `User-Update` to bill the customer's account. Step 3: `Contract-Update` places a RPC message containing the DVD info on the message queue. The message is consumed by `DVD-Update`. Step 4: `DVD-Update` invokes `Shipping` to ship the DVD. Step 5 and 6: When the customer returns the DVD, `DVD-Return` updates customer account and DVD repository.

the VMs that they run in. We assume that adversaries may have complete control over the VM after compromising a microservice. Thus, the security guarantee we offer is that the interaction among microservices of an application are completely monitored and mediated according to application's policy, even if individual microservices may be under control of adversaries.

Limitations of Prior Research. Prior research has focused on protecting applications from external threats [10], [11]. They often deploy security services (e.g., IDS/IPS) on network edges in order to monitor traffic that goes into or comes out of an application's private network. However, these approaches cannot address internal threats that come from compromised microservices within the application's private network. There are some prior work [12], [13] that try to extend monitoring to the communications among microservices. However, these approaches often rely on instrumentation of microservices themselves or their host VMs. Consequently, if the entire VM is compromised by adversary, the results collected by the monitoring services will no longer be trustworthy.

We want a solution that can flexibly monitor the network communication between microservices and enforce policies on them in order to detect or prevent both external and internal threats targeting cloud applications. We envision the following requirements for such a solution.

- Completeness: the solution should be able to monitor and enforce over both internal and external network events of a cloud application.
- Tamper resistance: the solution should work even if individual application VMs are under adversary's control.
- Flexibility: the solution should allow applications to specify their own policies over the kind of network events they want to monitor and enforce policies on.
- Efficiency: the solution should have minimal impact on network and CPU resources consumed.

III. Design

To monitor and enforce various network events in microservice applications, two requirements must be met. First, the solution needs to provide *complete mediation*—it must be able to monitor and enforce all security-sensitive network events, both external and internal. Second, the solution must be *tamper-proof*—it must protect itself from adversaries that may control individual microservices and their host VMs.

Our solution leverages modern networking technology (i.e., SDN) to separate the decision about where to place security monitor from the network flows themselves. The insight behind such design is that security monitors no longer need to sit on the network path (e.g., network edge, Application VMs) in order to monitor the network events, since the network connection seen by cloud applications are actually defined by software (i.e., via SDN). Consequently, we can place security monitors in their own VMs, hereafter called *security VMs*, which are deployed just like application VMs in the cloud. As a result, our solution can achieve tamper-proof since security VMs are isolated from application VMs; complete mediation since network events, either internal or external, can be delivered to the security VM via the cloud network that is defined by software; and the flexibility and efficiency since both security VMs and the way network events are delivered can be flexibly decided according to the application's needs.

To demonstrate our approach, consider Figure 1(b). Assume a security monitor is interested in customer requests sent to the `Contract-Update` service and the resulting internal messages sent by the `Contract-Update` service. In this case, the virtual network of the DVD store application can be re-programmed in a way that all the incoming and outgoing network traffic of the VM that hosts `Contract-Update` service are copied and forwarded to the security VM for analysis. Network virtualization based on SDN in the cloud allows the cloud infrastructure to control each and every network packet to or from individual application VMs, consequently providing the completeness guarantee that the security monitor will be able to monitor both external and internal network events. Such a design also lends itself to a tamperproof deployment of security monitors. Since security monitors reside in their own VMs, adversaries cannot evade or tamper with security monitor even if they have complete control over application VMs. In fact, much like a traditional security monitor deployed on network edges, security VMs are transparent to application VMs. Adversaries cannot propagate attacks from application

VMs to security VMs unless they can break VM isolation, which hypervisors are trusted to enforce.

In order to deploy security monitors in VMs, cloud infrastructure must be able to deliver relevant network events to corresponding security VMs. In the remainder of this section, we will describe the architectural support required from cloud infrastructure to do so and how flexibility and efficiency goals can be fulfilled.

A. FlowTap Primitive

Deploying security monitors in VMs requires architectural support from cloud infrastructure in order to deliver relevant network events to corresponding security VMs. The key challenge here is to provide *flexibility*. A naive solution that copies and forwards all network events, both internal and external, to a security VM is unlikely to fulfill the diverse security requirements of all cloud applications. We envision the following scenarios that a cloud application may need a security monitor for:

- An application wants to deploy a security monitor to simply log all internal and external network events seen by its microservices for later forensics purposes.
- An application wants to deploy a security monitor to protect its public facing microservices but trust its internal microservices, in a way similar to traditional security monitors on network edges.
- An application wants to deploy a special set of security monitors to protect certain microservices (e.g., SQL injection detection for DB service).
- An application wants to deploy a security monitor to selectively monitor certain type of network events (e.g., messages sent over message queue) but ignore other types of traffic (e.g., HTTP requests/responses).
- Building on the above scenarios, applications may require security monitors to be able to react to network events differently (e.g., passively monitor vs. block/allow vs. redirect for honeypot analysis).

The architectural support provided by the cloud infrastructure must be flexible enough to support all these scenarios.

The solution we propose is a primitive called *FlowTap*. FlowTap is a contract established between application and cloud infrastructure regarding how to deliver network events. This contract describes the monitoring functionality for each application VM by associating that VM with a security VM and the network events to be monitored, implementing the illusion that the security VM is physically resident on the network channel. FlowTap contracts may specify the types of network events and actions to take upon those events, but different mappings of events/actions may be delivered to different security VMs for the same target VM.

FlowTap (SRC, DST, Flow_Syntax, Action)
Flow_Syntax:
in_port, dl_vlan, dl_vlan_pcp, dl_src, dl_dst, dl_type, nw_src, nw_dst, nw_proto, nw_tos, nw_ecn, nw_ttl, tp_src, tp_dst, icmp_type, icmp_code, table, metadata, vlan_tci, ip_frag, arp_sha, arp_tha, ipv6_src, ipv6_dst, ipv6_label, nd_target, nd_sll, nd_tll, tun_id, tun_src, tun_dst, reg

Flow_Syntax Example (Monitor incoming HTTP requests):
nw_src = 0.0.0.0/0; nw_proto = TCP; tp_dst = 80

Figure 2. A prototype of FlowTap primitive that uses OpenFlow-based parameters for the flow syntax.

The FlowTap primitive is designed to be a cloud API that can be invoked by cloud applications. Figure 2 shows a prototype of the API. It accepts four parameters. The first parameter `SRC` is the target application VM which needs to be monitored. The second parameter `DST` is the security VM that hosts the security monitor. Both `SRC` and `DST` are specified using *port*. Port is an abstract concept in cloud that uniquely represents the connection between a VM and a virtual network.A port may or may not have an IP address. Working with ports, FlowTap primitive allows applications to specify the monitoring relationship without worrying about the actual deployment. For example, target VM may be migrated, suspended or rebooted with a new IP address, but as long as it is plugged into the same virtual network, cloud infrastructure will honor the contract and deliver the network events. Moreover, since port can work without IP address, security VM can be transparent (i.e., not addressable) to application VMs. This further protects security VM from compromised application VMs.

The third parameter `Flow_Syntax` describes a particular *network flow*. Network flow is the basic unit at which the cloud infrastructure handles traffic routing. It is defined using different fields of a network packet as shown in Figure 2. By specifying the `Flow_Syntax`, application can ask cloud infrastructure to selectively deliver specific types of traffic to a security VM. For example, if the application is only interested in monitoring HTTP traffic, it can specify a flow with destination TCP port to be 80. In this way, only HTTP requests/responses sent to or from microservices will be delivered to the security VM, with the rest of the traffic (e.g., database access) untouched. Working at the granularity of network flow, FlowTap primitive allows applications to specify different monitoring strategies with maximum flexibility that a cloud network may support[1].

The last parameter `Action` allows an application to choose different reaction strategies for each type of network event. Currently, FlowTap implements two kinds of strategies, namely forwarding and redirecting. On forwarding, relevant network events will be copied and forwarded to the security VM, with the original network events still delivered to their intended destination. On redirecting, the relevant network events will be directed to the security VM, and depending the decisions made by security monitor, the network events may or may not reach the their intended destination. Forwarding and redirecting essentially implement the passive monitoring and active enforcement respectively.

[1] This is because underlying routing devices in cloud (e.g., virtual switches) also process network traffic at the granularity of network flow.

Multiple FlowTap contracts can be established on the same target application VM with non-overlapping flows. For example, if a microservice both accepts HTTP requests and accesses the message queue (e.g., `Contract-Update` in Figure 1), the HTTP traffic can be forwarded to a Web Application Firewall (WAF) to block external attacks and the message over message queue can be forwarded to an internal security analysis tool to detect internal attacks.

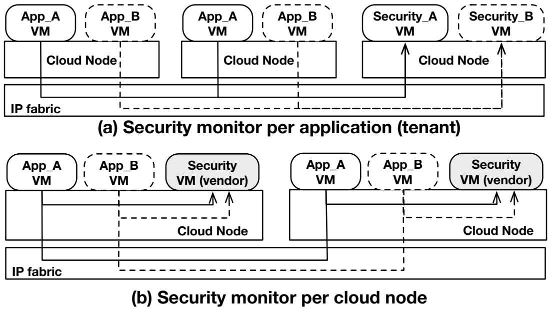
B. FlowTap Compiler

The FlowTap primitive allows traffic slicing at the granularity of flows, which operate at layer 4 and below. To support monitoring policies at a higher level of abstraction, we provide a FlowTap compiler, `ftc`. This tool translates a given set of high level policies – usually provided by the application – to a sequence of FlowTap API calls necessary to implement the policies. For example, a policy such as *redirect all incoming HTTP traffic for public-facing services to a WAF (web application firewall)* will be translated to a sequence of API calls of type $FlowTap(Service\_Port, WAF\_Port, (Proto = TCP \mid TcpDstPort = 80), Forward)$ and is pushed out to all the nodes that host VMs for the particular application. Currently `ftc` supports Datalog [14] as the policy language and is designed to be compatible with the policy language used in OpenStack Congress [15]. In this example, the policy could be written as a Datalog rule, *monitor_http*(*tcp_dport*, $InOut$, vm_1, vm_2) :- *http_In*(*tcp_dport*, $InOut$), *public_vm*(vm_1), *waf_vm*(vm_2), *monHttpAction*(_), where the r.h.s. contains rules to validate the context (the first three terms) and perform the set-up/action (monHttpAction). The details of policy language is left out due to space limitations.

The compiler is designed to generate FlowTap calls according to the network monitoring strategy chosen by the administrator(Refer to Section III-C) in a way that also maximizes the efficiency of the system. More concretely, in addition to policy, `ftc` also takes as input the utilization of the cloud resource (e.g., cloud node CPU usage, network load) and can dynamically compile the same policy into different set of FlowTap calls that maximize the efficiency of the system. For example, using the monitor per cloud node strategy mentioned below, `ftc` may slice the flows monitored internally on a cloud node and forward part of them to security monitor on another node for processing if the CPU of the first node becomes the bottleneck.

C. Monitor Per Application vs. Monitor Per Cloud Node

FlowTap primitive delivers each security VM its relevant network traffic. This unavoidably creates additional traffic on the network. Therefore, the next question is where do we place security VMs such that network monitoring can be efficient for cloud applications.



(a) Security monitor per application (tenant)

(b) Security monitor per cloud node

Figure 3. Two security monitor deployment strategies.

There are two distinct approaches for deploying security VMs, corresponding to two different service models. The first approach is to deploy a security VM per application as shown in Figure 3(a). In this case, each security VM is deployed as part of the application. The service model is self-service, where each application is responsible for building and maintaining its security VMs, setting up the FlowTap contracts, etc. Cloud vendor only exposes the FlowTap API for delivering the network traffic as specified by the application. The benefit of this approach is that it provides applications with maximum flexibility—applications can choose any security monitors and policies they like and can directly manage their security monitors.

However, this approach has a drawback in terms of efficiency. Since security VM may reside on another physical node, relevant network traffic will be delivered over the physical IP fabric. Although it appears to be one flat L2 segment to applications, physical IP fabric is built on top of multiple L2/L3 tunnels, which can degrade network performance. For example, we measured the network throughput for GRE (Generic Routing Encapsulation) tunnels between different cloud nodes in Table I. Results show that it is 5x slower than the virtual network within a cloud node. In addition, Gartner [16] reports that 80% of cloud application traffic are going to be east-west (i.e., from application VM to application VM) instead of north-south (i.e., from end-user to application VM). Consequently, the bandwidth on physical IP fabric will likely become a resource that cloud vendor charge applications for, so applications will want to minimize the utilization of such resource.

Another approach is to deploy a security VM per cloud node, as shown in Figure 3(b). In this case, security VM is deployed as part of cloud infrastructure, running as a security service provided by vendor to applications. The service model is pay-by-use where applications specify their policies for security monitoring, and cloud vendor is responsible for building and maintaining the security VMs, setting up the FlowTap contracts, etc. This approach has two benefits. First, it reduces the traffic on physical IP fabric since now relevant traffic is forwarded to local security VM[2]. Second, this approach offers an opportunity to better utilize cloud

[2] Security VMs on different nodes may still need to collaborate in order to enforce a global policy for an application, but the amount of information necessary to correlate events in a global policy has been found to be orders of magnitude smaller than the raw traffic [17].

resources. Since security monitoring is often CPU intensive, cloud vendor may leverage FlowTap and the compiler to re-balance the traffic forwarding such that security VMs on less loaded nodes get more traffic to monitor.

This approach brings up two challenges. First, application owners should still have the flexibility to deploy customized security policies to protect their applications. In other words, the security monitor deployed by cloud vendor should provide a comprehensive set of security primitives to analyze and operate over any kind of network traffic, and be general enough to support various kinds of security policies. While a specific design of such security monitor is out of scope of this paper, we argue that it is possible. The reason is that network traffic follows protocols. As long as a security monitor can extract fields from a network packet, the content analysis can be application policy specific. For example, the Bro network security monitor [10] already supports plugin policies in terms of Bro scripts to customize network security monitoring. Second, since VMs from different applications may run on the same node and be monitored by the same security VM, it is important that security policies from different applications do not interfere. For example, one application may have a security policy that blocks all HTTP traffic while another application may have a security policy that only allows HTTP traffic. This can be resolved by associating policies to network flows. The insight is that network flows from different applications cannot overlap, so if security policies are associated to flows, they cannot interfere with each other. We rely on the compiler to do so.

In our work, we adopted the second approach where we created a Security-as-a-Service provided by cloud vendor based on the Bro network security monitor [10]. In Section V-A, we demonstrate several security policies we used that can detect and block internal threats from compromised microservices. In general, we envision a hybrid approach will be adopted where certain common security monitors (e.g., malware detection, IP blacklist) will be deployed as a service provided by cloud vendor, but application owners have the freedom to deploy highly customized security monitors specific to their applications. Our FlowTap primitive is designed to be flexible enough to support either deployment case and service model.

IV. Implementation

Our prototype FlowTap is implemented on OpenStack Icehouse release. To implement FlowTap, we modified the virtual routing devices on cloud nodes, including the integration bridge (i.e., br-int) that connects to VMs and the tunneling bridge (i.e., br-tun) that tunnels the VM traffic across cloud nodes. We modified br-int such that when a packet of the target VM is submitted, it is processed through the following steps according to the FlowTap API: 1) the flow is compared with the flow syntax; 2) if it matches, it is duplicated (if the action is forwarding) or taken as it is (if the action is redirecting); 3) its destination MAC address is rewritten to be the MAC of the security VM; and 4) it is resubmitted. It is resubmitted to either a local port on br-int if the security VM is on the same cloud node, or to the br-tun for tunneling. We modified br-tun to establish a tunnel to the remote cloud node that hosts the security VM upon the execution of FlowTap API. The packet will be delivered through the tunnel to the security VM by remote br-int and br-tun based on its destination MAC.

Although the general implementation is straightforward, we run into several interesting issues. First, virtual bridges does not allow the same device port to be used for both incoming and outgoing flow. Consequently, if a packet comes from br-tun to br-int, we cannot duplicate it and send it out to br-tun again. The solution we adopted is to create another patch port between br-int and br-tun such that outgoing flow is separated from incoming flow. However, this creates another challenge. Since now there are two connections between br-int and br-tun, a loop is created. As a result, broadcast packets from a VM can propagate back to itself. This creates a serious issue for DHCP-discovery packets. Since the VM receives its own DHCP-discovery requests, Iptables on a cloud node will set the connection state to be invalid thus preventing the VM from getting valid DHCP-reply. One options is to modify the iptables such that invalid connections states are accepted. Another option is to add new flow tables rules to filter broadcast messages that are looped back. We adopted the second approach.

V. Evaluation

We evaluate our solution by: (1) demonstrate its effectiveness by showing an example network security monitor deployed via FlowTap and how it enforces various security policies to block internal threats of microservices; and (2) investigate the performance impact of FlowTap.

A. Case Study

To demonstrate the effectiveness of our solution, we deployed a Bro network security monitor via FlowTap on cloud. The security monitor enforces over the internal network events, including HTTP, message, and database access, among microservices of the example application shown in Figure 1(b). It enforces the following policies:

Connection Policy. This policy decides whether or not a microservice can have a direct connection to another microservice. For example, neither `Shipping` nor `DVD-Return` needs to have direct access to the database, therefore the policy would deny any connection attempts from these two services to the database. As a result, although they run within the same application network, they can gain no access to critical data even if they are compromised.

Request-specific Connection Policy. This policy defines what kind of request a microservice can make to another microservice. For example, `User-Create` service is allowed

	Node-Node	VM-VM (same node)	VM-VM (different node)
Throughput (mbps)	9200	12000	2600

Table I
NETWORK VIRTUALIZATION PERFORMANCE USING GRE TUNNEL.

to insert an entry into user database, but it is not allowed to make a query of a specific user. The security monitor parses the request body and checks for its legitimacy. Moreover, the security monitor enforces this policy based on user request. A microservice can only make certain requests to others as required for serving a particular user request. Thus, even if a microservice is compromised, it is confined through limiting its connections to other microservices.

Request Integrity Policy. This policy enforces over the content (e.g. checks for certain invariants, correlations, etc.) of a request to prevent compromised microservices from hijacking requests. For example, when user Alice rents a DVD, her request is handled by `Contract-Update`. However, a compromised `Contract-Update` may send requests to `User-Update` indicating user Bob to bill. Similarly, a compromised `User-Update` may modify the database such that the DVD is accounted to Bob instead of Alice. The security monitor enforcing request integrity policy analyzes the body of requests to ensure that the same user is referred to throughout the processing of the user request by multiple microservices. Similarly, the security monitor can check other application specific invariants and correlations to make sure that a compromised microservices cannot hijack how a user request is served.

B. Network Virtualization Performance

Table I shows the overhead associated with the tunneling. In the table, the column Node-Node measures the bare metal bandwidth between cloud nodes, serving as a baseline for comparison. The bandwidth is measured using the Netperf. The VM-VM columns measure the throughput between two VMs running on the same node or different nodes. As shown in the table, due to the GRE tunneling, the network performance in the different node case degrades dramatically, almost 5x slower. The reason is that the offload features on most existing NICs cannot be utilized by GRE outer header computation. Consequently, tunneling becomes a CPU intensive task. Other tunneling techniques such as STT and VXLAN may yield better performance. One interesting observation is that the network throughput between two VMs on the same node is 12000, even larger than the bandwidth for VM hardware (10Gb). This is due to the memory optimization in hypervisor which allows two VMs to exchange network packets very quickly.

C. FlowTap performance

Next, we evaluate the performance impact of the FlowTap. We are interested in knowing how the raw (TCP) network throughput of two communicating microservices is affected due to FlowTap. We consider four different deployment scenarios shown in Figure 4. For all the scenarios, we used Netperf to pump TCP packets with dummy content as fast as it could and used a dummy security VM that would passively read, count, and ignore the packets. We believe, this is the worst case scenario for FlowTap, as there is no computation latency involved for producing or consuming this dummy content. For simplicity, we name the two communicating service VM A and B, and the security VM S. S is setup to passively monitor outgoing traffic of A.

Scenario	(a)	(b)	(c)	(d)
Baseline (mbps)	2600	2600	12000	12000
FlowTap (mbps)	2100	2600	5100	9100

Table II
FLOWTAP PERFORMANCE UNDER DIFFERENT DEPLOYMENT SCENARIOS FOR RAW TCP THROUGHPUT WITH DUMMY CONTENT.

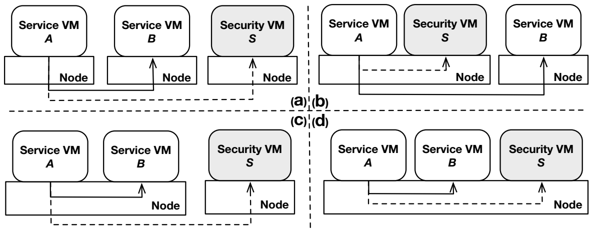


Figure 4. Different scenarios of deploying security VM.

In the first deployment scenario, we deployed A, B and S on different nodes. The baseline throughput between A and B is 2600 mbps, due to the GRE tunnels. With FlowTap, the throughput between A and B drops to 2100 mbps. The reason for the drop is the added latency for encapsulating packets for another tunnel—A to S in addition to A to B.

In the second deployment scenario, A and S run on the same node while B runs on a different node. In this case, the dominant overhead is the tunneling between A and B. Therefore we did not see a throughput drop since the local forwarding from A to S is very fast.

In the third deployment scenario, A and B run on the same node while S runs on a different node. In this case, the throughput between A and B drops from the baseline to 5100 mbps, decreasing by more than 55%. This is because of the tunneling from A to S. Since the packet delivery between A and B is fast, tunneling becomes the bottleneck, decreasing the throughput dramatically. However, we are curious why the throughput between A and B can be higher than the baseline throughput (2600 mbps), since in this case FlowTap also forwards the traffic to S on a different node. By looking at the packet counter, we found that the reason is because packets are dropped while they were forwarded to S. Since S is only a passive listener of packets, A will not stop and wait if buffers on S's host are filled up, thus sending packets at a faster speed than the baseline case.

In the fourth deployment scenario, A, B and S all run on the same node. We see a throughput drop from 12000 mbps to 9100 mbps. Since no tunneling is involved, FlowTap forwarding is the only source of overhead. FlowTap

	Baseline	FlowTap	Throughput loss
Throughput (req/s)	3195	3004	6%

Table III
FLOWTAP PERFORMANCE WHEN MONITORING A WEB SERVER.

currently incurs relatively large overhead due to packet copying and rewriting operations (e.g. rewriting destination MAC address). However, this can be avoided by adding more complex flow rules on virtual devices of cloud nodes such that the packets can be delivered to security VM without being modified. We are currently in the process of optimizing our implementation to address this problem.

As mentioned before, the above measurements are performed on raw TCP traffic between two dummy ends. In real world, the overhead of FlowTap is often amortized by application traffic. For example, we measured a case where external traffic to a web server is monitored by a security monitor on the same node. As shown in Table III, FlowTap causes about 6% throughput drop for the web server, which makes it a practical solution for real world usage.

VI. Related Work

Existing approaches to securing cloud applications fall into two categories. Infrastructure and/or platform-based security approaches, such as VMware vCNS [18], NSX [19], etc., extend the hypervisor/platform to provide distributed, and sometimes inline, monitoring for the cloud applications. They mostly try to implement monitoring techniques inspired by their physical counterparts (e.g. SPAN and/or TAP ports) in distributed virtual switches [20], firewalls [21] and routers [22]. Our work introduces flexibility to these techniques with more fine grained and dynamic control, and augments microservice-specific context to address security issues that are important for this architecture.

Application-based security approaches, such as in Netflix Fido [13] analyze API-level behaviors within cloud applications to build application profiles and then use the profiles to detect anomalous patterns. They, however, have two drawbacks. First, the analysis often uses hooks within the VM or the application to monitor the APIs and other application behaviors. If an adversary successfully compromises a microservice and escalates the privileges to control the VM that hosts the service, it can easily subvert the security of this framework. Second, this approach usually lacks the visibility into the underlying infrastructure, thus may lack capability to respond to the conditions (e.g. they cannot redirect traffic by themselves and need some infrastructure support). In some situations, they may also be susceptible to poor performance, e.g. when sending traffic or application logs to another host for analytics. We, in comparison, take a middle ground and leverage the application context from the RPC calls captured in inter-service communication on top of the monitoring infrastructure and program the infrastructure for enforcing security controls via automated and dynamic response. There are some application health monitoring technologies [23], but they often focus on individual applications.

VII. Conclusion

In this paper, we presented *FlowTap*, a primitive for the cloud infrastructure that enables it to support fine grain virtual network monitoring. FlowTap establishes monitoring relationships between microservices and security monitors, allowing them to enforce policies over the network traffic seen by the microservices. An empirical study shows that the FlowTap primitive is flexible enough to support various kinds of monitoring scenarios and policies with minimal overhead. Using the FlowTap primitive, cloud vendors can thus provide security-as-a-service for cloud applications that are based on microservice architecture.

References

[1] “Microservices,” http://martinfowler.com/articles/microservices.html.

[2] “REST API job trend,” http://www.indeed.com/.

[3] D. R. Engler, M. F. Kaashoek, and J. O’Toole, Jr., “Exokernel: An operating system architecture for application-level resource management,” in *SOSP 95*.

[4] S. Newman, *Building Microservices*. OREILLY., 2014.

[5] “Netflix,” https://www.netflix.com/.

[6] “Ebay,” https://www.ebay.com/.

[7] “Gilt,” https://www.gilt.com/.

[8] R. Annett, *Working with Legacy Systems*. Leanpub, 2015.

[9] “Netflix Responsible Vulnerability Disclosure,” https://help.netflix.com/en/node/6657#participating-security-researchers---2015/.

[10] V. Paxson, “Bro: A system for detecting network intruders in real-time,” in *USENIX Security 98*.

[11] R. Sommer and V. Paxson, “Enhancing byte-level network intrusion detection signatures with context,” in *ACM CCS 03*.

[12] J. Rhee and *et al*, “Uscope: A scalable unified tracer from kernel to user space,” in *NOMS 14*.

[13] “Netflix Fido,” https://github.com/Netflix/Fido.

[14] S. S. Huang and *et al, “Datalog and emerging applications: An interactive tutorial,” in* ACM SIGMOD 2011.

[15] *“OpenStack Congress,” https://wiki.openstack.org/wiki/Congress.*

[16] *“Facebook data center report,” http://blogs.gartner.com/andrew-lerner/2014/11/17/facebook/.*

[17] *M. Vallentin and* et al*, “The nids cluster: Scalable, stateful network intrusion detection on commodity hardware,” in* RAID 07.

[18] *“VMware vCloud Networking and Security Overview,” https://www.vmware.com/files/pdf/products/vcns/vmware-vcloud-networking-and-security-overview.pdf.*

[19] *Koponen and* et al*, “Network virtualization in multi-tenant datacenters,” in* NSDI 14.

[20] *“vSphere,” http://www.vmware.com/products/vsphere/features/distributed-switch.*

[21] *S. Ioannidis and* et al, “Implementing a distributed firewall,” in *ACM CCS 2000.*

[22] “DVR,” https://wiki.openstack.org/wiki/Neutron/DVR.

[23] “Amazon CloudWatch,” http://aws.amazon.com/cloudwatch/.