AL-SAFE: A Secure Self-Adaptable Application-Level Firewall for IaaS Clouds

Anna Giannakou, Louis Rilling, Jean-Louis Pazat, Christine Morin

AL-SAFE: A Secure Self-Adaptable Application-Level Firewall for IaaS Clouds

Anna Giannakou*, Louis Rilling†, Jean-Louis Pazat‡, and Christine Morin*
*Inria, IRISA
†DGA
‡INSA, IRISA

***Abstract*—Application-level firewalls filter traffic based on a white list of processes that are allowed to access the network. Although they have a complete overview of the system in which they are executed, they can be easily bypassed by knowledgable attackers. In this paper we present AL-SAFE, a cloud-tailored application-level self-adaptable firewall which combines the high degree of visibility of an application-level firewall with the isolation of a traditional standalone firewall. AL-SAFE is able to filter traffic at two distinct points in the virtual infrastructure and adapt the enforced rulesets based on changes in the virtual infrastructure topology and the list of services running inside the virtual machines. Our performance analysis shows that AL-SAFE imposes a tolerable delay to legitimate network connections while it is able to filter out all unauthorised packets.**

I. Introduction

Key IaaS clouds features include multi-tenancy, elasticity and on-demand availability. This allows tenants to manage resources dynamically and create, destroy or reconfigure virtual machines automatically. However, the same features also affect the ability of a security monitoring framework to successfully detect attacks in outsourced information systems [1] and sometimes introduce new security vulnerabilities.

Large-scale security frameworks include several components (firewalls, intrusion detection systems, log collectors etc.) that are located in different areas or even outside the virtual infrastructure. Consequently, changes in the number of virtual machines (VMs) (e.g. addition of new instances) or in their placement (e.g. live migration) require the security monitoring system to be adapted. To be able to cope with the high frequency of such changes, a security monitoring framework should be able to adapt its components automatically. To this end, we introduced SAIDS [2], a self-adaptable security monitoring framework, that can automatically adapt a network IDS to virtual infrastructure changes.

In this paper, we focus on application-level firewalls, which are another type of security device in IaaS clouds.

In contrast to typical host- or network-level firewalls which filter network traffic based on a list of rules that use IP addresses and ports, application-level firewalls operate based on a white list of processes that are allowed to access the network. This fine-grained filtering is achievable because application-level firewalls run inside the host operating system, and thus have a complete overview of the running applications.

Unfortunately, in the conventional design of application-level firewalls, isolation between the firewall and vulnerable applications is provided by their OS kernel, whose large attack surface makes attacks disabling the firewall probable. Hence, we address the following challenge: *Can we keep the same level of visibility while limiting the attack surface between infected applications and a trusted, application-level firewall?*

To address this challenge we designed and implemented AL-SAFE, a two-level application-level firewall that operates outside of the virtual machine it is monitoring, in a completely separate domain. By leveraging virtual machine introspection we retain the same level of "inside-the-host" visibility while introducing a high-confidence barrier between the firewall and the attacker's malicious code. AL-SAFE is able to reconfigure the enforced ruleset based on changes in the virtual infrastructure topology (virtual machine migration, creation, deletion) and in the list of services running inside the deployed VMs.

AL-SAFE consists of two components operating at distinct infrastructure locations: an edge firewall, that filters network traffic between the outside world and the cloud infrastructure, and a local switch-level firewall, that filters traffic in the local switch of each physical host. Both components, executed outside the untrusted virtual machine, become application-level firewalls by using virtual machine introspection.

Our contributions include the design of the self-adaptable introspection-based application-level firewall and a comprehensive performance and correctness evaluation of our design.

The rest of the paper is organised as follows: Section II presents related work. Section III outlines the main objectives of AL-SAFE and describes its architecture. Section IV presents our prototype implementation while Section V describes a thorough performance and security evaluation of AL-SAFE. Finally, Section VI concludes the paper with future work.

II. Related Work

In this section we discuss cloud-tailored firewalls and security applications that leverage the isolation properties offered by virtual machine introspection. Virtual machine introspection is a mechanism that allows, through memory mapping, indirect inspection of and control over the current state of a virtual machine from software running outside of the virtual machine. The approach is based on building higher-level semantics (data structures, files) that can be accessed by a monitoring application from the mapped memory pages.

Virtual machine introspection was introduced first by Garfinkel and Rosenblum in [3], and has been used to

implement secure host-based IDSes like Livewire [3] and IntroVirt [4]. The main idea is to locate the IDS in the hypervisor, and to let it monitor the operating system and applications of a VM using introspection. These IDSes do passive network monitoring of the VMs but do not however, address dynamic changes in the virtual infrastructure. AL-SAFE is able to handle dynamic infrastructure changes and automatically adapt the enforced ruleset.

Amazon EC2 provides IaaS cloud tenants with firewalls, named security groups [5], that are located outside the virtual machines and offer protection from attacks originating outside the cloud infrastructure by filtering inbound traffic only. The security groups are oblivious to the type and nature of the tenant applications thus making fine-grained application-based network traffic filtering impossible. Furthermore it is unclear whether the cloud provider adds rules to the security groups. Web Application Firewall [6] is another Amazon product offering rule-based protection against specific types of attacks (i.e SQL injection) and traffic filtering based on access control lists. AL-SAFE offers a better service regarding traffic filtering rules, because ports are only open when the application is running, and filtering is not limited to Web applications.

In xFilter [7] virtual machine introspection is used to create a self-protection mechanism against stepping-stone-attacks [8]. The outgoing packets are filtered in the hypervisor. xFilter cannot handle dynamic infrastructure changes (such as VM migration) and only filters outgoing traffic. AL-SAFE filters both incoming and outgoing connections and handles dynamic infrastructure changes by reconfiguring the filtering rules.

Our goal is to design a self-adaptable introspection-based application-level firewall that can filter inbound and outgoing traffic based on a white list of applications that are allowed to access the network. We leverage VM introspection for retaining the high degree of visibility of an application-level firewall while making it tamper-resistant to any user- or kernel-level malware located in the monitored VM.

III. Design of AL-SAFE

We propose a two-level introspection-based application-level firewall with a self-adaptable ruleset. The reconfiguration of the enforced ruleset depends upon changes in two different layers: service layer (addition or removal of services) and virtual infrastructure layer (e.g. VM creation, deletion or migration). In this section we outline AL-SAFE objectives together with our threat model and a detailed description of AL-SAFE architecture.

A. AL-SAFE Objectives

AL-SAFE should satisfy the following properties:

- **Self-adaptation**: the enforced ruleset should be configured with respect to dynamic changes that occur in a cloud environment, especially virtual infrastructure changes like VM creation, deletion and migration.
- **Service-based customisation**: the ruleset on both firewall levels should be configurable to only allow network traffic that originates from and to the tenant-approved services.
- **Tamper-resistance**: AL-SAFE should continue to operate reliably even if an attacker gains control of a monitored VM. In particular, the reconfiguration of the enforced ruleset should not rely on information originating from components installed inside the monitored guest.
- **Cost minimisation**: the overall cost in terms of resource consumption must be kept at a minimal level both for the tenants and the provider. This is achieved by sharing AL-SAFE's components between tenants.

B. Models

We briefly describe the system model of the cloud infrastructure and the threat model considered for AL-SAFE.

1) System model: We consider an IaaS cloud with a cloud controller that has a global view of the system. An Adaptation Manager [2] is located inside the cloud controller and is responsible for reconfiguring the installed security probes in the event of a change in the virtual infrastructure topology (VM migration, creation, deletion) or an update in the list of potentially running services (addition or removal). Customers pay for resources that are part of a multi-tenant environment based on a Service Level Agreement (SLA). Each customer is in control of an interconnected group of VMs that hosts tenant-selected services.

2) Threat model: We consider software attacks only, that originate both from malicious VMs as well as from outside the cloud infrastructure. Attackers can install malicious software on a victim VM by exploiting a software vulnerability in an application or the operating system. The exploit can execute both at user or kernel levels. The installed malware can get full control of the VM and use the network. However we consider that the provider infrastructure is secure. Malicious code cannot be injected at any part of the hypervisor or AL-SAFE's components.

C. Two-level Firewall Architecture

AL-SAFE is a secure, introspection-based, two-level, application-level firewall. It extends the SAIDS [2] self-adaptable security monitoring framework for IaaS clouds with a new type of security device. AL-SAFE rulesets are automatically reconfigured upon changes at two levels: the cloud infrastructure and the list of services running inside the monitored VM. AL-SAFE consists of five main components depicted in Figure 1: the Introspection component (VMI), the Information Extraction Agent (IEA), the Rule Generators (RG), the edge firewall (EF), that filters network traffic between the outside world and the cloud infrastructure, and a local switch-level firewall (SLF), that filters traffic in the local switch of each physical host. All components are run by the provider.

The introspection-based self-adaptation process is executed periodically as follows: first, the VMI introspects the memory of the monitored guest to obtain the list of processes attempting to access the network. Second, the IEA extracts the information for generating filtering rules and propagates it to the two Rule Generators. Finally the RGs create the switch-level and edge firewall rules and inject them in the firewalls.

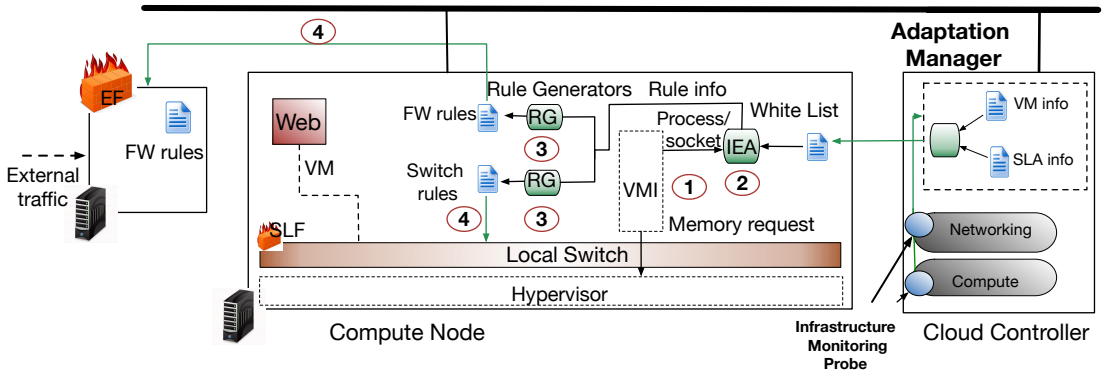

Figure 1. Flow of adaptation in AL-SAFE

The IEA takes as a parameter a tenant-defined white list of processes that are allowed to access the network.

The ***Introspection*** component is able to coherently access the VM's physical memory and uses a profile of the VM's operating system's kernel to interpret its data structures. Thus VMI first extracts the list of running processes, and then iterates over this list to check if a network socket figures in the per-process list of file descriptors. For each network socket found, VMI extracts the process name, the pid as well as source and destination port, IP address and protocol.

The introspection period is defined by the tenant and can be dynamically adapted by the Adaptation Manager [2].

The ***Information Extraction Agent*** compares the list of processes that attempt to access the network (connection list thereafter) with a tenant-defined white list of processes (white list thereafter). The Adaptation Manager [2] is responsible for sharing the white list with the Information Extraction Agent through a secure channel. The IEA allows connections from the connection list that figure in the white list, and blocks all other connections. The IEA propagates the connection information together with an ALLOW or BLOCK action to the ***Rule Generators***. Each RG creates the corresponding rule using all propagated information such as source port, source IP address, destination port, destination IP address and protocol. The generated rules are then injected in the two firewalls.

The ***Switch-level firewall*** filters network packets in the local switch using a list of ALLOW and BLOCK rules. At the switch level, filtering blocks malicious traffic originating from inside the cloud infrastructure at an early stage, thus reducing the load of monitored traffic in the remaining security devices. On a change in the virtual infrastructure topology (e.g. VM migration) the Adaptation Manager stops the periodical introspection process and oversees that the last valid information set from the IEA along with the introspection period is sent to the destination node. The destination's RG creates the switch-level rules and inserts them into the local switch.

The ***Edge firewall*** is located at the edge of the virtual infrastructure in a separate network device. The location of the EF ensures that external malicious traffic will be blocked at an early stage while there will be no performance penalties in the deployed VMs.

IV. Implementation

We implemented a prototype of AL-SAFE using the KVM hypervisor in a private cloud. We used OpenStack [9] as the cloud management system and Open vSwitch [10] as a multilayer virtual switch. A description of the implementation for each component of AL-SAFE follows.

- For VMI we used the LibVMI [11] introspection framework. In order to provide a coherent view of the VM's memory to the introspection process, we take a memory snapshot of the monitored guest before each introspection. LibVMI uses KVM's built-in memory access mechanisms in order to snapshot the VM by copying the whole memory of the introspected guest on a temporary file. During the snapshot the VM is paused and cannot make forward progress. To correlate running processes and open sockets inside the monitored VM we deployed the Volatility Memory Forensics [12] framework and used its driver for the Ubuntu 13.10 Linux kernel. Volatility can support any kernel version provided that a profile with the kernel symbols and data structures is created. The cloud provider would have to maintain a profile for each OS version deployed on the monitored VMs. A wrapper triggers introspection periodically and can adapt the introspection period dynamically. This wrapper is implemented in Python.
- To enable VMI on dynamic infrastructure changes (e.g VM migration), notifier hooks were placed inside the Nova function *pluggin_vifs()* that is responsible for creating the virtual interface(s) for the VM. The hooks pass all necessary information to VMI (VM name, id, version of running OS, etc) and start it immediately after the VM is resumed. All hooks are implemented in Python.
- For the switch-level firewall we use the stateless filtering capabilities offered by Open vSwitch while for the edge firewall we rely on the Nftables [13] stateful packet filtering framework deployed in a standalone Linux host.
- Both the IEA and the RGs are implemented in Python. The SLF RG produces OpenFlow filtering rules while the EF RG produces Nftables-compatible rules. The created rulesets are injected in parallel in the SLF and EF.

V. Evaluation

In this section we present a detailed evaluation focusing both on cost and correctness aspects of AL-SAFE. Since cost minimisation is one of AL-SAFE core objectives we evaluate the associated performance impact of deploying our firewall architecture both from the provider's and tenant's perspectives. Our experiments aim at identifying the overhead induced on normal cloud operations (such as VM migration) and also in tenant applications running inside untrusted VMs. We evaluate our approach under different scenarios that include tenant applications that are either process- or network-intensive. To identify the overhead of self-adapting the two-level application-level firewall on individual connections we utilise micro-benchmarks that calculate the setup time for TCP and UDP connections. In addition to the performance overhead of deploying AL-SAFE, both for tenants and the provider, we also calculate the average resource consumption in terms of RAM and CPU for the Introspection component of AL-SAFE. Section V-A focuses on the overhead of AL-SAFE in virtual machine migration while Section V-B focuses on the overhead in tenant applications. Section V-C presents the results of the micro-benchmarks while we show the resource consumption of our framework in Section V-D. Finally Section V-E validates the correctness of our approach.

To do our experiments we deployed a datacenter with three physical hosts: one cloud controller and two compute nodes. Each physical host has 48GB RAM and runs a 64bit Linux Ubuntu 14.04 distribution. The machines are interconnected with a 1Gb/s network. All the VMs deployed on the compute nodes run a 64bit Linux Ubuntu 13.10 distribution with 2 cores and 2GB RAM. We also deployed the Nftables firewall in a separate physical host with the same hardware as our cloud nodes. All reported results are compared to a baseline value obtained without AL-SAFE.

Before running our experiments we conducted a preliminary set of tests to calculate the time for a full snapshot of a 2GB VM's memory. We calculated the mean snapshot time value to 1.5 seconds. Since the technique used copies the whole memory of the VM into a dedicated file the size of the VM is the only factor affecting the snapshot time.

A. Overhead in Normal Cloud Operations

To calculate the overhead of AL-SAFE in cloud operations we calculate the time required to migrate a VM between two nodes. We choose migration as the most representative cloud operation as it addresses simultaneously the cases of VM creation and deletion. In the case of a migration the running introspection process is stopped and the last valid result is sent to the destination node. Moreover, only the switch-level firewalls (both at the source and the destination nodes) need to be reconfigured. We calculate the migration time under two different scenarios: an idle VM and a VM that runs a memory-intensive workload. We aim at proving that the time required to reconfigure the switch-level firewall is independent from the VM workload. To generate the required workload we utilised *bw_mem_wr* from the LMBench benchmark [14] suite with a 1024MB working set.

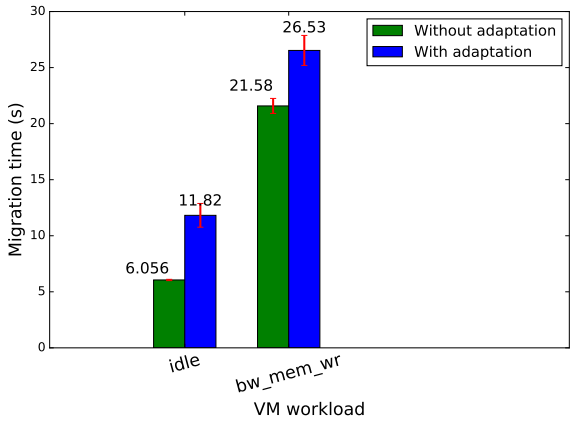

Figure 2. Migration time with and without adaptation

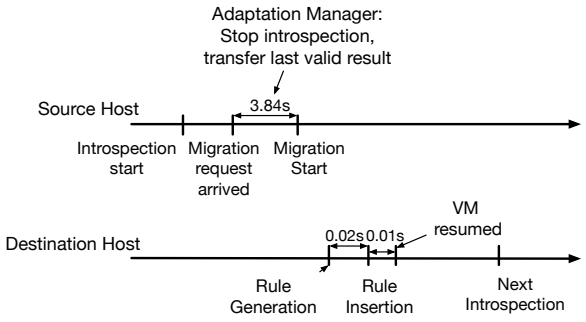

Figure 3. Breakdown of each phase in seconds

The results are presented in Figure 2. The imposed overhead in the migration operation is the same in both cases, which validates our hypothesis that the cost of adapting the firewall ruleset is independent from the VM workload.

A per-phase breakdown of the produced overhead is shown in Figure 3. We insert two rules per service in the switch-level firewall (one for ingress and one for egress traffic) and only one rule in the edge firewall. The relatively high amount of time (3.84 seconds) required for the Adaptation Manager to notify the Introspection agent of the source node, in order to interrupt the introspection and send the last valid results, is mostly due to a defect in the DNS configuration resulting in slow SSH connection establishment. This issue is fixed in the last version of the prototype.

B. Overhead in Tenant Applications

To evaluate the performance overhead of our approach as perceived by tenant applications running inside the untrusted VM, we deployed three different scenarios with distinct application profiles: for a process-intensive application we used a parallel build of the Linux kernel while for a network-intensive application we installed the Apache Web server [15] and we used ApacheBench [16] to generate different workloads. Finally to calculate the overhead of introspection on network

throughput we use Iperf [17]. The first scenario shows how the introspection period affects the execution time of the application whereas the second scenario shows the relation between the arrival of the requests in the introspection cycle and the server latency.

1) Linux Kernel: In this scenario we compiled a Linux kernel inside the untrusted VM and we varied the introspection period. The kernel was compiled with a configuration including only the modules loaded by the running kernel of the VM, using gcc 4.8.4 with a degree of parallelism of 3. The VM is not expected to start services that use the network during the execution time of the experiment thus no adaptation of the firewalls is required. The mean value over five repetitions is shown in Figure 4. The results clearly demonstrate a dependency

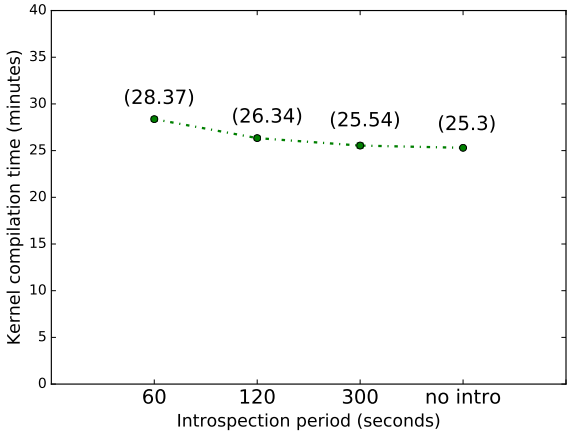

Figure 4. Impact of the introspection period on kernel compilation time

between the period of introspections and execution time. The highest overhead (12%) is observed when the introspection period is 60 seconds. This is because each introspection requires a snapshot of the running VM which freezes the VM for a short period of time. Evidently, more introspections requires more freezing time for the VM, which requires longer execution time.

2) Apache Web Server: In this scenario we examine two aspects of our design: first the dependency between the introspection period and the Web server throughput and second the dependency between the arrival of the connection request in the introspection cycle and the Web server latency. The second aspect shows the impact of using periodic introspection on the availability of a new Web server instance, like in a cloud scale-up operation. For both aspects the client is located outside the virtual infrastructure. For the first aspect no adaptation of the firewalls is required (a preliminary phase to allow the connection between the server and the client is executed), while the only varying parameter is the introspection period. We run the experiment for 3 minutes and register the result. The workload consists of 750,000 requests from 1000 concurrent clients.

The results shown in Figure 5 validate our previous observation regarding introspection period and performance degradation. In this scenario, the highest number of introspections

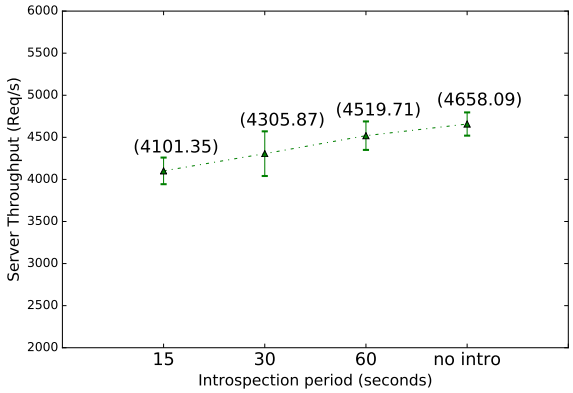

Figure 5. Impact of the introspection period on server throughput

(20 for the 15 seconds period) imposes the highest cost in the server's throughput (12%).

For the second aspect we fix the introspection period at 30 seconds and we start the Web server at port 80 between two introspections. Thus an adaptation of both firewalls is required in order to allow the connections from the client to pass unimpeded. In this experiment we vary the time of the connection request (right before introspection, in the middle of introspection, at the end of introspection and after introspection). The workload consists of 50,000 requests from 1000 concurrent clients. The results are shown on Figure 6.

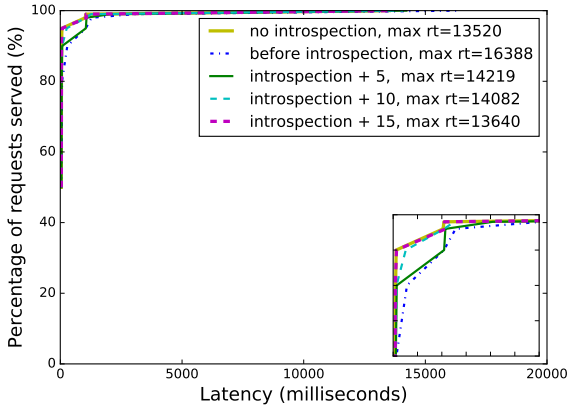

Figure 6. Request service time for different times in the introspection cycle

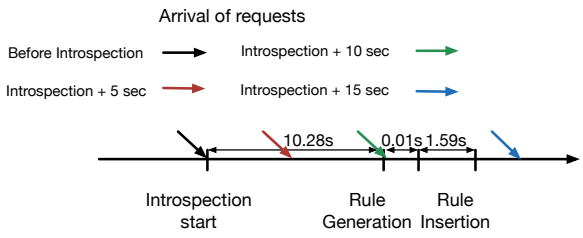

Figure 7. Cases of request arrival time with respect to the introspection cycle

The highest value for the maximum response time (16,388 milliseconds) is observed when the client requests are issued

right before the introspection takes place. Indeed, in order to establish the connection, the client application has to wait for the introspection to be completed, the rules to be generated by the two separate rule generators and then injected in the two firewalls (two rules per service for the switch-level firewall and one for the edge firewall). A per-phase breakdown of the produced overhead is shown in Figure 7. When the requests are issued at the end of introspection, we observe a maximum response time of 14,082 ms. The produced overhead (compared to a maximum rt of 13,520 ms without introspection) results from the time required to reconfigure the two firewalls. The time required to reconfigure the edge firewall is significantly larger than the one for the switch-level firewall due to the establishment of a secure connection between the node that hosts the VM and the firewall node, as explained in V-A.

3) Iperf: In this scenario we install Iperf in the untrusted VM which acts as a server and we utilise a separate host outside the cloud infrastructure as a client. Iperf measures the maximum available bandwidth on an IP network. Before the experiment is executed we run a preliminary phase that reconfigures both firewalls to allow the connection, such that no adaptation is taking place during the experiment. We execute the experiment for 300 seconds. The mean results over

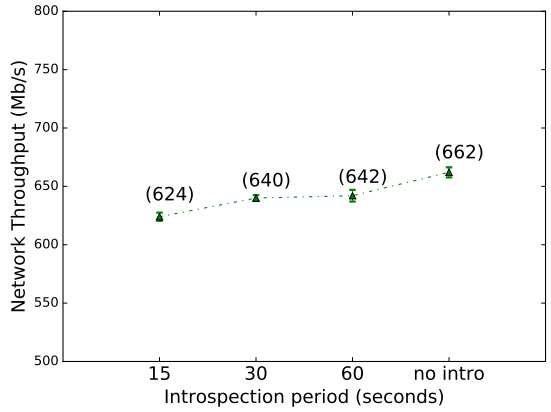

Figure 8. Impact of the introspection period on network throughput

five repetitions are shown in Figure 8. The results confirm our previous observation regarding introspection period and performance overhead. Indeed a shorter introspection period results in more snapshots that evidently result to more downtime for the VM. In this case the highest overhead (5.75%) is observed in the 15 seconds case (20 snapshots).

C. Micro-benchmarks

We examine the effect of AL-SAFE on a single TCP or UDP connection. For the TCP connection we created a simple TCP client-server program to measure TCP connection times. We examine both inbound and outbound connections. For UDP communications we measure the time to transmit a small block of data and receive an ECHO reply (round trip time). In both cases the introspection period is 15 seconds. Two rules per service are inserted in the switch-level firewall and one in the edge firewall.

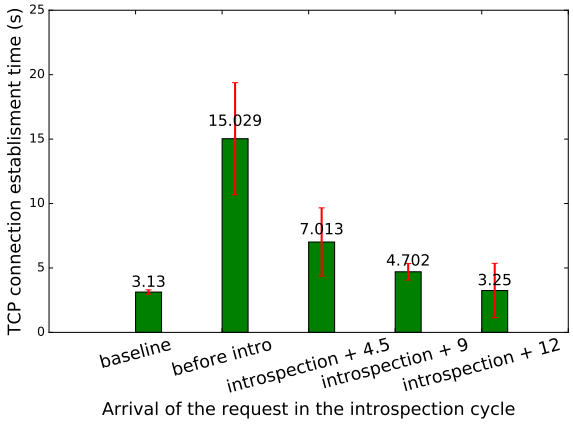

Figure 9. Inbound TCP connection establishment time

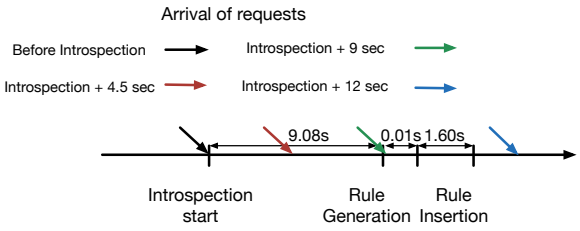

Figure 10. Cases of TCP request arrival time with respect to the introspection cycle

1) Inbound TCP connection: In this scenario we install the TCP server in the monitored guest (two rules in the switch-level firewall are needed and only one in the edge firewall in order to allow the communication with the server). The results are shown in Figures 9 and 10. The case with the smallest overhead is when the request is issued at the end of introspection (introspection + 9 sec, thus the only delay is due to firewall reconfiguration). As explained in Section V-B2 the observed overhead is due to the necessary firewall reconfiguration.

2) Outbound TCP connection: In this experiment the TCP client is installed in the monitored guest. In contrast with an inbound TCP connection where the connection port is known a priori, an outbound TCP connection, shown in Figure 11, faces the limitation of an unknown port number. Thus initiating a request right before introspection is now the best case scenario with the smallest overhead. In all other cases the time period between the time of the request and the next introspection has to be added to the connection establishment time.

3) UDP round trip time: In this micro-benchmark we install the process receiving the ECHO request inside the monitored guest. The results shown in Figure 12 are similar to the case of inbound TCP connections. A detailed description of the per-phase delays is shown in Figure 13 (two rules inserted in the switch-level firewall and one in the edge firewall).

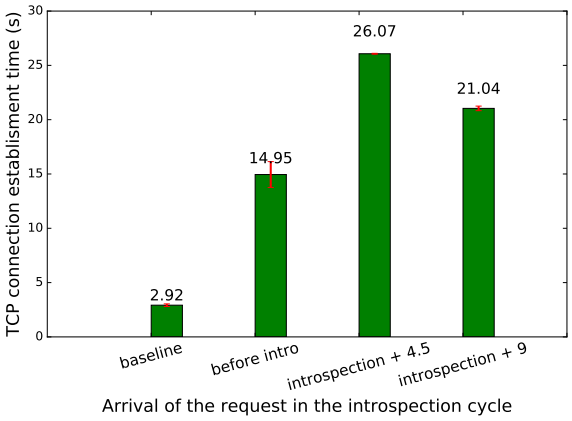

Figure 11. Outbound TCP connection establishment time

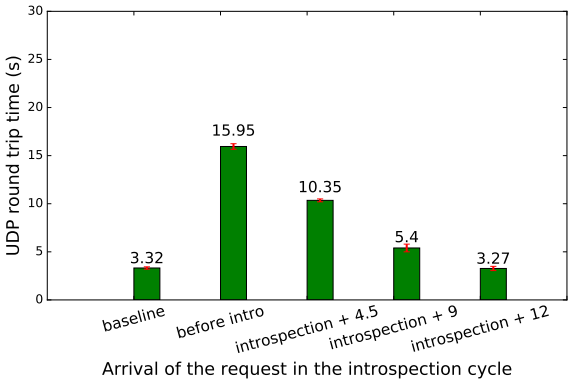

Figure 12. Inbound UDP round trip time

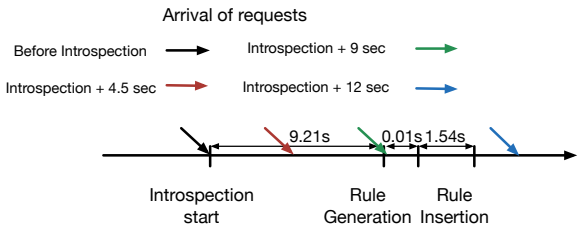

Figure 13. Cases of datagram arrival time with respect to the introspection cycle

D. Resource consumption

In this Section we discuss the cost of AL-SAFE in terms of CPU consumption and RAM. We focus our analysis on the Introspection component of our framework as it is the one expected to consume the most resources. Since the introspection mechanism extracts the necessary information about network sockets by iterating the process list of the running VM it is obvious that the number of processes affects both the execution time of the VMI and the required resources. We calculate the CPU and RAM utilisation of the introspection process in our Web server scenario (Section V-B2), with a generated workload of 750,000 requests from 1000 concurrent clients, over ten executions. Since our Web server is configured with an event-based module, it is expected to generate a lot of child processes, each one handling a pre-specified number of threads. We compare the result with the resources consumed by the VMI in the Iperf scenario (Section V-B3) where only a single process is created to handle the connection socket. The high cost of introspection in terms of memory is because Volatility loads the whole snapshot file (in both cases 2 GB) into memory. Results are shown in Table I.

Table I
RESOURCE CONSUMPTION OF THE INTROSPECTION COMPONENT

Application	*Real (s)*	*Usr (s)*	*Sys (s)*	*CPU%*	*Memory (MB)*
Apache	13.6	5.04	2.21	53.6	2193
Iperf	11.9	3.75	1.60	45	2193

E. Correctness and Security Analysis

One of the main goals of AL-SAFE is to successfully block unauthorised connections. We have validated the correctness of our generated rules both for inbound and outbound connections. For intra-cloud connection attempts the switch-level component of AL-SAFE successfully intercepted all packets from processes that were not in the white list. For extra-cloud inbound connections the packets were stopped by the edge firewall. In both cases no unauthorised packets reached or left the untrusted VM.

In a typical system, software exploits can directly affect the execution of an application-level firewall. Exploits combine network activity from a user-level component along with a kernel-level module that hides the user-level component from the view of the application-level firewall. The malicious exploit likely obtains full-system privileges and can thus halt the execution of the firewall. The malicious kernel-level module can alter the hooks used by the in-kernel module of the application-level firewall so that the firewall is simply never invoked as data passes through the network. Conventional application-level firewalls fail under these types of attacks. AL-SAFE withstands attacks from these types of exploits.

AL-SAFE denies all unknown connections by default. In a production system where services have sufficiently long lifetimes, this tackles the case of an attacker timing the introspection period and attempting to use the network between two consecutive introspections. The performance overhead of this choice on each connection is outlined in Section V-C.

An attacker can hijack a connection after it has been established and verified by AL-SAFE as legitimate. He can use a software exploit to take control of a particular process bound to the port or use a kernel module to alter packets before they are sent out to the local switch network interface. To counter this issue we could place dedicated Intrusion Detection Systems in the infrastructure, using the approach of SAIDS [2].

An attacker who fully controls the VM can also tamper with kernel data structures to control introspection results. To counter such attacks we could use approaches to check the VM's kernel integrity [18], [19].

Finally we analyze the potential vulnerabilities added by AL-SAFE to the provider infrastructure. AL-SAFE's components are exposed to three kinds of potentially malicious input. First, the white list of processes, provided by the tenant, is expressed in a simple tabulation-separated-values text format for which the parser is easy to make robust. Moreover, no complex interpretation is required since the values of each entry match fields of the firewall rules.

Second, network packets are processed by the OpenFlow tables inserted in the local switch, and by the rules inserted in the edge firewall. Assuming that both filtering engines are robust, the added rules can be considered safe since the only actions allowed are to allow or drop traffic.

Third, introspection parses kernel data structures in the VMs in order to extract the list of active network sockets together with their owner process name. The parsing phase relies on commodity tools that may be vulnerable to out-of-bound memory accesses and reference loops in the parsed structures. Out-of-bound accesses can be avoided using languages like Python, used by Volatility. To protect against reference loops, as a last option a timeout could be used to stop introspecting. The extracted information is only compared to the white list of process names or inserted as port numbers (resp. IP addresses) in the filtering rules. It is thus sufficient to check that extracted values are 16 bits integers (resp. valid IP addresses).

VI. Conclusion and Future Work

We have designed AL-SAFE, a secure application-level firewall that is located outside the VM but is able to retain through virtual machine introspection the same degree of visibility as an "inside-the-host" solution. AL-SAFE filters traffic at two distinct points in the virtual infrastructure and is able to adapt the enforced ruleset based on changes in the virtual infrastructure as well as in the list of services running inside the monitored VMs.

We have conducted a thorough evaluation of our approach examining both performance and correctness aspects. We have shown that the overhead in cloud operations such as VM migration is independent from the VM workload. This overhead is lower than the migration time.

Our results show a dependency between the introspection period and the generated overhead for tenant applications running inside the untrusted VM. Increasing the introspection period depending on the type of activity inside the VM (fewer introspections for compute-intensive applications that do not use the network) could significantly decrease the overhead.

In the case of servers opening new ports or applications initiating new outgoing communications, the setup is delayed until the next introspection cycle completes. The duration of introspection cycles could be reduced in two ways. First, instead of establishing a secure connection with the edge firewall at each introspection cycle, we could use a persistent connection. Second, instead of using Volatility to inspect the VM's kernel data structures, we could implement the introspection algorithm directly in LibVMI. Indeed, Volatility implements a very generic Python engine for describing VM data structures, which makes introspection algorithms much slower than C implementations in LibVMI.

AL-SAFE performs introspection periodically, which delays the network connectivity of newly started services and clients. To reduce this overhead, AL-SAFE could introspect on watchpoints, e.g. on listen() and connect() syscalls on TCP sockets.

In order to make AL-SAFE stateful at both levels, we are also improving the prototype using the recently added feature of connection tracking in Open vSwitch 2.5.

We plan to address cost minimisation by combining the security monitoring of tenants and provider infrastructures. We also intend to expand our architecture by including other types of devices such as log collectors and aggregators.

References

[1] N. u. h. Shirazi, S. Simpson, A. K. Marnerides, M. Watson, A. Mauthe, and D. Hutchison, "Assessing the Impact of Intra-Cloud Live Migration on Anomaly Detection," in *Proc. CloudNet*, 2014.

[2] A. Giannakou, L. Rillling, J. L. Pazat, F. Majorczyk, and C. Morin, "Towards Self Adaptable Security Monitoring in IaaS Clouds," in *Proc. CCGrid*, 2015, Doctoral Symposium.

[3] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. NDSS*, 2003.

[4] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting Past and Present Intrusions Through Vulnerability-specific Predicates," in *Proc. SOSP*, 2005.

[5] "Amazon inc." (accessed May 20, 2016), https://d0.awsstatic.com/whitepapers/Security/AWS_Security_Whitepaper.pdf.

[6] "Amazon inc." (accessed September 15, 2016), https://aws.amazon.com/blogs/aws/new-aws-waf/.

[7] K. Kourai, T. Azumi, and S. Chiba, "A Self-Protection Mechanism against Stepping-Stone Attacks for IaaS Clouds," in *Proc. UIC/ATC*, 2012.

[8] S. Staniford-Chen and L. T. Heberlein, "Holding intruders accountable on the Internet," in *Proc. Security and Privacy*, 1995.

[9] "OpenStack," (accessed May 20, 2016), version Juno http://www.openstack.org/.

[10] "Open vSwitch," (accessed May 20, 2016), version 2.9 http://openvswitch.org/.

[11] "LibVmi," (accessed May 20, 2016), version 0.12 https://github.com/libvmi/libvmi/releases.

[12] "Volatility Memory Forensics," (accessed May 20, 2016), version 2.4 http://www.volatilityfoundation.org.

[13] "Nftables," (accessed May 20, 2016), version 0.5 http://netfilter.org/projects/nftables/.

[14] "LMbench - Tools for Performance Analysis," (accessed May 20, 2016), version 3 http://lmbench.sourceforge.net.

[15] "Apache Web Server," (accessed May 20, 2016), version 2.4 http://www.apache.org.

[16] "Apache HTTP server benchmarking tool," (accessed May 20, 2016), version 2.4 https://httpd.apache.org/docs/2.4/programs/ab.html.

[17] "Iperf," (accessed May 20, 2016), version 2.0.5 https://iperf.fr.

[18] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring Operating System Kernel Integrity with OSck," in *Proc. ASPLOS*, 2011.

[19] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes," in *Proc. SOSP*, 2007.