I Heard It through the Firewall: Exploiting Cloud Management Services as an Information Leakage Channel

Hyunwook Baek*
baekhw@cs.utah.edu
University of Utah
Salt Lake City, UT, USA

Eric Eide
eeide@cs.utah.edu
University of Utah
Salt Lake City, UT, USA

Robert Ricci
ricci@cs.utah.edu
University of Utah
Salt Lake City, UT, USA

Jacobus Van der Merwe
kobus@cs.utah.edu
University of Utah
Salt Lake City, UT, USA

ABSTRACT

Though there has been much study of information leakage channels exploiting shared hardware resources (memory, cache, and disk) in cloud environments, there has been less study of the exploitability of shared software resources. In this paper, we analyze the exploitability of cloud networking services (which are shared among cloud tenants) and introduce a practical method for building information leakage channels by monitoring workloads on the cloud networking services through the virtual firewall. We also demonstrate the practicality of this attack by implementing two different covert channels in OpenStack as well as a new class of side channels that can eavesdrop on infrastructure-level events. By utilizing a Long Short-Term Memory (LSTM) neural network model, our side channel attack could detect infrastructure level VM creation/termination events with 93.3% accuracy.

CCS CONCEPTS

• **Security and privacy** → **Distributed systems security**; *Firewalls*; • **Computer systems organization** → **Cloud computing**; • **Networks** → *Cloud computing*;

KEYWORDS

cloud management, cloud security, side channel, OpenStack

1 INTRODUCTION

Resource sharing is a fundamental part of cloud computing. By multiplexing virtual resources (e.g., virtual machines, virtual networks, virtual firewalls, etc.) across an infrastructure, a cloud provider maximizes resource utilization of the infrastructure and offers cloud users flexible scaling of virtual environments with minimal costs.

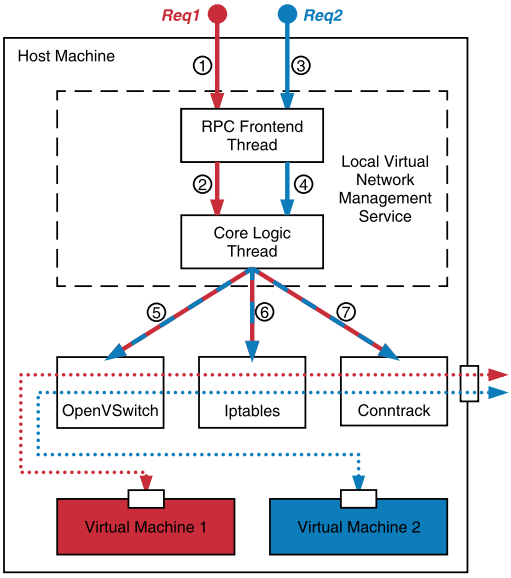


Figure 1: Resource sharing of two requests

However, shared resources also cause interference among cloud tenants and can even be exploited as information leakage channels by malicious users to make critical security breaches. For example, if an attacker's virtual machine (VM) can be successfully placed in a physical machine hosting victim VMs, the attacker VM can exploit such information leakage channels to detect if it is co-resident with a victim [17, 21, 26], to degrade the performance of a victim [20], or to break the virtual isolation and steal confidential information from compromised and non-compromised victims [14, 22, 24, 25, 27–29]. These side/covert-channel attacks are still being actively studied and becoming more feasible and practical.

The underlying mechanisms exploited by the previously studied information leakage channels were mostly limited to hardware architecture-level mechanisms managing a specific set of hardware resources: CPU, L2/L3 caches, memory, and network devices. However, under the hood of a cloud platform, cloud tenants share resources not only at the hardware level but also at the software level: the processes, threads, kernel modules, and networks that

*Currently at Google. Work done at the University of Utah.

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA

form cloud management systems. Especially in cloud management planes, it is commonplace for a single software instance to process multiple requests from different tenants, both at the central cloud controllers and at the distributed cloud management components. For instance, for two co-resident VMs, if each of their users makes a request to connect each instantiated VM connected to a virtual network, the two requests will go through the same virtual network management software instances such as a local cloud management service, a local OpenVSwitch service, and a netfilter kernel module, as illustrated in Figure 1. More importantly, these requests may share some parts of their execution paths as illustrated in steps 5, 6, and 7 of Figure 1 (e.g., due to batch-processing mechanisms for performance optimization). Therefore, the processing times of the two requests may mutually influence each other.

Our key insight is that if a user can keep track of the processing times of his/her own infrastructure-level requests, the user can obtain footprints of infrastructure-level information—e.g., virtual firewall update times or start/end times of other users' VMs. Since this type of information is not obtainable by previously known side channels, the potential impact of this new type of side channel can be significant.

It is challenging to monitor the infrastructure-level activities from a cloud user's side because a user does not have visibility into the cloud infrastructure-level events. One may wonder if the user can utilize the data provided by cloud providers such as APIs for the current virtual resources or event logs [1]. Though it might be possible for some cases, resource-state information offered by the cloud providers is not always based on the actual event times at the edge, so the states of virtual resources provided by the cloud provider may not be consistent with their actual states at the edge [5].

In this paper, we introduce a novel technique to exploit *cloud networking services* (CNS) as an information leakage channel. We manipulate the timing of infrastructure events by requesting, via the cloud provider's API, specific types of modifications to a tenant's virtual firewall rules (often called "Security Groups"). We detect these effects using specially crafted probe packets that monitor changes to firewall state. To demonstrate the utility of this approach, we implement two different classes of CNS-based covert channels for communication between otherwise-isolated tenants and an *Infrastructure Event Tracker* by exploiting the CNS as a side channel in OpenStack. By deploying a deep neural network model utilizing Long Short-Term Memory (LSTM), the infrastructure event snooper detects VM creation/termination events in its host with 93.3% accuracy and predicts their numbers of virtual interfaces with 83.1% accuracy.

To summarize, our contributions include the following:

- We illustrate that the software architecture processing requests from different tenants is exploitable as an information leakage channel through a measurement study in OpenStack's network management stack. To the best of our knowledge, this is the first work that shows the exploitability of shared software resources in cloud management planes as an information leakage channel (§3).
- We devise a novel CNS monitoring mechanism to exploit CNS as an information leakage channel (§4).
- We demonstrate the exploitability of the cloud management services by implementing two different CNS-based covert channels between two isolated VMs (§5.1, §5.2).
- We demonstrate a proof-of-concept of a CNS-based side channel by implementing an infrastructure event snooper to eavesdrop on infrastructure-level events (§5.3).
- We discuss mitigation strategies for these attacks (§7).

2 BACKGROUND

In this section, we briefly introduce the internal architecture of OpenStack to show the fundamental mechanism of the CNS-based information leakage channel.

In OpenStack, a VM has a virtual firewall, which consists of a list of rules specified in the security groups that the VM belongs to. The "security group rules" of a security group are instantiated into firewall rules when the security group is applied to a specific VM. OpenStack implements the virtual firewall using *Linux Iptables* in each host machine. Specifically, in each host machine, there is an OpenStack component called Neutron-OpenVSwitch-agent that is a key CNS component managing most of the host-side network resources, including the iptables. When a user requests the application of a security group to a VM, the cloud controller sends this request to the machine hosting the VM, and the agent in the host retrieves the request and correspondingly updates the iptables.

The main part of the Neutron-OpenVSwitch-agent is an infinite loop that processes RPC requests. The agent does not immediately process a request when the request is received. Instead, it collects requests and periodically iterates over the RPC loop to process the collected requests at once, which is a general strategy for service-oriented architectures [8] to improve throughput and end-to-end latency [7]. The loop iteration period can be configured by changing the value of the variable `polling_interval`, for which the default value is two seconds. (In the rest of this paper, we will assume `polling_interval` is set to be two seconds.) However, this variable does not guarantee that the RPC loop takes exactly two seconds: instead, the system guarantees that each iteration takes *at least* two seconds. This is internally implemented by making the process sleep at the end of each iteration if the iteration took less than two seconds, as shown in the following code:

```
while True:
    start = now()
    ... # process the enqueued tasks
    elapsed = now() - start
    if elapsed < polling_interval:
        sleep(polling_interval - elapsed)
```

Therefore, when a user makes a request to update a VM's firewall, the actual update of the corresponding iptables is unlikely to happen immediately. If the request arrives just before the next iteration starts, the corresponding iptables rule will be created very soon, but it may take a very long time if the previous iteration is not finished. Naturally, if two or more users make requests within the same period and their VMs are hosted by the same physical machine, the requests will be processed together as illustrated in Figure 1, and the latency of the requests will be influenced by each other. Under this situation, if a VM can know the execution duration of the current iteration (i.e., `elapsed` in the code above[1]), it can detect

[1]To avoid confusion, in the rest of this paper, we use the term *elapsed time* only to refer the value of the variable `elapsed`.

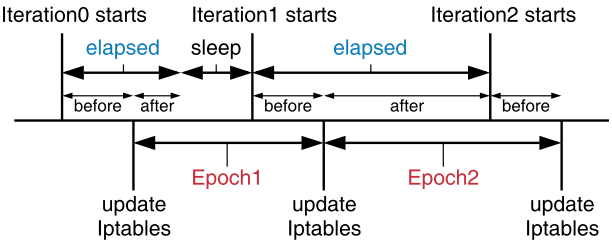


Figure 2: Time line of the RPC loop and Epochs. `before`/ `after` refer the elapsed times before/after the iptables are updated. Note that the k-th Epoch is the Epoch between the $k-1$-th iteration and the k-th iteration.

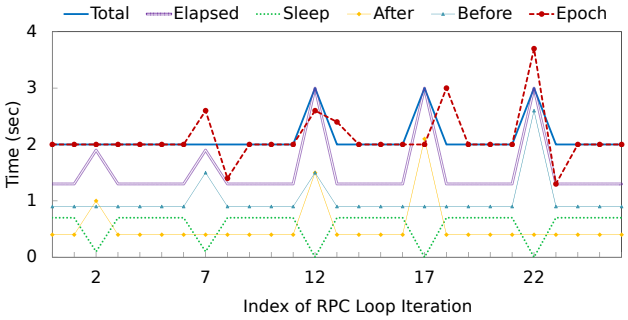


Figure 3: Various impacts of requests on Epochs. `Before` and `After` refer to the elapsed time before and after the iptables are updated. Note that `Before` and `After` show up only if the iptables are updated in that iteration. `Total` is the sum of `Elapsed` and `Sleep`.

when that duration is affected by a high request load. Of course, in reality, a VM cannot directly know the execution durations of the process running in the host. However, as explained earlier, since updating the VM's firewall also happens at some point of the iteration, if the VM can know an interval between two consecutive firewall-updating events, the VM may utilize this as an alternative to the execution duration of the process. We call this interval from an iptables update to the very next iptables update an "*Epoch*." As illustrated in Figure 2, an Epoch may or may not be close to the actual execution duration.

In Section 3, we assess the magnitude of impact of user requests on the execution duration of the shared RPC loop and show the feasibility of exploiting Epoch as an information leakage channel.

3 MEASUREMENT STUDY

Since every request received by the Neutron-OpenVSwitch-agent is processed at some point of the RPC loop, it is obvious that every single request has some impact on the execution duration of an iteration of the RPC loop. However, the impacts of requests may appear in various patterns of Epochs as shown in Figure 3. For example, in the case of the second iteration in the figure, the request increased the elapsed time *after* updating the iptables, but was counter-balanced by the sleep time, so the impact could not be observed between Epochs. In contrast, in the case of the seventh iteration, the request increased the elapsed time *before* updating the iptables, and it made Epochs oscillate noticeably. In addition, as one can see from the next three requests (the 12th, 17th, and 22nd iterations), the impacts of requests on Epochs may appear differently from the impacts on the total execution duration of the iterations. The impact of a request on the elapsed time *before* updating the iptables can influence the size of the current Epoch, but the impact on the elapsed time *after* updating may influence on the size of the next Epoch only if the total execution time of the current iteration exceeds the `polling_interval`.

In this section, through measurement taken in a cloud running OpenStack Mitaka, we show the impact of requests and their combinations on the elapsed times *before* and *after* updating the iptables. To measure the actual execution duration of the RPC loop, we modified the source code of Neutron-OpenVSwitch-agent and Iptables to print timestamps at several points in the RPC loop.[2] We first analyze the general impact of different user-level requests on the elapsed times in Section 3.1. In Section 3.2, we introduce some specific ways to *permanently* increase the elapsed times.

3.1 One-time Impact

We first measured the change in the elapsed times as we made different requests to the shared service. In particular, we measured the elapsed times for four different firewall-related requests: adding a rule, deleting a rule, attaching a security group, and detaching a security group. For the case of adding and deleting a rule, we first made a security group for a VM and measured the time taken to add a rule to (or delete a rule from) the group. Note that, since the Epoch can be properly monitored by a VM only if the VM makes some changes in its own firewall at every iteration, understanding elapsed times of these firewall-related requests is important to build CNS-based information leakage channels. For the case of attaching and detaching a group, we made a group with one rule and measured the time for making the VM be attached to (or detached from) the group. One of the interesting features of these requests is that the adding-a-rule request and the attaching-a-group request yield the same result in the iptables: i.e., for both cases, the iptables chain for the VM will gain a new rule. Likewise, the deleting-a-rule request and the detaching-a-group request also have the same result.

The first four bars in Figure 4 show the elapsed times for the four different requests made by the receiver VM. (The sleep time is not included.) For reference, without any requests, the elapsed time was only around 130 milliseconds. As one can see from the figure, each iteration took around 1,200 milliseconds for adding/deleting a rule, but the elapsed time increased by 1,200 milliseconds for attaching/detaching a group (400 milliseconds for `before` and 800 milliseconds for `after`). We also measured the change in elapsed time when multiple requests are processed together within the same iteration. Making two addition/deletion requests during the same Epoch was not noticeably different from making a single request, as shown in

[2]This modification helped our measurement study, but is not necessary to exploit the side channel "in the wild."

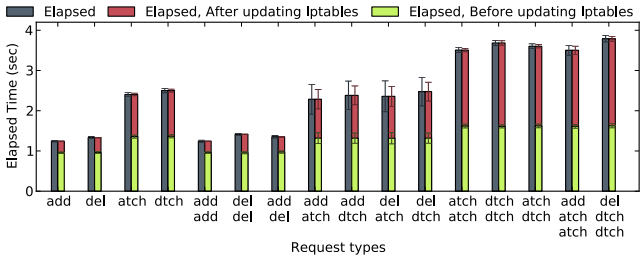


Figure 4: Average and standard deviation of elapsed time for requests (`add`, `delete`, `attach`, and `detach`) and their combinations. Each experiment was repeated 100 times.

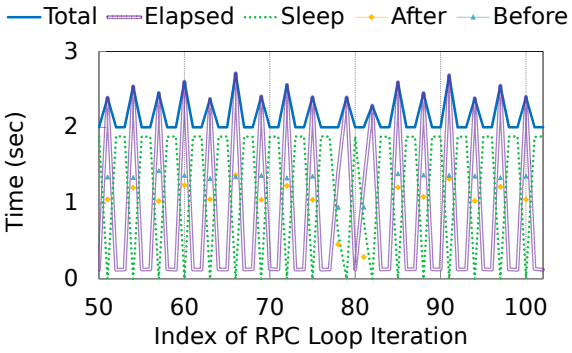


Figure 5: Execution durations of the RPC loop iterations while `add+atch` and `del+dtch` requests are arriving. `add+atch` and `del+dtch` requests were made repeatedly one after another.

the next three bars in Figure 4. Likewise, when we combined one addition/deletion request with one attachment/detachment request, the elapsed time was similar to a single attachment/detachment request. However, when two security group attachment/detachment requests were made in the same Epoch, the elapsed time was increased by 1,100 milliseconds (250 milliseconds `before` and 850 milliseconds `after`). From this result, we can see that in order to monitor *other tenants'* activity (reading from the side channel), addition/deletion requests are more suitable, since their effects are not cumulative with other activity. Likewise, in order to *manipulate* the elapsed time during Epochs (writing to the side channel), attachment/detachment requests are more useful, since their impact is greater and the effects of each individual request are still visible even when combined with other requests.

However, the result also shows potential reliability problems when using attach/detach requests to manipulate Epoch lengths, because they showed high variance when combined with add/delete requests (Figure 4 `add/atch`, `add/dtch`, `del/atch` and `del/dtch`). We analyzed this problem by examining the functions invoked internally in the agent at each iteration. We found that those requests showed high variance in elapsed times not because the elapsed time of the requests themselves are erratic, but because the agent sometimes postponed a portion of the work to the next iteration. One can also see this effect in Figure 5, which shows the actual

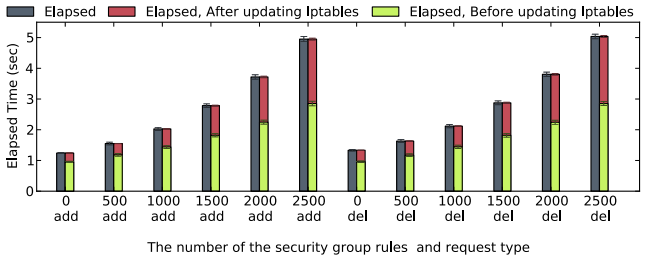


Figure 6: Average and standard deviation (over 100 runs) of elapsed time for `add` and `del` requests as the number of security group rules is increased.

execution durations of the RPC loop while we were measuring the elapsed times for `add/atch` and `del/atch`. Here, at the 78th and 81st iterations, one can see that the requests arrived during those iterations (from the fact that the iptables were updated), but the time-consuming tasks were postponed to the next iterations; the execution durations of the 79th and 82nd iterations were increased instead. Though this happens less frequently as we combine more requests, we could not find a way to completely remove this phenomenon, and thus have to account for it in the construction of our side channel.

We also found the elapsed time for requests can be affected by rules that are already in place. For the add/delete requests, as we increased the number of existing rules in the security group to/from which a rule would be added/deleted, the elapsed time for the requests exponentially increased (linearly in the `before` period but exponentially `after`). Figure 6 shows the result. We also changed the number of rules present in the security groups targeted for attach/detach requests. Surprisingly, for the detaching-a-group request, the elapsed time saw little influence from the number of rules (as shown in the right side of Figure 7). For group-attach requests, only the elapsed time `before` updating the iptables was increased as the number of rules in the group increases, and that time only grows linearly.

From these results, we gained a few insights into exploiting Epochs as an information leakage channel. First, a VM may dynamically manipulate Epochs with various combinations of requests and rules. Second, a VM may extract infrastructure-level information such as types of events (i.e., requests) and the environmental setup regarding the event (e.g., the number of security group rules).

3.2 Long-term Impact

While measuring the impact of the number of rules on Epochs, we found another interesting way to increase the size of Epochs: by giving a VM a large number of rules in a specific way, we could increase the elapsed times for requests against other co-resident VMs belonging to different tenants. We originally observed this phenomenon in an older version OpenStack, Icehouse. In Icehouse, if we increased the number of security group rules for a VM, the elapsed time for *any* firewall-related requests to the same host were increased. In a newer version of OpenStack, Mitaka, it seemed that this phenomenon had disappeared. Even in Mitaka, however, when we attached a security group with many rules and added one

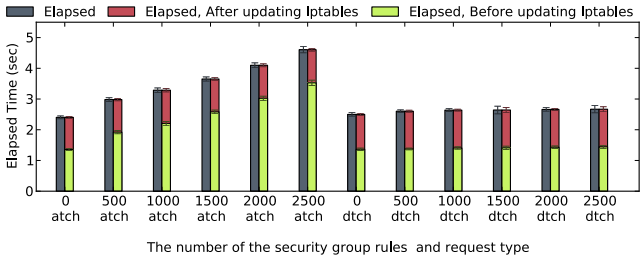


Figure 7: Average and standard deviation (over 100 runs) of elapsed time for `atch` and `dtch` requests as the number of security group rules is increased.

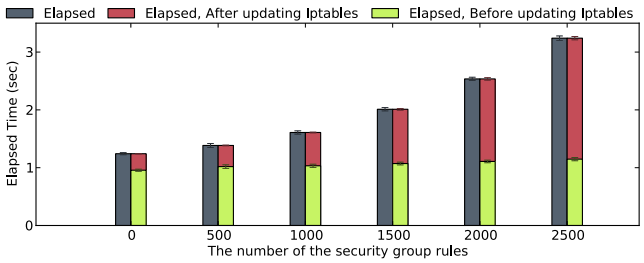


Figure 8: Average and standard deviation (over 100 runs) of elapsed time for `add` requests as the number of security group rules used by the long-term impact request increases.

additional rule to the group, it could increase the elapsed time as we saw in Icehouse. Figure 8 shows the measurement result for adding a rule in Mitaka. Other requests (deleting, attaching, and detaching) also showed identical results.

According to our source code analysis of OpenStack Mitaka, this phenomenon is due to poorly optimized code. When a new rule is added to an existing security group, the Neutron-OpenVSwitch-agent caches all rules in the group in a list. Later, when there is a request, the cached list is unnecessarily retrieved, and this increases execution duration.

Though this is obviously a bug that should be fixed, it is a very useful feature for attackers, and it illustrates that unless cloud software systems are specifically built to resist observable timing effects, they can be powerful tools for side channels. For instance, if the `polling_interval` is set to be very long, it becomes hard to detect some requests impacting on elapsed time `after` the iptables update (as we saw from the case of the second iteration in Figure 3). In this case, if the attacker increases the overall elapsed time to as much as `polling_interval`, there would be no sleep time for any iteration and the signal from those requests becomes clearer. In addition, the attacker may exploit this "feature" to intentionally make the processing time for other requests slow. The attacker does not even need to repeatedly make requests to conduct these attacks—all the attacker needs to do is to attach a security group with a large number of rules and add one more rule to it.

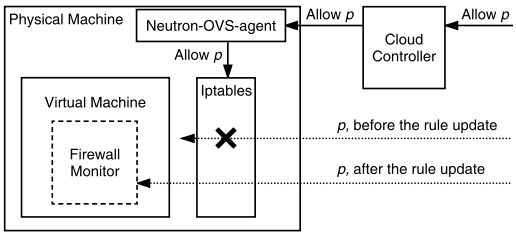


Figure 9: Monitoring update times of a virtual firewall

4 MONITORING EPOCHS

In this section, we present our techniques to measure the durations of Epochs. Since the durations of Epochs inherently tell the execution times of the shared agent, it can be used as a side channel for the infrastructure-level events in the host.

Figure 9 illustrates a basic architecture for monitoring the update times of virtual firewalls, with which the durations of Epochs can be trivially calculated. When a cloud controller receives a request to add a firewall rule allowing a probing packet p, it forwards the request to the local agent. The agent lets the iptables add the rule correspondingly, and the rule is finally added to the iptables (at time t). Meanwhile, a series of probing packets p is being sent to the monitoring VM and, naturally, only the probing packets arriving at the iptables after t can be successfully sent to the VM. Therefore, the VM can estimate the update time of the iptables from the arrival times of probe packets. Of course, there are still several questions to be answered for this mechanism to work:

- When and how frequently should the requests and probe packets be sent?
- Who sends the probe packets and who does the rule-updating requests?
- What kind of packets/rules can be used for probing?

Since the answers vary depending on the environment and the goal, we discuss different design options in the following subsections.

4.1 Update+Probe Technique

As we saw in Section 3, if a VM can know the time duration of an Epoch, it can exploit this information to guess the cloud management-level events that happened during the Epoch or to send/receive a signal to co-resident VMs by intentionally making a request to influence the Epoch. A precise measurement of firewall update times is the key for this technique. In Section 3, we could directly measure the update time, since we were operating at the level of the cloud provider. Yet from a cloud user's perspective (within an unprivileged VM), we can only "guess" the update time—*"somewhere after a firewall update request left my node."*

The open-ended probabilistic range of the update time can be narrowed down if the VM can (1) generate a set of packets that exclusively match the updated rule and (2) observe the exact time when these packets start to be successfully passed through the firewall. For example, if a VM had no rule to allow any ICMP packet in, and if it received an ICMP packet (at t_2) after it made a request to add an ingress rule for ICMP packets (at t_1), it can know there

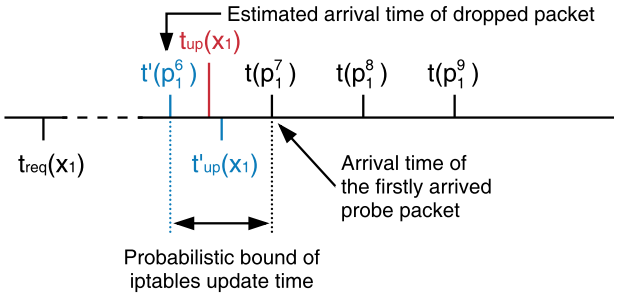


Figure 10: Time line of an UPDATE+PROBE step. $t(p_i^j)$ refers to the arrival time of j-th probe packet for the rule x_i. $t_{req}(x_i)$ refers to the time when the VM sent the request for the rule x_i. $t_{up}(x_i)$ refers to the update time of the iptables for the requested rule x_i. The prime sign means it is an estimated value.

must have been an firewall update event between t_1 and t_2. Let a probe packet p_i be a packet that exclusively matches a rule x_i. Then, the closer to the actual update time the p_i's firewall passing time is, the tighter the upper bound of the update time estimation the VM can get.

In practical terms, we can obtain such a tight upper bound by continuously sending a series of probe packets ($p_i^1 \ldots p_i^n$) and picking the one that arrives first (say p_i^j). Furthermore, we can also probabilistically tighten the lower bound of the update time by estimating the arrival (and drop) time of the previous packet p_i^{j-1}. Figure 10 summarizes the time line of this updating and probing step, which we refer to as an UPDATE+PROBE step.

Recall that our goal is not just to know the update time of a firewall rule but to perform continuous monitoring of Epochs. Therefore, we should continuously repeat UPDATE+PROBE steps at least once per iteration of the virtual networking service's RPC loop. The simplest way to do this is repeating UPDATE+PROBE every n seconds, where n is smaller or equal to the minimum polling interval of the RPC loop, as described in Algorithm 1. Here, *GetNextRule*() is a function that returns a new rule that is disjoint from any existing rules; *SendRequest*() is a function that sends a firewall rule update request to the cloud controller; *MonitorFirstProbes*() is a function that monitors the arrival times of the target probe packets, returning the first one that arrives (ignoring any subsequent ones, which give us no more timing information); and ϵ is a parameter to adjust the iteration period of REQUEST(). ($\epsilon \simeq polling\_interval - n$ if the time spent for an iteration of REQUEST() is negligible.)

Though it is harmless to perform UPDATE+PROBE more than once per RPC loop iteration, doing so will increase the number of API calls to the cloud controller and may increase the chance of being detected or rate-limited by cloud administrators. Thus, an ideal monitoring mechanism should perform an UPDATE+PROBE step once per iteration of the RPC loop.

We can design an algorithm that performs UPDATE+PROBE exactly once per RPC loop iteration by letting the next UPDATE+PROBE

Algorithm 1 Iterative UPDATE+PROBE method

```
1: procedure REQUEST(t)
2:     while True do
3:         rule ← GetNextRule()
4:         SendRequest(rule)                    ▷ non-blocking
5:         SendProbes(rule)                     ▷ non-blocking
6:         sleep(t)
7: Thread(REQUEST, [polling_interval − ϵ]).start()
8: while True do
9:     p ← MonitorFirstProbe()
10:    ReportEvent(p)
```

Algorithm 2 Reactive UPDATE+PROBE method

```
1: do
2:     rule ← GetNextRule()
3:     SendRequest(rule)                        ▷ non-blocking
4:     SendProbes(rule)                         ▷ non-blocking
5:     p ← MonitorFirstProbe(rule)
6:     ReportEvent(p)
7: while True
```

Algorithm 3 n-Reactive UPDATE+PROBE method

```
1: procedure REQUEST(t, n)
2:     for i ∈ {1, . . . , n} do
3:         rule ← GetNextRule()
4:         SendRequest(rule)                    ▷ non-blocking
5:         SendProbes(rule)                     ▷ non-blocking
6:         sleep(t)
7: rule ← GetNextRule()
8: SendRequest(rule)                            ▷ non-blocking
9: SendProbes(rule)                             ▷ non-blocking
10: while True do
11:    p ← MonitorFirstProbe()
12:    ReportEvent(p)
13:    if IsNewEvent(p) then
14:        Thread(REQUEST, [polling_interval/n, n]).start()
```

start as soon as the updating event of the previous request is reported, as described in Algorithm 2. We call this algorithm the *Reactive UPDATE+PROBE method*.

The reactive UPDATE+PROBE method is based on the assumption that the request of the next UPDATE+PROBE step arrives at the RPC front end before the next iteration of the RPC loop begins. According to the measurement result shown in Section 3, the gap between the iptables updating time of the current RPC loop iteration and the start of the next RPC loop iteration was long enough (> 800 ms with the default polling interval setup) for a round-trip of an update request. However, if the cloud controller is under a heavy load, the round-trip time may increase, and the method may miss one or more iptables update events and report a large single Epoch (that actually consists of multiple Epochs). As a remedy to this problem, we can modify the reactive UPDATE+PROBE algorithm to perform the UPDATE+PROBE steps n times, with a gap of $polling\_interval/n$ after the identification of iptables update, as shown in Algorithm 3. Figure 11 illustrates a time line of the method.

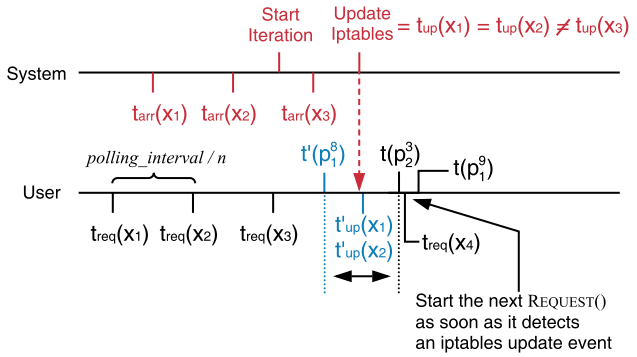


Figure 11: n-Reactive UPDATE+PROBE method. $t_{arr}(x_i)$ refers to the time at which the request for the rule x_i arrived at the edge.

4.2 Deployment

In Section 4.1, we have assumed that the VM can send packets through the firewall and observe if they are passed through. The simplest way to do this is letting another VM (say a helper node) send or receive the probe packets, though this can make precise timing difficult. However, a helper node is not always necessary: in many cases, the VM can send probe packets that pass through the firewall and come back to itself. We call packets with this property *boomerang packets*: we define a boomerang packet for a firewall rule x_i as an uniquely identifiable packet that (1) is sent by the source node and (2) is delivered back to the source node (3) without bypassing the firewall rule x_i. In reality, there are various mechanisms to generate boomerang packets that work in different network environments. For instance, if a virtual gateway router of a VM is allowed to forward packets through the interface that the packets came from, the VM can manipulate the layer-3 address of packets to generate boomerang packets without additional virtual interfaces or helper nodes. Consider an ICMP boomerang packet as an example. If we make a layer-3 boomerang packet with an ICMP echo request header as follows:

```
<Egress Probe Packet>
srcMAC:A_MAC dstMAC:GW_MAC
srcIP:A_IP dstIP:A_IP proto:ICMP
type:8 code:0 id:123 seq:355
```

(where GW_MAC refers to the MAC address of the gateway), then the gateway will forward this packet back to the source node after it changes MAC addresses as follows:

```
<Ingress Probe Packet>
srcMAC:GW_MAC dstMAC:A_MAC
srcIP:A_IP dstIP:A_IP proto:ICMP
type:8 code:0 id:123 seq:355
```

This ingress probe packet does not bypass the ingress firewall, because this does not match what the Connection Tracking System (conntrack) [3] expects: the conntrack waits for the corresponding ICMP echo reply. Therefore, as long as we do not send an ICMP echo reply, the boomerang probe packet goes through the ingress-firewall and is processed by the explicit rules; it may or may not be allowed through, depending on the rules that have been configured in the security group. This feature allows one to utilize both the ingress and egress rules for probing the iptables update time. In the rest of this paper, we explain our work based on the single-interface scenario with layer-3 ICMP boomerang packets. Further details about boomerang packets are found in our technical report [4].

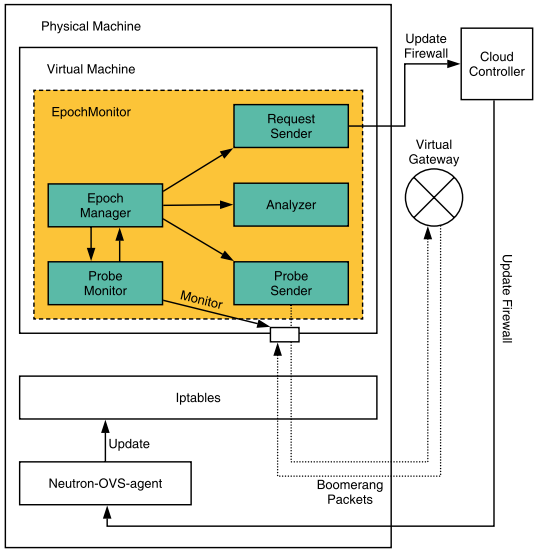


Figure 12: Architecture of *EpochMonitor*

4.3 Practical Epoch Monitor

Figure 12 shows the architecture of our Epoch monitoring system, called *EpochMonitor*. The *EpochMonitor* is designed as a stand-alone system that can be applied to various environments including restricted environments (*e.g.*, monitoring with a single VM with a single virtual interface). For a stand-alone system to monitor its own Epochs, two conditions must be satisfied: (1) the monitoring VM should be able to configure its own virtual firewalls, typically by making API calls to the cloud infrastructure, and (2) the monitoring VM should be able to generate probe packets that go through the virtual firewall and come back to itself. In this paper, we will simply assume the two conditions are met.[3]

The *EpochMonitor* consists of five components: Epoch manager, request sender, probe sender, probe monitor, and analyzer. The Epoch manager orchestrates other modules to conduct the given Epoch monitoring such as the iterative UPDATE+PROBE and the reactive UPDATE+PROBE algorithms. The request sender sends firewall rule-updating requests through the cloud API. The probe sender starts to generate a series of corresponding boomerang probes, which can be monitored at the virtual interface of the VM. The probe monitor keeps checking this virtual interface and reports back to the Epoch manager once it has found a change of Epoch. Once an Epoch change is detected, the Epoch manager immediately starts the next Epoch probing by repeating the previous steps,

[3]For further discussion of the conditions and deployment environment, please see our technical report [4].

as we described in Algorithm 2. The analyzer receives collected information from the Epoch manager, estimates every Epoch transition time, and exports the processed information. Note that, in the iterative Update+Probe method, the request sender and the probe sender do not need to be orchestrated by the Epoch manager once they have started. This means we can place these components outside of the system, in case the conditions for the stand-alone monitoring are not met.

5 EXPLOITATION

To show the practicality of the CNS-based information leakage attack, we implemented prototypes of two different covert channels between isolated VMs: i.e., VMs that through cloud mechanisms are supposed to be isolated so that communication is impossible. The first is a classic covert channel between two VMs residing in the same physical machine (§5.1). The second is a broadcast-style covert channel, where the sender VM sends messages to multiple receiver VMs scattered across a data center (§5.2). In addition, we implemented an *infrastructure event snooper* that leverages Epochs as a side channel for detecting VM creation and termination events (§5.3).

Especially in a cloud environment, since many different role players are involved, there can be various covert-channel attack scenarios. One of the probable scenarios would be *a covert channel attack from an appliance seller*. In cloud appliance markets [2, 16], cloud users can purchase VM images from sellers. Once an appliance user purchases an image, the cloud user can configure and create an actual VM instance based on the image in his or her own virtual data center. Since appliance sellers also want to protect their software in the VMs, it is commonplace for sellers to prevent direct access to the VMs through techniques such as allowing only SSH. Under this environment, we can imagine a possible covert-channel attack scenario: the image seller may try to obtain some sensitive information in the user's VM, based on the seller's image, through a covert-channel attack. Since the information is leaked through a covert channel, the user cannot prevent this attack even if he or she isolates the VM from the public network. In addition, since the user cannot directly access the VM, it is not easy for the user to determine if the VM leaks some sensitive information or not.

5.1 Single-node Covert Channel

Threat Model: We assume that the sender and the receiver VMs are co-resident in a physical machine, and the sender VM is completely isolated from the external network so that it cannot directly leak any confidential information through the network. We also assume that the sender VM has access to the cloud API system and that it has control over its own firewall rules through the API. Under this setup, the sender and the receiver VMs can make use of a covert channel based on the firewall to send a confidential message to the receiver VM. Figure 13(a) illustrates the environmental setup of this threat model.

Covert Channel Mechanism: We exploit the property of the shared execution that *any operation for one VM can influence the execution time of operations for another VM* for this covert channel. To be more specific, the sender VM controls the durations of Epochs by adjusting the amount of the load of the shared execution. The message is encoded as a stream of bits. When the sender VM wants to send a '1' for the next Epoch, it makes firewall-related requests that will be reflected during the next iptables update event. To send a '0,' it simply does nothing. Meanwhile, the receiver VM keeps monitoring the durations of Epochs using any Update+Probe method. In this way, the receiver can read a series of durations of Epochs, which are either short for '0' or long for '1.'

Note that the sender VM also needs to monitor the durations of Epochs to time its sending of the firewall change messages. Since the sender needs to take action reactively to the iptables update event, the reactive Update+Probe method is preferred to the iterative Update+Probe for the sender VM.

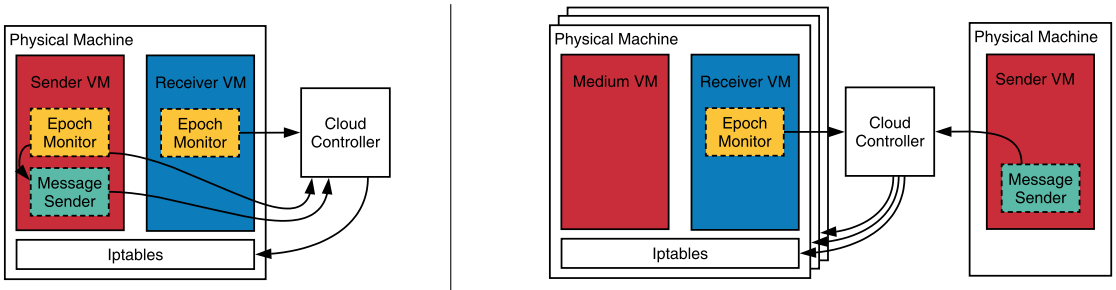
Implementation: We have implemented a prototype of the single-node covert channel for OpenStack Mitaka in Python. Since both the sender and receiver VMs need to monitor the durations of Epochs, both VMs used the *EpochMonitor* with add/delete-a-rule requests. The sender additionally uses attach/detach-a-group requests to send information on the channel. As we saw in Section 3, there can be a *task-postponing* problem if we use add/delete requests together with attach/detach requests. To address this, we have the sender use two Epochs to send each bit: the sender sends the actual bit in the first Epoch and is always idle in the second. Thus, the receiver can get the bit either from the first Epoch or, if the task was postponed, from the second. For the probes, we utilized ICMP echo request boomerang packets with ingress rules.

When the receiver notices a change of Epoch, it decodes the duration of the previous Epoch as one bit of the message. The sender reacts to a change of Epoch by sending either an overhead-introducing request or not depending on the next message bit. This is implemented in the message sender module, which receives push notifications from the *EpochMonitor*.

5.2 Multi-node Covert Channel

Threat Model: In contrast to the single-node scenario, we do not assume the sender and the receiver necessarily co-reside for the multi-node covert channel. We also do not assume that the sender VM has control over its own firewall rules. Instead, we assume that the sender VM has control over firewall rules for some other VMs scattered across the data center. Of course, the sender VM is also assumed to be isolated from the external network so that it cannot send confidential messages directly to the receivers through the network. In addition, we assume that the receiver VMs are co-resident with one or more of the VMs whose firewalls are controlled by the sender VM. Figure 13(b) illustrates this environment.

This scenario is suitable for a case where it is difficult or even impossible for the receiver VM to co-reside with the sender: for instance, a case where the sender VM is deployed as a "dedicated instance." As more cloud users are concerned with co-residency attacks, cloud providers have started to support dedicated-instance options, where an entire physical machine is dedicated to a specific customer. Since the cost of dedicated instances is greater than that of normal VMs, it is reasonable for cloud users to deploy only the VMs that handle sensitive information as dedicated instances. Under this restricted environment, the classic hardware-based covert channels do not work. We will show that the sender can leak confidential



(a) Single-node Covert Channel (b) Multi-node Covert Channel

Figure 13: Covert channel setups. The color of each VM represents the tenant it belongs to. The medium VM in (b) is a VM whose firewall is controlled by the sender VM.

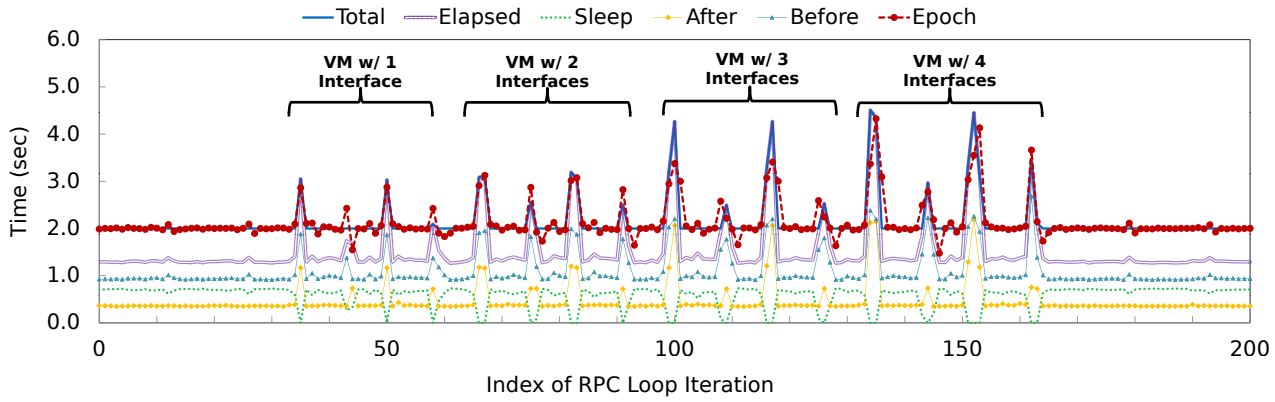


Figure 14: Signals of creating/terminating VMs in Epochs. We created and terminated eight VMs one-by-one and monitored their impact on Epochs by running the EpochMonitor in a co-resident VM. The number of interfaces for each VM is indicated.

messages through the firewalls of other VMs that it controls as a covert channel under this environment.

Covert Channel Mechanism: Under the given environment, we should first consider the limitation that the sender VM is unable to know whether messages are sent successfully. Since the firewall update requests do not make any changes on the sender's own VM's iptables, the sender is unable to "read" the signal on the channel. Moreover, even if the effects of the requests were visible to the sender, it is not meaningful for the sender VM to monitor its own iptables update events because the updates are not synchronized across hosts, meaning that the Epoch start times will be different on the sender and receiver(s). Therefore, it is important for this covert channel to send messages in a noise-tolerant way. In our proof of concept, we simply assume the message can be safely sent if each bit of the message is sent for n seconds repeatedly. A future implementation could use forward error correction for greater efficiency.

To send the same bit repeatedly for n seconds, we could reuse the single-node covert channel mechanism. However, this would require the sender to invoke many API calls when it sends a bit '1': i.e., the sender would need to make an overhead-introducing request every second for n seconds. Instead, we use a different way to influence the shared execution. In Section 3.2, we saw some requests that can permanently increase or decrease the execution time of the iptables update process. Therefore, the sender can exploit these special requests to send either '1' (by "permanently" increasing the execution time) or '0' (by "permanently" decreasing the execution time) for as long as it needs. API requests are now only necessary on the "edge" between transmitting a '0' and a '1' or vice versa. The receiver uses the same monitoring mechanism as the previous covert channel.

Implementation: As a proof of concept, we implemented this multi-node covert channel for OpenStack Mitaka in Python. The sender attaches or detaches a security group with a large number of rules (e.g., 2,000) as its method of "permanently" increasing and decreasing the execution time of iptables updates. The architecture of the receiver is similar to the previous covert channel's implementation except that it uses the iterative Update+Probe mechanism and utilizes both the creation and deletion of rules.

5.3 Snooping on Infrastructure Events

Epoch as a Side Channel: Since the RPC loop processes most network-related requests, infrastructure-level network changes could leave their marks on Epoch times. This means that, if a VM keeps monitoring such infrastructure-level activities through the Epochs, and if it can distinguish different events from the Epochs, it

would be possible to extract further infrastructure-level information such as *"the number of VMs the host is running"* and *"the sizes of security groups attached to this host."* This would represent a meaningful new class of side-channel attacks targeting infrastructure-level information, which can provide valuable information about the cloud provider itself, and can potentially be used to improve other attack vectors.

Based on this idea, we implemented a prototype of an *infrastructure event snooper* using the *EpochMonitor* in OpenStack. In particular, we found that VM creation and termination events make relatively strong signals on a sequence of Epochs as shown in Figure 14. This is because VM creation and termination involve virtual network-level changes in multiple network components. For the same reason, we can also see that we may estimate *"the number of virtual interfaces of a created/terminated VM"* through this channel. Since cloud providers typically limit the number of virtual interfaces for a VM by its flavor (i.e., the larger the VM is, the more virtual interfaces it may have), this information could be used as a good indicator to estimate the size of the VM as well.

Detecting Events: Since the same type of event causes a similar pattern of sequences of Epochs, detecting an event from a sequence of Epochs can be understood as a sequence classification problem. We used a Long Short-Term Memory (LSTM) [12] model; LSTM is a deep-learning approach well known for its good performance in sequence classification problems [11, 13, 19]. To apply the LSTM model, we used each subsequence of Epochs with a fixed window size as a data point, and the label of each event was given to the data point that covers all the Epochs influenced by the event and has the longest Epoch at the center. For the current prototype, we used a neural network consisting of four LSTM layers, and covered nine different classes: *Idle*, *Creating a VM with* $\boldsymbol{n}$ *interfaces*, and *Terminating a VM with* $\boldsymbol{n}$ *interfaces* (where $\boldsymbol{n} \in \{1, 2, 3, 4\}$). Further details about the experimental setup and the evaluation results are presented in Section 6.4.

6 EVALUATION

In this section, we demonstrate the feasibility of our CNS-based information leakage channel by presenting practical evaluation results for both types of covert channels, as well as the infrastructure event snooper. For this evaluation, we deployed an OpenStack Mitaka IaaS cloud in the Utah Emulab network testbed [23].

6.1 Accuracy of *EpochMonitor*

We first evaluated the accuracy of *EpochMonitor*'s estimation for the size of Epochs. We ran two EpochMonitors in different VMs concurrently and had both measure the sizes of Epochs while we were generating arbitrary requests. We evaluated the accuracy by comparing their results to the ground truth, which is directly collected from the host machine's firewall agent. The evaluation result showed that the estimations of both VMs were very precise. Across 460 Epochs measured, the first VM showed a root mean squared error (RSME) of 684 microseconds, and the second showed an RMSE of 649 microseconds. The maximum errors of the two VMs were 1.54 milliseconds and 25.5 milliseconds, which is sufficient for distinguishing different requests, which typically show more than 100 milliseconds of difference as described in Section 3.

6.2 Single-node Covert Channel

For this evaluation, we created two co-resident VMs: a sender and a receiver belonging to different tenants. We had both the sender and the receiver run *EpochMonitor* with the reactive Update+Probe method. Both VMs used add/delete requests to monitor Epoch lengths. The sender used attach/detach requests to send messages by manipulating Epoch durations. As discussed in Section 5.1, to properly handle the *task-postponing* problem, we used two Epochs to send each bit. The sender sent the message "`hello world`" encoded in ASCII. Figure 15(a) shows the "`hello`" part of the result.

As one can see from the figure, the message was sent without much noise. With a naive decoding method that simply interprets any Epoch taking longer than 2.2 seconds as a bit '1,' the receiver had a 0% error rate. For a real-world deployment, more robust error-handling strategies would be required since the environment could be noisier. This could easily be accomplished through more distinguishable patterns as we saw in Section 3 or through forward error correction.

For the "`hello world`" message, the actual bandwidth of the covert channel was 0.211 bits/second. Note that, since we sent '1' bits using two Epochs, the best bandwidth for this covert channel is 0.25 b/s (in case the sender sends only '0's). Though the bandwidth can be further improved by utilizing more patterns and encoding multiple bits in an Epoch, we believe the current bandwidth would be enough for some use cases such as co-residency detection or leaking cryptographic keys.

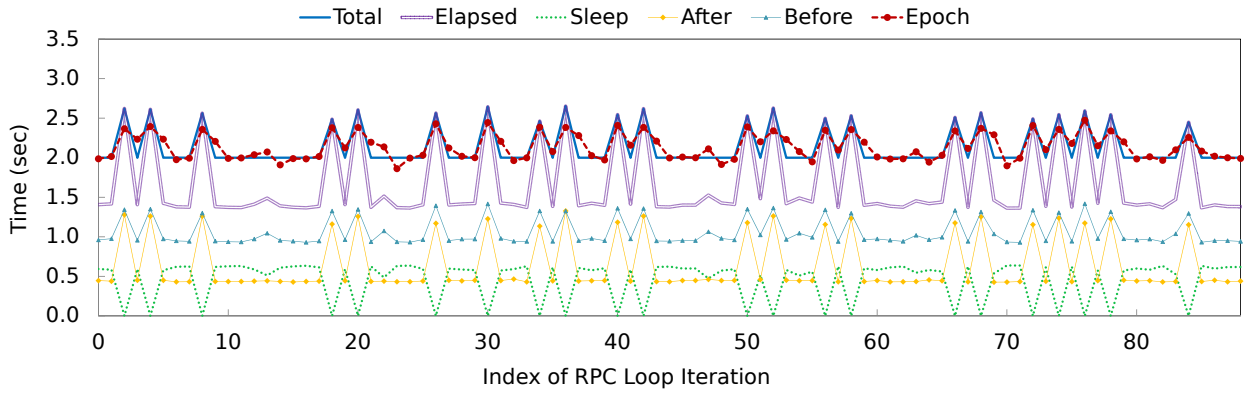
6.3 Multi-node Covert Channel

For the multi-node covert channel attack, we used three VMs. In this case, the sender and receiver are on different hosts. The third VM, used as an intermediary, is co-resident with the receiver. As introduced in Section 5.2, the sender and intermediary VMs belong to the same tenant, and the sender could update the firewall of the intermediary. The receiver belongs to a different tenant. The receiver ran *EpochMonitor* with the reactive Update+Probe method and issued add/delete requests, and the sender sent the string "`hello world`" through the intermediary using long-term impact requests, attachment and detachment of a security group with 2,500 rules. The sender used 10 seconds to send each bit. Figure 15(b) shows the "`hello`" part of the message as observed by the receiver.

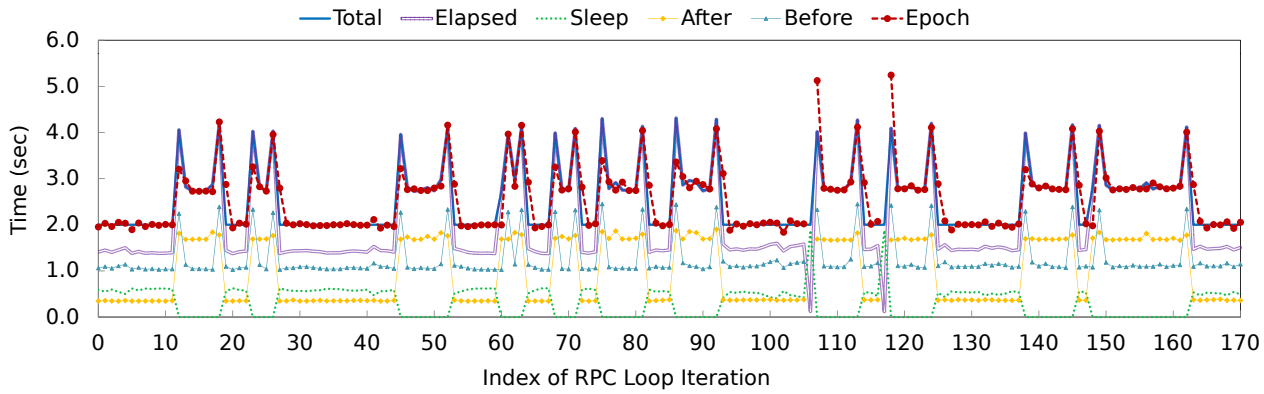
One can see from the results that the receiver saw the message very clearly. The bandwidth was only 0.1 b/s since each bit was sent for 10 seconds. There were several iterations (for example, the 107th and 118th) in which no firewall-updating requests arrived. This is because the receiver used the reactive Update+Probe method. This could be "fixed" by deploying the iterative or n-reactive Update+Probe methods.

6.4 Infrastructure Event Snooper

We evaluated the performance of the infrastructure event snooper for two different types of physical machines in the Utah Emulab network testbed, which we will refer to as type-1 [10] and type-2 [9] hosts. For each type of host, we first collected 100 data points for each class of event as training data: call this data `TR1` for type-1 hosts and `TR2` for type-2. We then generated three different LSTM models for evaluation: `M1`, `M2`, and `MC`, which were trained on `TR1`,



(a) Single-node covert channel



(b) Multi-node covert channel

Figure 15: Execution durations of the RPC loop while sending "hello" through covert channels

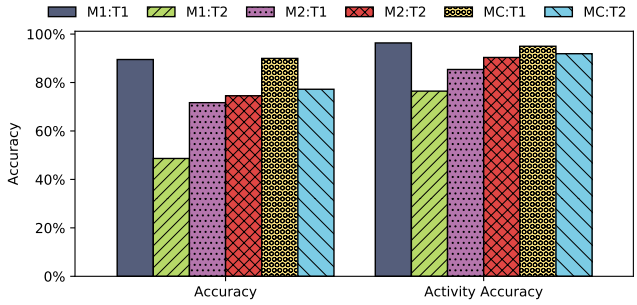


Figure 16: Accuracies of the infrastructure event snooper with different LSTM models. The accuracy of a model x against test data y is labeled x:y.

TR2, and both, respectively. For training, we used 75% of the data for actual training and the other 25% as validation data so that we could halt the training before overfitting began. Next, we collected test data for each type of host (call these data sets T1 for type-1 and T2 for type-2) by creating and terminating 100 VMs in a random order, where the VMs were configured to have a random number of virtual interfaces between one and four. To prevent bias on the evaluation results due to too many trivially identifiable "Idle" data points, we pre-filtered the test data using a threshold and obtained 219 data points for T1 and 259 for T2, which respectively contain 19 and 59 "Idle"-class data points.

Figure 16 shows the evaluation results. In addition to regular (exact-class) accuracy, the figure also presents *activity accuracy* that counts any data point whose activity is correctly classified as a true positive (i.e., ignoring the number of interfaces). As one can see from the figure, the model M1 showed a large performance gap between T1 and T2. This is because the Epoch sequence patterns in the type-2 hosts were more varied, and thus the model M1—which was trained on relatively stable data TR1—could not classify some "erratic" signal patterns in T2. However, for the model trained on both (MC), one can see good performance for both test data sets, achieving 83.1% accuracy and 93.3% activity accuracy on average.

In addition, for VM creation events, the models showed remarkably high true positive rates. For instance, as shown in Table 1, the model MC showed 100% true-positive rate for VM creation activity when ignoring the number of interfaces.

Table 1: Classification result of the model `MC` against the test data set (`T1 + T2`). The Roman numerals represent the number of virtual interfaces of the VMs. The underlined entries count *correctly classified* data points, and the bold-faced entries count data points whose *activities are correctly classified*: Idle, VM Creation, or VM Termination.

			Classified								
			Idle	VM Creation				VM Termination			
				I	II	III	IV	I	II	III	IV
Ground Truth	Idle		***72***					*6*			
	VM Creation	I		***46***							
		II			***50***	***12***					
		III				***35***	***3***				
		IV					***54***				
	VM Termination	I	*2*					***31***	***13***		
		II	*2*	*9*	*1*			***4***	***34***	***10***	***2***
		III							***1***	***33***	***4***
		IV		*1*	*11*						***42***

7 MITIGATION TECHNIQUES

Adjusting the Polling Interval: A simple way to reduce the potential for information leakage through a cloud's networking services is to increase the services' polling intervals. For example, if one sets the polling interval to 10 seconds, since it is very rare for actual network-level updates to take longer than 10 seconds (as we saw in Section 3), the loop will complete within 10 seconds most of the time, making it hard for attackers to distinguish difference in Epochs between different requests. Though this approach is readily available and may suppress the attack to some degree, there are two problems with this approach. First, this approach does not prevent the attacker from sending/receiving signals sent through the elapsed time `before` updating iptables. Second, this approach may increase either the response time of the request or the chance for virtual resources to be *functionally inconsistent* [5]: different parts of the tenant's infrastructure could see inconsistent resource states for extended periods of time.

Setting the polling interval to be very short is not an effective mitigation strategy. Though it would seem that this prevents multiple requests from being processed within the same iteration, intensive requests (such as attach/detach requests) will still produce noticeable delays.

Request Rate Limiting: Since rate-limiting is a general strategy to suppress DoS-style attacks targeting API front ends, one may consider using rate-limiting to suppress information leakage through shared services as well. However, compared to DoS-style attacks, the actual request rate needed for these channels is very low. For running *EpochMonitor* with the reactive Update+Probe method, we need just two requests per polling interval. We believe more involved analysis on the request rates of tenants is required to detect this type of attack at the service level. One of the possible approaches in this regard is to devise a policy that may effectively throttle requests from these attacks and enforce the policy by extending the existing distributed resource management systems [15, 18].

8 RELATED WORK

Ristenpart et al. [17] first introduced several information leakage channels and their potential impacts in the cloud. Specifically, the authors utilized covert channels based on hard disks and caches to verify the co-residency of two cooperative VMs, achieving a bandwidth of 0.2 b/s. This cache-based cloud covert channel has been further improved through a number of studies [14, 24, 25], and the latest result achieved a transfer rate of up to 1.2 Mb/s [14]. Bates et al. [6] exploited the physical network interface as a side channel to detect co-residency; in their scenario, the probing VM could detect co-residency with the victim by monitoring the network performance change of the victim. All these previous studies focus on hardware-level shared resources, while our focus is on infrastructure-level software services.

In addition to the aforementioned work, Ristenpart et al. [17] exploited additional side channels based on the behavior of a *cloud management system*—including host machines' IP addresses, inter-VM network round-trip times, and numerical distances of internal IP addresses—to detect co-residency as well as the VM placement policy of the cloud. Varadarajan et al. [21] and Xu et al. [26] showed that the previous approaches do not work any more in modern cloud platforms, but still there exist several factors that may increase the probability of co-residency. In the sense of analyzing and utilizing the properties of cloud management systems, these studies are the most similar to our work. However, since our side channel is based on the fundamental software architecture (i.e., shared service), it would be more difficult to suppress this type of side channel.

9 CONCLUSION AND FUTURE WORK

We have shown that management services in the cloud can be exploited to build an information leakage channel. Through our evaluation, we have demonstrated that we can conduct robust and unique information leakage attacks by exploiting this channel in OpenStack. As future work, we plan to explore the feasibility of this type of attack in different cloud platforms. We also plan to investigate the extensibility of the attacks presented in this paper to other shared services.

ACKNOWLEDGMENTS

We thank the anonymous SoCC reviewers for their valuable comments on this work. We performed our experiments in the Utah Emulab network testbed [23]. This material is based upon work supported by the National Science Foundation under Grant No. 1314945.

REFERENCES

[1] Amazon Web Services. 2018. AWS CloudTrail. Retrieved Aug. 24, 2018 from https://aws.amazon.com/cloudtrail/
[2] Amazon Web Services. 2018. AWS Marketplace. Retrieved Aug. 24, 2018 from https://aws.amazon.com/marketplace
[3] Pablo Neira Ayuso. 2006. Netfilter's connection tracking system. *;login:* 31, 3 (June 2006), 34–39. https://www.usenix.org/publications/login/june-2006-volume-31-number-3/netfilters-connection-tracking-system
[4] Hyunwook Baek, Eric Eide, Robert Ricci, and Jacobus Van der Merwe. 2018. *Monitoring the Update Time of Virtual Firewalls in the Cloud.* Technical Report

UUCS–18–005. University of Utah. http://www.cs.utah.edu/docs/techreports/2018/pdf/UUCS-18-005.pdf

[5] Hyunwook Baek, Abhinav Srivastava, and Jacobus Van der Merwe. 2017. CloudSight: A Tenant-Oriented Transparency Framework for Cross-Layer Cloud Troubleshooting. In *Proc. CCGRID*. 268–273. https://doi.org/10.1109/CCGRID.2017.97

[6] Adam Bates, Benjamin Mood, Joe Pletcher, Hannah Pruse, Masoud Valafar, and Kevin Butler. 2012. Detecting Co-Residency with Active Traffic Analysis Techniques. In *Proc. CCSW*. 1–12. https://doi.org/10.1145/2381913.2381915

[7] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. 2014. Adaptive Stream Processing using Dynamic Batch Sizing. In *Proc. SoCC*. 1–13. https://doi.org/10.1145/2670979.2670995

[8] Thomas Erl. 2005. *Service-Oriented Architecture: Concepts, Technology, and Design*. Pearson Education India.

[9] Flux Research Group. 2018. D430: The Emulab Dell R430 (aka "d430") machines. Retrieved Aug. 24, 2018 from https://wiki.emulab.net/wiki/d430

[10] Flux Research Group. 2018. D710: The "d710" Nodes. Retrieved Aug. 24, 2018 from https://wiki.emulab.net/wiki/d710

[11] Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks* 18, 5–6 (July–Aug. 2005), 602–610. https://doi.org/10.1016/j.neunet.2005.06.042

[12] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[13] Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional LSTM-CRF Models for Sequence Tagging. *CoRR* abs/1508.01991 (2015), 10. http://arxiv.org/abs/1508.01991

[14] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *Proc. IEEE S&P*. 605–622. https://doi.org/10.1109/SP.2015.43

[15] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2015. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *Proc. NSDI*. 589–603. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/mace

[16] Microsoft. 2017. Azure Marketplace. Retrieved Aug. 24, 2018 from https://azuremarketplace.microsoft.com/en-us

[17] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proc. CCS*. 199–212. https://doi.org/10.1145/1653662.1653687

[18] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. 2017. Distributed Resource Management Across Process Boundaries. In *Proc. SoCC*. 611–623. https://doi.org/10.1145/3127479.3132020

[19] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27*, Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger (Eds.). Curran Associates, Inc., 3104–3112. https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks

[20] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M. Swift. 2012. Resource-Freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense). In *Proc. CCS*. 281–292. https://doi.org/10.1145/2382196.2382228

[21] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2015. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *Proc. USENIX Security*. 913–928. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/varadarajan

[22] Zhenghong Wang and Ruby B Lee. 2006. Covert and Side Channels Due to Processor Architecture. In *Proc. ACSAC*. 473–482. https://doi.org/10.1109/ACSAC.2006.20

[23] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. OSDI*. 255–270. https://www.usenix.org/legacy/event/osdi02/tech/white.html

[24] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Proc. USENIX Security*. 159–173. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/wu

[25] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. 2011. An Exploration of L2 Cache Covert Channels in Virtualized Environments. In *Proc. CCSW*. 29–40. https://doi.org/10.1145/2046660.2046670

[26] Zhang Xu, Haining Wang, and Zhenyu Wu. 2015. A Measurement Study on Co-residence Threat Inside the Cloud. In *Proc. USENIX Security*. 929–944. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/xu

[27] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proc. USENIX Security*. 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[28] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. 2011. HomeAlone: Co-Residency Detection in the Cloud via Side-Channel Analysis. In *Proc. IEEE S&P*. 313–328. https://doi.org/10.1109/SP.2011.31

[29] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proc. CCS*. 305–316. https://doi.org/10.1145/2382196.2382230