Detecting Malicious Co-resident Virtual Machines Indulging in Load-Based Attacks

Smitha Sundareswaran and Anna C. Squciarini

College of Information Sciences and Technology
Pennsylvania State University
University Park, PA 16802

Abstract. Virtualization provides many benefits for Cloud environments, as it helps users obtain dedicated environments abstracted from the physical layer. However, it also introduces new vulnerabilities to the Cloud such as making it possible for malicious VMs to mount cross-VM attacks through cache based side channels. In this paper, we investigate load-based measurement attacks born specifically as a result of the virtualization in Cloud systems. We develop a framework to identify these attacks based on the observation that the events taking place during the attacks lead to an identifiable sequence of exceptions. We test the accuracy of our framework using the Microsoft Azure infrastructure.

1 Introduction

Cloud computing provides great benefits to the consumers through effective and efficient services in the form of infrastructure, platform, and software. These types of services are offered by all leading Cloud Computing Service Providers (CSPs), including Microsoft Azure, Amazon EC2, and Rackspace [1, 3, 15]. The advantages of such a service model are exemplified in the reduced operational costs, and efficient resource allocation and usage.

To date, security issues have remained the biggest thorns in the full blown adoption of these services [6,8,19]. Most of the current and recent work on Cloud computing security focuses on ensuring the privacy of general outsourcing techniques (e.g. [20]). Furthermore, recently there has been a interest in attacks which particularly target the weaknesses of the Cloud Computing architecture's general design [5, 10], particularly due to the use of virtualization [8, 9]. Virtualization of computing resources is a prominent feature of the Cloud providers, regardless of the type of service being offered (i.e. infrastructure, platform, and software). However, virtualization also produces unique side-channels for attacks, which cannot be controlled by usual information flow procedures. The virtual machines (VMs) may be malicious themselves [22] or the VMs' image may be compromised [23]. Precisely, recent work [2,16,25] found that Cloud systems leak information about location of the Cloud instances, letting attackers collocate an instance with another specific instance. Thus, if an attacker can cause a victim's Cloud instance to leak information covertly, and if covert channels with sufficient bandwidth exist, unauthorized leakage might be possible.

S. Qing et al. (Eds.): ICICS 2013, LNCS 8233, pp. 113–124, 2013.
© Springer International Publishing Switzerland 2013

In this paper, we focus on load-based measurement attacks, which are covert side-channel attacks born specifically as a result of the virtualization in Cloud systems [16]. In general, a covert channel attack is an attack which takes place when two entities or processes communicate with each other via channels that are hidden and therefore not subject to the general access control techniques. These channels can be formed by relying on time-based operations [2], such as opening and closing a file at a certain time, or can rely on techniques such as port knocking [24]. In the context of Clouds, these attacks are based on shared physical resources, such as the physical host's cache to create a side-channel in VMs that are otherwise segregated. Attacks based on covert-channels not only exploit co-residence, but also cover the basic requisites of identifying a particular VM, instantiating a VM co-resident with the VM of interest, and communicating data about the VM of interest. Other attacks, such as the VM conflicts arising due to competitors sharing a physical host described in [17], and the adversarial VMs presented in [25] are extensions on the same theme.

We design our attack detection framework based on empirical observations on the attacks patterns and behavior. We observe that by identifying the patterns of exceptions, and whether the number of exceptions in a given time period crosses a carefully crafted threshold, one can identify on-going attacks, and with further analysis, zoom in on the types of attacks being carried out. Our solution therefore relies on extracting and detecting event-based patterns, where the events are comprised of exceptions, and on establishing a baseline frequency for the total number of events occurring in an allotted time.

We test our system's accuracy using Windows' Azure architecture, where we co-host multiple VMs. We show our achieved accuracy even in increasing noise of busy VMs, which have a large number of active programs and tasks.

The paper is organized as follows. We discuss the related work in Section 2. Our threat model is discussed in Section 3. Section 4 describes the attacks we handle. The design of our framework is described in Section 5. Experimental results are discussed in Section 6. Finally, we conclude in Section 7.

2 Related Work

Since Ristenpart's seminal work [16], there has been a lot of interest in side-channel attacks. Accordingly, many have introduced new extensions of the original attacks and tackled some of them, or their variations [7, 11–13, 17, 25].

Cleemput [7] discusses compiler based mitigations for timing attacks. Similar to the approach discussed in this paper, the authors solution uses compiler instructions to look for attacks, in as much that some exceptions are issued by the compiler. The authors' proposed solution however does not consider VMs or co-resident systems, instead it focuses on loss of cryptographic secrets by timing attacks. A possible solution to load-measurement attacks is offered by Sun et al. [11], who consider the load sharing between co-resident virtual machines in a Cloud. They observe that two co-resident VMs may pose a "threat" to each other due to a need for common resources, which may enable each to learn

secrets of the others. Kong et al [12] suggests a combination of hardware and software approaches to provide a solution to side-channel attacks. While they do not consider VMs or Clouds in particular, they propose using a special set of Load instructions that inform the software when the load-misses in the cache. As we show in the next sections, although all similar to our work in their intention, existing approaches are vastly different from ours. We are similar to Sun's work in terms of objective, but their primary focus is on conflict resolution through negotiation, rather than on detection. Kong's proposal also shares some similarities with our work, i.e. it seeks to eliminate cache based interferences, but yet differs vastly. Our idea is to exploit existing instructions as signals of an attack as opposed to Kong's approach, that is based on forming new ones.

A variation on this theme is offered by Choi and colleagues [4]. The authors discuss an authentication scheme which may be used for Cloud Computing systems. An authentication scheme of this kind is useful in providing the attacker with an identity, but it does nothing to prevent such attacks, and is therefore complementary to the scheme proposed in our work. In addition, side-channel analysis has been often used to mitigate problems due to co-residency, such as competitive organizations' VMs being co-resident or high loads on shared physical resources [13, 17, 25]. Also related to our methodology is work on software component thefts [21] from Wang. Wang and colleagues exploit the notion of dependencies among system calls to detect various attacks. We do not analyze system calls' dependencies in our approach, so as to focus on the events which lead to the system calls.

3 Threat Model

Our architecture is built to identify the attacks launched between tenants of co-existing VMs. We assume that all the VMs are compatible on a physical level, and they may run any applications intended by the tenant. The provider and its infrastructure are assumed to be trusted. That is, we do not consider attacks that rely on subversion of the Cloud's administrative functions. This in turn means that we trust any software that is run on the physical machines or the VMs that the Cloud hosts. Our threat model considers non-insider adversaries, who manage to get a VM hosted on the same physical machine as their victim's by chance or intentionally [16, 17]. We assume that a malicious party can run and control many instances in the Cloud, simply by leasing the required storage space. The targeted victims are tenants who run confidential services in the Cloud. Any data leakage, including data about the usage of the VMs, can breach the confidentiality of the victim. From the victims' point of view, the co-existing VMs on the physical machine could be benign, or malicious by attempting to find information about other co-existing VMs through the cache-based side-channel. Although a tenant can trust that his VM is not willfully malicious, the attacker can manipulate all shared physical resources at his own gain. Shared resources include CPU caches, branch target buffers, and network queues. By properly controlling and observing information gathered from these resources,

information may be leaked unwittingly to the attacker. In particular, we focus on load-based co-residence detection as this type of attack is a common and well-know example of network probing attacks.

Notice that our model is a generalization of the threats discussed in the seminal works [16, 17], which discussed the load-variation attacks tackled in this work. In our threat model, however, we do not require an existing VM to have a conflict with a newly migrated VM, for the existing VM to be malicious (as in [17]). Further, different from [16], we do not differentiate between attackers who are interested in simply attacking any known hosted service, and attackers interested in attacking a particular victim service. Due to these differences, we no longer can depend on shared services to point out possible conflicts.

4 Covert Attacks

4.1 Attack Description

Load-based attacks require two steps: placement and extraction. Placement refers to the attacker placing their malicious VMs on the same physical machine as that of a victim. Extraction refers to extracting confidential information via a cross-VM attack using side-channels. Cross-VM information leakage is due to the sharing of physical resources (e.g., the CPU's data caches). In this work, we focus on extraction, assuming placement is given. A malicious VM can detect co-residence in many ways [16]. When the attacker has some knowledge of computational load variation on the target instance, no network-based detection techniques are needed. The attacker can actively cause load variation due to a publicly-accessible service running on the target (for example, HTTP, POP3 or FTP services). Publicly-accessible services are not suspect for an intrusion detection system as they normally are not access restricted. Hence, any accesses or measurements on these public services often remain unnoticed. In our work, we consider the existence of such publicly-accessible services as the primary condition for an attack. The attacker may also be able to detect co-residence without resorting to actively creating any load variations if he has a priori information about some relatively unique load variation on the target. For example, knowing that a certain website experiences heavy traffic from 9 am to 5 pm, and in the remaining time, no traffic or negligible traffic is experienced on a daily basis can provide useful a priori information for the attacker. In this case, based on the time of the day, an attacker can detect the co-residence of a VM by identifying the physical hosts which experience a similar load variation. The difficulty (or ease) of detection would be based on the comprehensiveness of the apriori information.

One of the best known ways for accurate measurement of cache usage is based on three main functions: *prime, probe and trigger* [16]. *Priming* consists of reading a contiguous buffer `B` of size `b`. The buffer `B` is located on the CPU cache of the physical host. `b` is large enough that a significant portion of one of the lower level caches (L1, L2 or L3) on the physical host is filled by the contents of buffer `B`. The buffer `B` is read in s-byte offsets where s is the size of the cache.

The next step of load measurement is *trigger*, which is busy looping. Busy looping is a technique in which a process repeatedly checks if a condition is true. The attacking VM busy loops until the CPU counter cycle jumps by a large value, so that is allows enough time for the processes of the other VMs to run. Thus, the cache is filled by data accessed by the victims. The final step is *probing*, which consists of reading `B` again at s-byte offsets. When carrying out the probe by reading b/s memory locations of `B`, the attacker uses a pseudorandom order, and the pointer-chasing technique described in [18], as using the pseudorandom order prevents the CPU's hardware prefetcher from hiding the access latency. The time of the final step's read gives the load sample, where the load sample is measured in number of CPU cycles.

Identifying whether or not a VM is co-located prepares the ground for the attacker to mount more intrusive attacks, such as colluding with a rogue program on the co-resident VM even if other channels of communications are stopped. Further, knowing that a specific VM is co-resident allows the attacker to find some meta data about the owner of the VM: an attacker can correlate the service hosted on the VM with the name of a company running the service. This not only tells the attacker the victim's preferred Cloud provider, it also allows her to identify the regions in which the victim is running certain services.

4.2 Preliminary Evaluation

In order to gauge a better understanding of the attacks' dynamics, we recreated the scenario required for co-residence detection. The testbed consisted of 5 Windows Server 2008R2 SP1 based VMs [1], where each of the VMs had their TCP ports 80 enabled to allow HTTP services. Out of these 5 VMs, 3 of them were simply bystanders used to create noise. One of the VMs functioned as the attacker VM, while another was the victim VM. The victim additionally hosted an Apache Tomcat application server, with a single webpage. Because load-based detection can occur in many ways, we executed the attacks under various settings. First, to detect whether two VMs were co-resident, we created a high load on one of the VMs (the target VM) using the LoadUI tool [14]. LoadUI is a utility commonly used for load testing, and is being used to to simulate the attacker VM in a systematic and rigorous fashion. The attacker VM collected 100 load samples on a public HTML page of size 10KB both during the load variation and when the load variation was not done. It tracked the variation in size of the load samples: if the target and the attacker VM were co-resident the load samples taken during the load induced by the LoadUI tool were larger.

To communicate with a co-resident VM (assuming all other communication channels are closed), we rely on the simple cache-covert channel attack wherein the attacker sending the message idles while transmitting "0", but frantically access the memory to transmit "1". The receiver checks the latencies by collecting a load sample or accessing a memory block of his own. While communicating

[1] We ensured that the VMs were co-resident by checking their PhysicalHostName through their registry keys.

with a rogue program can be done simply by using load variations, where a high load indicates a "1" and a low load indicates a "0", a noisy channel can reduce the efficiency of this method of transmission. To overcome the effects of noise, the rogue program needs to be able to cause a sufficiently high load to be "heard" over the channel by the colluding attacker. Empirically, we can see that a load spike above 100000 CPU cycles is necessary for a clear co-residence detection. Further, a process generally crashes when the process has a utilization of 67%-70%. Our systems crashed upon reaching 8100000 CPU cycles, when channel noise was created by loading 3 other VMs. The noise was measured by taking load samples from the cache at any one of these VMs. So long as the load spike of over 100000 CPU cycles can be achieved without crashing the loaded process, the simple method of load variations to transmit a message works well.

In case of a high probability of crash due to noisy channels, the use of the prime+probe+trigger method is preferable. To test the detection of co-residence on a noisy channel, we simulated a channel with high noise by having 5 co-resident VMs, out of which one was the attacker VM, and one was the victim, while the others were just meant to create noise. We had the co-resident VMs performing I/O operations, while the attacker VM measured took 100 load samples on a public HTML pages varying in size from 1Kb to 10 KB over a period of time varying from 12 seconds (for the 1KB page) to 120 seconds (for the 10KB pages). The measurements were then paused for a period of 30 sec. after which they were repeated while simultaneously making numerous HTTP get requests from a third system to the target. The attack was successful in that the malicious VM was able to detect the co-residence when the HTTP requests were made.

These initial tests provided us with insights on the scope and effectiveness of each of these attacks. We observed that there is always a pre-set sequence of events that yields to an attack. All the events are observable from the physical host in the form of exceptions.

5 Design of Co-residency Attack Detector

Our initial experiments confirmed that each attack instance incur into a notable load increase accompanied by a given pattern of system calls and exceptions. Accordingly, our solution consists of few main steps: (1) collection of system calls occurring at the physical host, and the exceptions which may be specific to the attacks, to (a) identify the VM causing the exception, and the process that spawned the exception, and (b) identify whether conditions sufficient for the attack exist, and (2) processing of these exceptions to detect the load-variation attacks discussed in Section 4. Each of the steps is associated with a logic module, which we refer to as **Observer** and **Defender**, in what follows. The **Observer** and **Defender** are implemented as part of a trusted VM.

5.1 Observer

The Observer component is designed to dynamically collect metrics indicative of suspicious load variations. We specifically focus on tracking network processes,

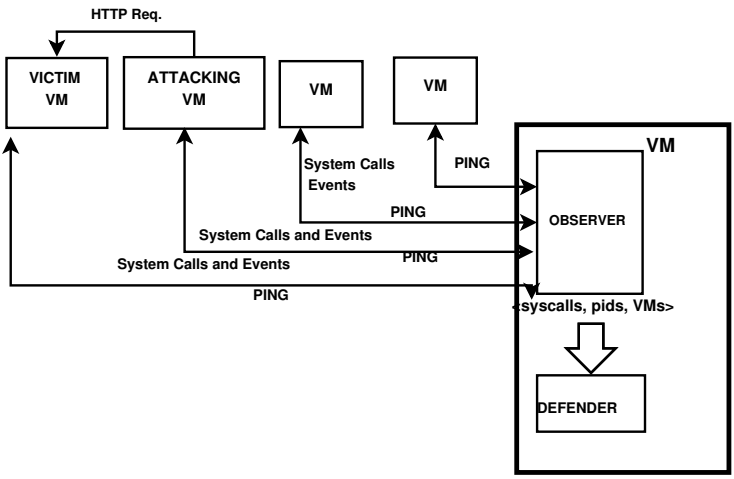

Fig. 1. System Design

their CPU loads and spawned system calls. Hence, the Observer has three main tasks: (1) extract systems calls and interrupts of monitored processes, (2) map the identified system calls and interrupts to specific programs from the specific VMs which generated them, and (3) determine whether the conditions necessary for an attack to be carried out co-exist. In order to extract and map the system calls, the Observer spawns multiple tracing threads, instantiated by means of debugging tools, such as Linux strace and WinGB in Windows.

To quantify the necessary conditions for an attack, the Observer uses some baseline system metrics on CPU utilization by processes observed in absence of attacks, as well as the expected number of system calls experienced by any given process. Specifically, a training phase is performed first, during which processes are monitored under minimum activity for over 72 hours (or more), such that each process activity remains lower than 1%. We observe the number of system calls per second for each process, denoted as $SysC_t^{PID}$, with PID denoting a specific process. Obviously, the base number of system calls per second is unique to each process. Nevertheless, we observed the following pattern. Given a process PID, let $SysC_t$ be the base rate per second, and let CPU_UT its percentage of CPU utilization.

$$AVG_t^{PID} = \begin{cases} SysC_t + 0.02 * SysC_t & \text{if } CPU\_UT < 20\% \\ SysC_t + (SysC_t * 2^{(CPU\_UT-0.2)/5}) \frac{CPU\_UT}{5} & \text{if CPU\_UT}>20\% \end{cases} \quad (1)$$

The above equation indicates that a general 2% rate increase exists when the process activity increases up to 20%. Above 20% CPU utilization, the rate starts exponentially increasing, for each 5% increase in process activity. When the number of system calls increases exponentially, the monitoring becomes exponentially difficult too. Based on the above, we determine CPU utilization threshold, denoted as τ, which ranges from 10% to 20% for a given process. τ denotes the

point to start the monitoring and is chosen so that the attack cannot be hidden in the explosion of system calls. In addition, the system calls generated by each process are then averaged over a time interval T according to Equation 1, to AVG_T^{PID}. The gathered data is stored with the trusted VM where the observer is hosted.

Upon gathering sufficient training data on all possible network processes triggered by the VMs, the Observer labels as suspicious each process PID if (a) the CPU activity is above a given threshold τ, and (b) PID is a network process. Specifically, with respect to (b), upon crossing the τ CPU threshold activity, the Observer checks the event logs which are downloaded from the monitored VM to the trusted VM hosting the Observer and Defender, to identify if a particular external host or a group of hosts has been trying to ping or otherwise activate the process. If the increase in activity is indeed caused by external systems, the Observer alerts the Defender to check for possible attack patterns.

5.2 Defender

Once it receives the IDs of the VMs, the corresponding suspected processes and the exceptions from the Observer, the Defender starts searching for attack-specific patterns. It specifically starts monitoring for patterns if the network processes reach a high load due to network events, per the information obtained from the Observer. Each pattern consists of a particular sequence of exceptions, wherein both the type of exceptions observed, their order, as well as the frequency of particular system calls within the sequence matter. Of course, system calls may be suppressed by a sophisticated attacker at the originating VM. Hence, before searching for such patterns the Defender completes a sanity check, by verifying whether the observed $AVG_{\overline{T}}^{PID}$ over a normalized time interval $\overline{T}$ matches the corresponding $AVG_{\overline{T}}^{PID}$ stored in the system during training, per each suspicious PID. If no system calls are suspected suppressed, the pattern search starts. Otherwise, the process halts and a suspicious activity is detected.

As discussed in Section 4, load-variation attacks can be carried out in multiple ways, all of which result in different patterns of system calls. Due to this "polymorphic" nature of load-variation attacks a single approach may not suit all the possible ways according to which the load-variation is measured. Therefore, similar to an intrusion detection system, it is possible to implement various pattern recognition methods or security policies, zeroing-in the different forms of these attacks. We provide the discussion of two sample patterns that can be detected. The selected examples are representative of (1) the load-variation technique which requires the least effort from the attacker end, and (2) the most well-known load variation, based on prime, probe and trigger.

Load Variations by simple HTTP requests The Defender checks for patterns that involve socket creation, connection acceptance and socket deletion. These calls include `sys_accept` (Accepts 3-way TCP handshake), `sys_poll` (waits for http request), and `sys_read` (reads payload). `sys_socketcall` is often seen during various stages of a socket based connection as it supports a number of sub-commands to create, open and close the socket. `sys_accept` and `sys_poll` are

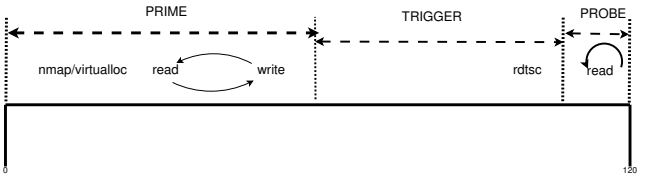

Fig. 2. Typical pattern of a Probe, Prime, Trigger Load-based attack

seen early on during the socket creation stage. In case of an attack, these calls are repeated multiple times in quick succession within a short time period due to the large number of sockets created for causing a load variation. In particular, the larger the number of sockets created in a given time unit (roughly 120 seconds), the greater the chance of an attack.

Probe-Prime-Trigger Method The Defender uses the mapping provided by the Observer to identify the suspicious exceptions generated in the prime, probe or trigger phases of load measurement. The pattern followed by load-based attacks includes a combination of system calls and interrupts generated during load measurement. Precisely, `mmap` for Linux, `virtualloc` for Windows, `read()`, `write()`, `malloc` and `rdtsc` are used. No additional interrupts or system calls are needed. When load measurement is being carried out, these exceptions occur continuously over a time period of 90 to 120 seconds depending on the attack duration: the `mmap` call or the `virtualloc`, followed by the `read()`, and then the `write()` occur repeatedly (the sequence occurs over 5 to 10 times due to repeated priming) during the first half, followed by a long gap after which another `rdtsc` calls occur to signal the trigger. The `rdtsc` is then followed by repeated `read()` to signal the probe phase. This pattern is shown in Fig. 2

Note that the challenge of pattern detection lies in the fact that these system calls generated in any variant of the attacks occur during normal operations of the VM also. The detection of some variants, such as the one using prime probe and trigger is more accurate (see Section 6) because the attacker has to carry out these steps within a set time and in a given order, generating a more recognizable pattern, than say just HTTP (or FTP or POP3) requests. While HTTP requests occur every day, the probability of seeing the steps of prime probe and trigger in the wild are lesser than seeing a HTTP request.

6 Experimental Evaluation

In this Section we discuss the tests performed to validate our framework's accuracy, and scalability. The tests were performed on Azure's infrastructure [3]. The testbed includes 5 Windows Servers, and is described in Section 4.2. One of server implemented the Observer and Defender, and is assumed trusted. The main goal of evaluation on Azure was to ascertain the detection accuracy during an attack in a noisy channel. The channel noise was varied roughly in steps of about 50000 CPU cycles (it is not possible to accurately control the noise level)

from 200000 to 900000 cycles. Channel noise was created by loading 3 out of the 5 co-hosted VMs. The noise was measured by taking load samples from the cache at any one of these 3 VMs.

First, we conducted two experiments: The first set of experiments was aimed at detecting the accuracy of co-residence detection using the simple load-variation technique. Channel noise was generated by the 3 co-hosted VMs by running multiple subsequent programs on each VM. In the first 346 ms in CPU time, the VMs execute in parallel a program which performed route finding algorithm, so as to cause excessive file reads and writes. In the following 346 ms of CPU time, in each of the 3 VMs, using LoadUI we simulate 100 users' activities on a 10KB Web page. Finally, both the route algorithm and the LOADUI users activities were executed for the remaining 346 ms/CPU. We varied the threshold of detection τ from 20% to 10%. For each set threshold, we increased the channel noise as described above. In all of our experiments, we had no false positives, but rather had a decreasing number of false negatives as we decreased the threshold. Precisely, with τ set at 20% and 18%, we reported 5 false negatives when then channel noise was higher than 620000 CPU cycles. Henceforth, the overall accuracy was low, only 58%. With τ set at 16% and 14% the accuracy increased to 66.7%. Our best results are obtained when $\tau = 10\%$, wherein 83.3% accuracy was achieved.

The second set of experiments used the prime+probe+trigger to carry out the load variations. The same experimental settings were used. As for the experiment above, we tested various detection thresholds, from 20% to 10%. The overall accuracy is consistently higher than the simple load variation attack, for all cases. Accuracy ranges from 66.7% for $\tau = 20\%$ to over 90% (for $\tau = 10\%$). All of the errors were false negative, and reported when the noise was 670000 or above. The only false negative reported with $\tau = 10\%$ was experienced when the channel was at its highest, above 880000. The improvement in detection accuracy occurs due to the unique pattern of system calls that occurs during the prime, probe and trigger phases (see Figure 2).

We then executed a final experiment wherein we tested how sensitive our mechanism is to high volume of noise, and whether and to what extent this can lead to false positives. The final test was conducted on a similar set up as the other two tests, except this time there was no attacking VM. We had three executions: in one run the load variations were caused using LoadUI, in the second run they were caused using the route planning algorithm, and in the third run either the LoadUI or the route planning algorithm were used. The load was again varied from 2000000 to 8800000 over an average of 346 ms in CPU cycles. τ was varied too, between 10% and 20%. Results are reported in Table 1. We notice that the higher τ, the lower the rate of false positives. Intuitively, this is explained as one of the VMs causes a load variation, meeting one of the conditions for a probable attack earlier in case of a low threshold. Therefore, there is always a tradeoff between the number of false positive and false negatives, and the percentage above minimum must be decided according to which is more tolerable.

Table 1. Deceptive loads: load variation was divided amongst all the 4 co-resident VMs. 1 denotes a correct true negative, 0 denotes a false positive.

CPU/Threshold	20%	18%	16%	14 %	12%	10%
200,000	1	1	1	1	1	1
230,000	1	1	1	1	1	1
300,000	1	1	1	1	1	1
340,000	1	1	1	1	1	1
420,000	1	1	1	1	1	1
480,000	1	1	1	1	1	1
520,000	1	1	1	1	1	1
570,000	1	1	1	1	1	1
620,000	1	1	1	1	1	1
670,000	1	1	1	0	0	0
740,000	1	1	0	0	0	0
790,000	0	0	0	0	0	0
840,000	0	0	0	0	0	0

7 Conclusion

In this paper, we studied covert side-channel attacks that arise as a consequence of virtualization in Cloud systems. We proposed a framework to identify load-based attacks according to our analysis of system calls generated by the attacker and the victim. Our evaluation, conducted on a real Cloud testbed, demonstrates the accuracy of our approach. We plan to extend our architecture to extract more probing patterns, and additional polymorphic forms of existing attacks.

Acknowledgement. The work from Squicciarini was partly funded under the auspices of the National Science Foundation Project # 1250319.

References

1. Amazon Web Services, `http://aws.amazon.com/`
2. Aviram, A., Hu, S., Ford, B., Gummadi, R.: Determinating timing channels in compute clouds. In: Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop, CCSW 2010, pp. 103–108. ACM (2010)
3. Chappell, D.: Windows Azure (2009), `http://www.microsoft.com/windowsazure/resources/`
4. Choi, T., Acharya, H.B., Gouda, M.G.: Is that you? Authentication in a network without identities. Int. J. Secur. Netw. 6(4), 181–190 (2011)
5. Christodorescu, M., Sailer, R., Schales, D.L., Sgandurra, D., Zamboni, D.: Cloud security is not (just) virtualization security: a short paper. In: Proceedings of the 2009 ACM Workshop on Cloud Computing Security, pp. 97–102. ACM (2009)
6. Cisco. Cloud Security: Choosing the right email security deployment (2010)
7. Cleemput, J.V., Coppens, B., De Sutter, B.: Compiler mitigations for time attacks on modern x86 processors. ACM Trans. Archit. Code Optim. 23, 1–23 (2012)

8. Cochrane, N.: Security experts ponder the cost of cloud computing (2010), http://www.itnews.com.au/news/174941,security-experts-ponder-the-cost-of-cloud-computing.aspx
9. Hay, B., Nance, K., Bishop, M.: Storm clouds rising: Security challenges for iaas cloud computing. In: Hawaii International Conference on System Sciences, pp. 1–7 (2011)
10. Kandukuri, B.R., Paturi, V.R., Rakshit, A.: Cloud security issues. In: IEEE International Conference on Services Computing, pp. 517–520 (2009)
11. Kong, J., Aciicmez, O., Seifert, J.-P., Zhou, H.: Deconstructing new cache designs for thwarting software cache-based side channel attacks. In: Proceedings of the 2nd ACM Workshop on Computer Security Architectures, pp. 25–34. ACM (2008)
12. Kong, J., Aciicmez, O., Seifert, J.-P., Zhou, H.: Hardware-software integrated approaches to defend against software cache-based side channel attacks. In: High Performance Computer Architecture, pp. 393–404. IEEE (February 2009)
13. Okamura, K., Oyama, Y.: Load-based covert channels between xen virtual machines. In: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC 2010, pp. 173–180. ACM (2010)
14. Ole: loadui: A uniquely cool approach to interactive distributed load testing. In: DevoXX - The Java Community Conference (2010)
15. Rackspace, http://www.rackspace.com/
16. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 199–212. ACM (2009)
17. Sun, P., Shen, Q., Chen, Y., Wu, Z., Zhang, C., Ruan, A., Gu, L.: Poster: LBMS: load balancing based on multilateral security in cloud. In: Proc. of the 18th ACM Conference on Computer and Communications Security, pp. 861–864. ACM (2011)
18. Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on aes, and countermeasures. J. Cryptol. 23(2), 37–71 (2010)
19. VanTil, S.: Study on cloud computing security: Managing firewall risks (2011), http://resource.onlinetech.com/study-on-cloud-computing-security-managing-firewall-risks/
20. Wang, Q., Wang, C., Li, J., Ren, K., Lou, W.: Enabling public verifiability and data dynamics for storage security in cloud computing. In: ESORICS, pp. 355–370 (2009)
21. Wang, X., Jhi, Y.-C., Zhu, S., Liu, P.: Behavior based software theft detection. In: Proceedings of the 16th ACM Conference on Computer and Communications security, CCS 2009, pp. 280–290. ACM (2009)
22. Wang, Y., Wei, J.: Viaf: Verification-based integrity assurance framework for mapreduce. In: Proc. of the 2011 IEEE 4th International Conference on Cloud Computing, CLOUD 2011, pp. 300–307. IEEE Computer Society (2011)
23. Wei, J., Zhang, X., Ammons, G., Bala, V., Ning, P.: Managing security of virtual machine images in a cloud environment. In: Proceedings of the 2009 ACM Workshop on Cloud Computing Security, CCSW 2009, pp. 91–96. ACM (2009)
24. Zander, S., Armitage, G., Branch, P.: A survey of covert channels and countermeasures in computer network protocols. Commun. Surveys Tuts. 9(3), 44–57 (2007)
25. Zhang, Y., Juels, A., Oprea, A., Reiter, M.K.: Homealone: Co-residency detection in the cloud via side-channel analysis. In: Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP 2011, pp. 313–328. IEEE Computer Society (2011)