Impact of Memory DoS Attacks on Cloud Applications and Real-Time Detection Schemes

Zhuozhao Li
University of Chicago
zhuozhao@uchicago.edu

Tanmoy Sen
University of Virginia
ts5xm@virginia.edu

Haiying Shen
University of Virginia
hs6ms@virginia.edu

Mooi Choo Chuah
Lehigh University
chuah@cse.lehigh.edu

ABSTRACT

Even though memory-based denial-of-service attacks can cause severe performance degradations on *co-located* virtual machines, a previous detection scheme against such attacks cannot accurately detect the attacks and also generates high detection delay and high performance overhead since it assumes that cache-related statistics of an application follow the same probability distribution at all times, which may not be true for all types of applications. In this paper, we present the experimental results showing the impacts of memory DoS attacks on different types of cloud-based applications. Based on these results, we propose two lightweight, responsive Statistical based Detection Schemes (SDS/B and SDS/P) that can detect such attacks accurately. SDS/B constructs a profile of normal range of cache-related statistics for all applications and use statistical methods to infer an attack when the real-time collected statistics exceed this normal range, while SDS/P exploits the increased periods of access patterns for periodic applications to infer an attack. Our evaluation results show that SDS/B and SDS/P outperform the state-of-the-art detection scheme, e.g., with 65% higher specificity, 40% shorter detection delay, and 7% less performance overhead.

ACM Reference Format:
Zhuozhao Li, Tanmoy Sen, Haiying Shen, and Mooi Choo Chuah. 2020. Impact of Memory DoS Attacks on Cloud Applications and Real-Time Detection Schemes. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3404397.3404465

1 INTRODUCTION

Infrastructure-as-a-Service (IaaS) cloud providers (e.g., Amazon [1] and Microsoft [7]) provide elastic resources for tenants to deploy applications in the cloud. To maximize the resource utilization, cloud providers use the virtualization techniques (e.g., hypervisors [11, 38, 43]) to place virtual machines (VMs) from different tenants on the same physical machine (PM). Even though current hypervisors can isolate both memory and physical memory pages [49], most of the underlying hardware memory resources of a PM are still shared by its VMs from different tenants.

Previous studies [48, 49] show that a malicious tenant can exploit the multi-tenancy feature in the cloud to launch *memory Denial-of-Service (DoS) attacks* which cause severe resource contention on the shared memory resources. There are two types of memory DoS attacks: One is atomic bus locking attack that keeps sending bus locking signals to prevent other VMs from using the memory buses. The other is cache cleansing attack that keeps cleansing the cache lines of other VMs to increase the cache misses. The goals of the memory DoS attacks are similar as the goals of network DoS attacks, i.e., preventing victims from accessing certain resources and degrading the performance of the victim applications, which ultimately prevents the owner of the victim VM from offering high quality services. As a prerequisite to perform memory DoS attacks, a malicious tenant needs to intentionally *co-locate* her/his VM(s) with victim VMs on the same PM, which has been shown to be feasible in [35, 39, 46]. Results in [48, 49] show that the memory DoS attacks can cause severe performance degradation of distributed applications (e.g., Hadoop MapReduce) up to 3.7 times, and 38 times increase in the response time of an e-commerce website.

Existing solutions that partition memory resources among VMs to enhance the performance isolation [10, 16, 47, 52, 53] are not efficient because they either waste the memory resources or cannot defeat all types of memory DoS attacks. Some studies [10, 53] propose to monitor the performance of the applications running on the VMs in a PM and then migrate impacted VMs to other PMs when there is resource contention [11, 22, 37, 38, 42]. However, VM migration is not sufficient to handle memory DoS attacks because a malicious tenant can easily co-locate with other VMs of the targeted services again [35, 39, 46].

Zhang *et al.* [49] proposed to periodically detect the attacks on a VM by examining whether the cache-related statistics (e.g., the number of cache misses and cache accesses) in real time follow the same probability distribution as those statistics without attack. However, this detection method is not robust for all applications, since it postulates that cache-related statistics of an application follow a certain probability distribution at all times. Through our measurement studies, we demonstrate that this method may generate many false positives for some applications since cache-related statistics of an application may not follow the same probability distribution at different times. In addition, their method throttles VMs in order to collect real-time statistics and such throttling generates large performance overhead on the applications running on the

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404465

co-located VMs. In addition, to avoid large performance overhead, such throttling cannot be performed frequently, which increases the detection delay Thus, to effectively and efficiently defeat the memory DoS attacks, it is crucial to design a detection scheme, which is *robust* to different applications, *responsive* to the attacks, and *lightweight* (little performance overhead) on the applications running on the VMs.

To meet this critical need, in this paper, we design a detection scheme to detect memory DoS attacks. We first conduct a measurement study of different types of cloud-based applications to understand how the memory DoS attacks impact these applications. We observe that the memory DoS attacks cause significant increases/drops on the cache-related statistics. Besides, if an application has periodic cache access pattern (denoted as *periodic application*), we observe that its periodical time period is enlarged when it is under the attacks. We propose a Boundary-based Statistical Detection Scheme (*SDS/B*) and a Period-based Statistical Detection Scheme *(SDS/P)* that leverages the observations to detect the attacks. Both SDS/B and SDS/P take the cache-related statistics collected by hardware Processor Counter Monitor (PCM) as input and detects whether there is an attack. Our detection schemes satisfy all three requirements – robust (the statistical method is suitable for all applications since we utilize statistics directly observed from the attacks), responsive (the statistics are monitored in real time) and lightweight (we use lightweight PCM tools and low-complexity statistical methods).

We have implemented SDS/B and SDS/P independently, and have implemented a Statistical-based Detection System (*SDS*) that includes both SDS/B and SDS/P on a real server. The evaluation result demonstrates that SDS outperforms the previous detection scheme in [49] by up to 2% higher recall, up to 65% higher specificity, up to 40% shorter detection delay, and up to 7% less performance overhead. The contributions of our paper are:

- We have conducted measurements to study how memory DoS attacks impact the cloud applications.
- We have designed two lightweight, accurate and responsive statistical-based detection schemes called SDS/B and SDS/P to detect memory DoS attacks.
- We have implemented SDS on a real server and demonstrated the effectiveness of SDS in terms of detection accuracy, detection delay and incurred performance overhead on applications running on co-located VMs.

Below, Section 2 presents background and related works. Section 3 describes our measurement study. Section 4 and Section 5 present our main design and experiment evaluation. Section 6 discusses the broader impact and Section 7 concludes this paper.

2 BACKGROUND

2.1 Shared Hardware Memory Resources

We briefly introduce the shared hardware memory resources in clouds. In clouds, each server may have multiple processor sockets, each of which has multiple CPU cores. Each CPU core has its private L1 and L2 caches, while all the cores share the same last level cache (LLC). Current Intel processor has a ring-based bus to interconnect the CPU cores, LLC, Integrated Memory Controllers (IMCs), system agent and etc. Besides, the memory controller bus connects the LLC to the schedulers in IMC, and the DRAM bus connects the IMC schedulers to the DRAM. In this paper, we assume that the memory buses and LLC may be shared across VMs from different tenants.

2.2 Memory DoS Attacks

Atomic bus locking attack. In modern processors, several atomic operations temporally lock all the internal memory buses in the socket to guarantee atomicity [2, 6]. In the atomic bus locking attack, the attack VM of a malicious tenant generates continuous atomic locking signals by repeatedly requesting atomic operations, which prevents the co-located VMs from using the memory bus resources and degrades their application performance.

LLC cleansing attack. A VM can launch a program to evict the LLC cache lines used by other VMs on the same server, which increases the cache miss rate of the programs on those VMs and degrades the performance. In order to detect cache lines frequently used by other VMs, the attack VM first allocates a memory buffer covering the entire LLC on its own VM. Next, the attack VM accesses some cache lines belonging to each cache set and figures out the maximum number of cache lines which can be accessed without causing cache conflicts (i.e., evicting the lines loaded by itself). If this number is smaller than the set associativity, it means that other VMs have frequently occupied some cache lines in this set. Finally, the attack VM launches the LLC cleansing attack by repeatedly cleansing these cache lines.

2.3 Related Work

VM co-location. Ristenpart *et al.* [35] first identified the threat of VM co-location in the cloud, which enables a malicious tenant to conduct LLC-based attacks including memory DoS attacks. Though the providers had improved their cloud management schemes since then, the works in [12, 39, 46, 49] were still able to achieve co-location with various VM configurations in different cloud platforms at a low cost (e.g., less than $8) in the order of minutes.

LLC-based attacks. Previous studies show that a malicious tenant can exploit the sharing feature in the cloud to extract cryptographic keys from the victim VM through cache side-channel attacks [31, 50, 51], or to transfer information using cache operations [35, 45] in a way that is not allowed by the cloud providers. Unlike these LLC-based attacks that aim to extract or transfer information, the memory DoS attacks aim to maximize the effects of resource contention and hence degrade the performance of applications running on the victim VM.

VM migration. VM migration [11, 22, 30, 37, 38, 42] have been well studied in the cloud. However, simply performing VM migration when a VM's performance is affected is not sufficient to defeat memory DoS attacks, since the malicious tenant can easily co-locate with a VM of the target tenant again, as mentioned in [12, 39, 46, 49].

Performance isolation. Many studies [10, 15, 16, 20, 23, 31, 47, 52] focused on enhancing performance isolation and proposed to partition the cache or memory to different VMs based on fairness to mitigate the resource contention. However, these solutions are not effective in defeating the memory DoS attacks. The cache partitioning disallows the sharing of LLC and may result in significant wastage of LLC resources. In addition, the cache partitioning cannot

defeat the bus locking attack since the bus is still locked during atomic operations.

Memory DoS attack detection. Zhang *et al.* [49] proposed a scheme to detect memory DoS attacks. They used the two-sample Kolmogorov-Smirnov (KS) test [33] to examine whether the cache-related statistics in real time follow the same probability distribution as the statistics when there is no attack. This detection scheme cannot provide accurate, responsive detection for some types of applications, and generates large performance overhead on the applications running on the co-located VMs due to the throttling.

Unlike the work in [49], SDS uses low complexity statistical-based methods to provide accurate, responsive and lightweight detection of memory DoS attacks.

3 MEASUREMENT STUDY

To design an effective detection scheme against the memory DoS attacks, it is essential to understand how the attacks impact different types of applications. In this section, we select some representative applications in different categories (including database, machine learning, deep learning, data-intensive, web search) and study the impacts of memory DoS attacks on different applications.

3.1 Applications and Metrics

The applications we select are listed below.

Machine learning applications. We select four applications from HiBench [5] tools to study, namely Bayesian Classification (Bayes), Support Vector Machine (SVM), k-means clustering (k-means), and Principal Components Analysis (PCA). The input data for these workloads is automatically generated using the HiBench tools.

Database applications. We select the Hive [4] queries (Aggregation, Join, and Scan) performing typical OLAP transactions described in [34]. The input data for these workloads is generated using the HiBench tools.

Data-intensive application. TeraSort is a standard data-intensive benchmark in Hadoop platform [3, 25–29]. Its input data is generated by the Hadoop TeraGen program.

Web search application. We select a widely used web search application PageRank from HiBench to study. PageRank is an algorithm used by Google Search to rank websites in their search engine results. The data source is generated from web data whose hyperlinks follow a Zipfian distribution.

Deep learning application. We select FaceNet, a TensorFlow implementation of the face recognizer described in [36]. The input data is the face recognition training dataset provided by Microsoft [8].

The metrics we study for these applications are listed as follows. Such cache-related statistics can be collected using the PCM tool every T_{PCM} seconds.

The number of LLC accesses every T_{PCM} time (denoted as *AccessNum*). For bus locking attack, we measure AccessNum because this attack prevents victim VM from using the memory buses to access memory.

The number of LLC misses every T_{PCM} time (denoted as *MissNum*). For LLC cleansing attack, we measure MissNum since this attack frequently evicts the data of victim VM from the LLC, which requires the victim VM to read the data from the main memory.

We conducted the measurement study on a local server with configuration comparable to a cloud server. We did not experiment on a real cloud server because the PCM tool requires privilege control to the hypervisor, which is not provided to users in the cloud. Specifically, the server has one CPU Intel Xeon E5-2660 with 14 physical cores (28 logical cores due to hyper-threading) and 32GB RAM. The LLC has 35MB and 20-way set-associative. We installed KVM hypervisor on the server and created 2 VMs with Ubuntu 14.04 installed, one functioning as the attack VM and the other as the victim VM.

3.2 Insufficiencies of KStest

In [49], given a server that provides a detection service for memory DoS attacks, the detection system (we call it *KStest*) periodically (every L_R seconds) performs the following operations for a PROTECTED VM that requests this detection service:

(1) It first stops the executions of all other VMs except the PROTECTED VM using execution throttling, and collects a set of cache-related statistics of the PROTECTED VM for W_R seconds as *reference samples*, which represent the cache-related statistics under no attack. A sample is defined as a data point of AccessNum or MissNum collected by PCM tool.

(2) It resumes the running of all other VMs and performs the following two steps once every L_M seconds: (i) It collects a set of statistics for W_M seconds as *monitored samples* for the PROTECTED VM; (ii) Next, it compares if the reference samples and the monitored samples follow the same distribution. If the distributions are different for *four* consecutive times ($4L_M$ seconds), then it will declare that there is an attack.

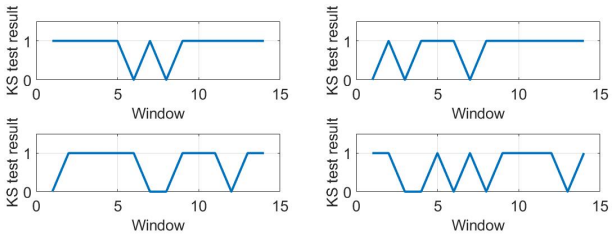
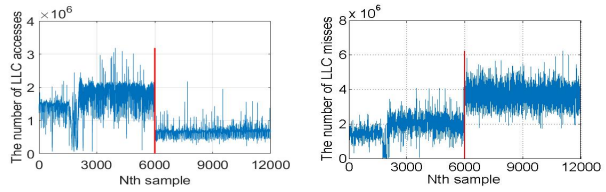


Figure 1: KStest results of TeraSort – no attack launched.

In [49], the authors show that KStest works well for applications including web, database, memcached and load-balancer applications in e-commerce. However, they did not evaluate many other typical cloud-based applications. To evaluate the effectiveness of KStest of other applications, we ran TeraSort on the victim VM without any attack. In the experiments, we followed the same KStest steps and settings of parameters as in [49], i.e., $T_{PCM} = 0.01s$, $W_R = W_M = 1s$, $L_M = 2s$, and $L_R = 30s$.

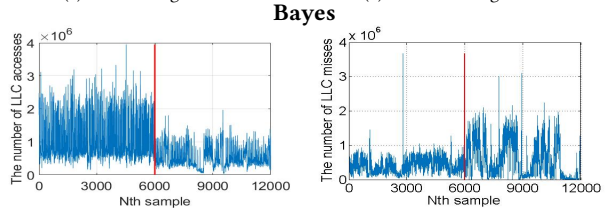
The four plots in Figure 1 show the KStest results of TeraSort for four L_R intervals (from twenty L_R intervals) when there is no attack. In the plots, a value 1 indicates that the two sets of samples have distinct probability distributions; a value 0 indicates that they have the same distribution. We see that even when there is no attack, the probability distributions for TeraSort at different times may not be the same (i.e., KStest=1). All these four plots indicate that this KStest method will declare there is an attack since there are more than four consecutive "1"s in the plots. Besides, from the KStest results of all the twenty L_R intervals in our experiment, we found that more than

60% of them indicate that there is an attack. Thus, applying KStest to detect attacks is highly likely to generate many false positives.



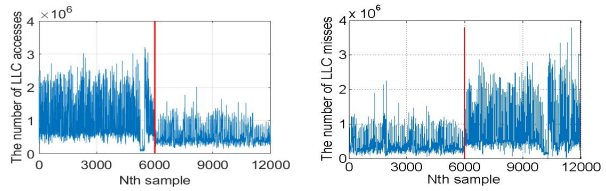
(a) Bus locking attack (b) LLC cleansing attack

Bayes



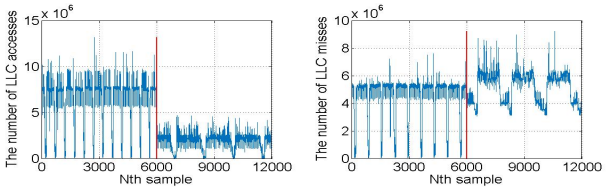
(c) Bus locking attack (d) LLC cleansing attack

SVM



(e) Bus locking attack (f) LLC cleansing attack

k-means



(g) Bus locking attack (h) LLC cleansing attack

PCA

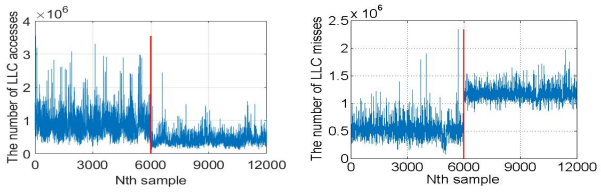
Figure 2: Machine learning applications.

We also tested other typical cloud applications. From the KStest results of all twenty L_R intervals in our experiments, KStest declares an attack around 30% of the times in Bayes, 35% in SVM, 20% in k-means, 60% in PCA, 40% in Aggregation, 40% in Scan, 30% in PageRank, 55% in FaceNet when the attack is absent.

3.3 Impact of Memory DoS Attacks

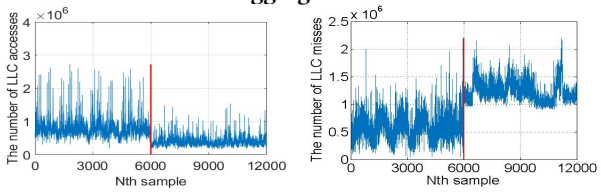
To study the impact of the attacks, we collected the cache-related statistics of each application for 120 seconds; for the first 60 seconds, the application ran normally without being attacked, but in the next 60 seconds, the application was under either a bus locking attack or LLC cleansing attack. In the result figures below, the red lines separate the two stages – without and with attack.

Bus locking attack. Figures 2(a), 2(c), 2(e), 2(g), 3(a), 3(c), 3(e), 4(a), 5(a) and 6(a) show the AccessNum over the 120s for five different types of applications under the bus locking attack. From all the figures, we notice that the AccessNum for every application suffers significant drop after the bus locking attack is launched. This is because the bus locking attack sends the lock signals to lock all



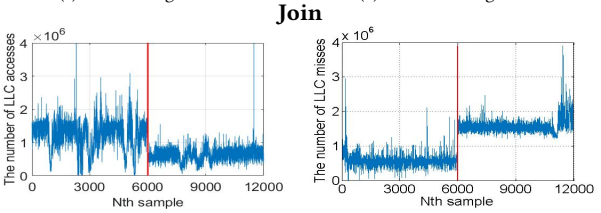
(a) Bus locking attack (b) LLC cleansing attack

Aggregation



(c) Bus locking attack (d) LLC cleansing attack

Join



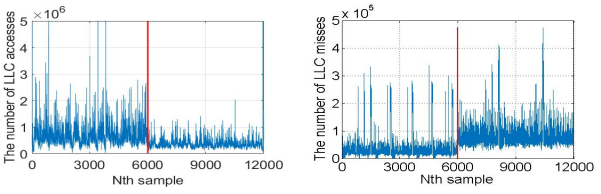
(e) Bus locking attack (f) LLC cleansing attack

Scan

Figure 3: Database applications.

buses in the processor socket, which prevents the applications from accessing LLC.

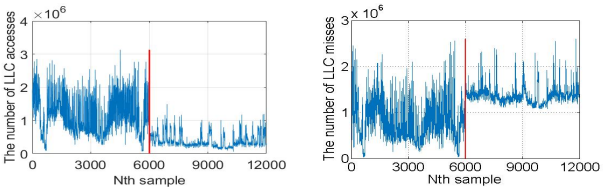
In addition, we clearly see that PCA in Figure 2(g) and FaceNet in Figure 6(a) have periodic patterns of AccessNum, i.e., the same LLC access patterns repeat periodically with a regular time *period*. This is because such applications often repeatedly perform the same computations on different batches of data. We call such applications with periodic patterns in cache-related statistics *periodic applications*, and other applications that do not have such periodic patterns as *non-periodic applications*.



(a) Bus locking attack (b) LLC cleansing attack

TeraSort

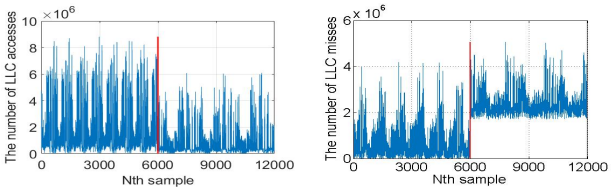
Figure 4: Data-intensive applications.



(a) Bus locking attack (b) LLC cleansing attack

PageRank

Figure 5: Web search applications.



(a) Bus locking attack (b) LLC cleansing attack

FaceNet

Figure 6: Deep learning applications.

When a periodic application is under attack, in addition to the drop in AccessNum, we also observe from Figures 2(g) and 6(a) that the period at which such patterns repeat themselves increases. This can be explained as follows: Originally, the application requires a certain amount of time to finish processing a batch of data. When it is under attack, the application requires a longer time to finish its computations on the same batch of data.

LLC cleansing attack. Figures 2(b), 2(d), 2(f), 2(h), 3(b), 3(d), 3(f), 4(b), 5(b) and 6(b) show the MissNum over the 120s for five different types of applications under LLC cleansing attack. We observe that the MissNum for all the applications increases after the LLC cleansing attack is launched. This phenomenon occurs because the cleansing attack continuously cleanses the LLC, which increases the MissNum. In addition, the period of the periodic applications also increases (shown in Figures 2(h) and 6(b)) after the LLC cleansing attack starts since more time is needed to process the same batches of input data.

3.4 Exploration and Summary

To design a detection scheme, we explored different approaches to analyze the cache-related statistics. For example, we expected that when there is no attack, the cache-related statistics at different times would be more correlated with each other than that when there is an attack. Thus, we explored the spectral coherence [32], cross-correlation [41] and Pearson correlation [24] approaches on many applications including Bayes, SVM, k-means, PCA, Aggregation, Join, Scan, TeraSort, PageRank and FaceNet. However, we discover that these approaches are not useful for detecting both attacks since the correlations among the cache-related statistics do not show any decreasing trend after the attacks are launched.

We summarize our observations in the measurement:

- *Observation(1): All applications suffer a significant AccessNum decrease in the bus locking attack and a significant MissNum increase in the LLC cleansing attack.*
- *Observation(2): The periodic applications show prolonged periodicity for both kinds of attacks.*

Based on Observation(1), we design SDS/B that exploits the decrease/increase in the cache-related statistics to detect the attacks. Based on Observation(2), we also devise SDS/P for periodic applications that leverages the increase in the period of repeatable cache access patterns to detect the attacks.

4 DESIGN OF SDS

In this section, we introduce the design of SDS, which will be deployed in the hypervisor on each server by the provider.

4.1 Data Preprocessing

To collect raw cache-related statistics, the cloud providers run the PCM tool on the hypervisor of each server. We use $\{A_1, A_2, ...\}$ to denote the real-time statistics (i.e., AccessNum for bus locking attack and MissNum for LLC cleansing attack). We define a time series as a series of data points collected/computed in real time.

Due to the significant decrease (increase) in the cache-related statistics, one naive detection approach is to trigger the alarm when a data point A_i drops (increases) by a threshold (e.g., 50%) of prior data point A_{i-1}. However, it is not unusual that the cache-related statistics of many practical applications to have random variations over time. Thus, directly thresholding the raw data may lead to inaccurate detection of attacks.

To overcome the challenge of random variations, we propose to use sliding window based moving average (MA) [44]. In MA, we use W to denote the window size and ΔW to denote the step size for moving windows. We compute the average of W data points at a time. When ΔW new data points become available, we slide the window ΔW ahead and a new average of W data points in the new window is computed. This process is repeatedly conducted in real time. We denote M_n as the average of the raw data points $\{A_{1+n*\Delta W}, A_{2+n*\Delta W}, ..., A_{W+n*\Delta W}\}$ in the n^{th} window, i.e.,

$$M_n = \frac{1}{W} \sum_{i=1+n*\Delta W}^{W+n*\Delta W} A_i. \quad (1)$$

Based on the ahove process, by the n^{th} window, we will have a time series $\mathbf{M} = \{M_0, M_1, ..., M_n\}$.

To smooth the data further, instead of using the simple MA that gives all the past observations equal weight, we use a well-known enhancement called exponential weighted moving average (EWMA) which assigns exponentially decreasing weights to prior data over time, i.e.,

$$S_n = \begin{cases} M_0, & \text{if } n = 0 \\ (1-\alpha)S_{n-1} + \alpha M_n, & \text{otherwise} \end{cases} \quad (2)$$

where S_n is smoothed result at the n^{th} window, and $0 < \alpha < 1$ is the smoothing factor. A larger value of α reduces the level of smoothing and gives higher weight to recent data.

4.2 Detection Design

In this section, we describe SDS/B for both non-periodic and periodic applications, and propose SDS/P for periodic applications to detect the attacks.

4.2.1 All Applications. Considering that the bus locking attack and the LLC cleansing attack present different impacts on the applications (i.e., drops in AccessNum for bus locking attack and increases in MissNum for LLC cleansing attack), we propose SDS/B where the collected cache-related statistics typically lie within a range with a lower and upper bound so that we can flag an anomaly when the collected statistics go out of bound.

When there is no attack, let us denote the mean of as μ_E and the standard deviation of the time series of EWMA as σ_E. We propose to define a normal range as $[\mu_E - k\sigma_E, \mu_E + k\sigma_E]$, where $k > 1$ is a pre-defined boundary factor. When an EWMA value S_n becomes available in real time, we check if the condition satisfies

the following

$$C_n = (S_n < \mu_E - k\sigma_E) \text{ or } (S_n > \mu_E + k\sigma_E). \quad (3)$$

SDS/B triggers an attack alarm at time n if the EWMA values are out of the normal range consecutively for H_C times. Such a threshold is used to avoid false positives.

Several questions still need to be answered:

How to determine the mean μ_E and the standard deviation σ_E of each benign VM? It is reasonable to assume that a benign VM is in a safe state (i.e., not under any attack) immediately after it is newly started or migrated, since the malicious tenant needs to conduct VM co-location again. The providers can collect the cache-related statistics of a benign VM at that time.

How fast can the attacks be detected? When there is an attack, the EWMA values are expected to become anomalous. Since SDS/B needs H_C consecutive anomalies to trigger the alarm, the time to detect the attack is no shorter than collecting H_C EWMA values. Recall from Equations (1) and (2) that only when ΔW new raw data points are collected in real time, SDS/B will compute the next EWMA value. The time for a new raw data point depends on the sampling time T_{PCM} of the PCM tool. As a result, the shortest detection delay for SDS/B is $H_C \cdot \Delta W \cdot T_{PCM}$ time.

How to select appropriate constant k and H_C to guarantee high detection accuracy? The cloud provider needs to select appropriate k and H_C to guarantee that the attacks can be detected with a certain confidence level. Since there are many applications running in the cloud, the chosen parameters should work for all applications so that the cloud providers do not need to use different values for different applications.

If we can model the cache-related statistics of all applications using some common probability distributions (e.g., Gaussian Distribution), we could derive the parameters with certain confidence using the properties of the modeled distribution. However, due to the large variety of cloud-based applications, it is hard to use one probability distribution to summarize all applications. Thus, we leverage the *Chebyshev's inequality* [19] that can be applied to any probability distributions to infer the parameters for all the applications so that a certain confidence level is guaranteed. Let $Pr(\cdot)$ represent the probability, and X be a random variable with mean μ and non-zero standard deviation σ. The Chebyshev's inequality describes that

$$Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}. \quad (4)$$

Based on Chebyshev's inequality, the EWMA values without attack will be outside the range (condition C_n) with a probability of at most $\frac{1}{k^2}$ and H_C consecutive occurrences of such condition occurs with a probability of at most $(\frac{1}{k^2})^{H_C}$. Thus, k and H_C can be chosen based on the desired confidence level that there is an attack when an alarm is triggered. For example, if we desire a 99.9% confidence, the probability that a false alarm is triggered should be as low as 0.001. Based on Equation (4), we could have many options to select k and H_C, e.g., $k = 2, H_C = 6$ or $k = 1.125, H_C = 30$.

Typically, we expect that k (i.e., the normal range) is small to avoid false negatives but sufficiently big to avoid false positives. Given a confidence level, H_C decreases as k increases and vice versa. Thus, there is a tradeoff between k and H_C: larger k may generate false negatives while larger H_C may result in higher detection delay.

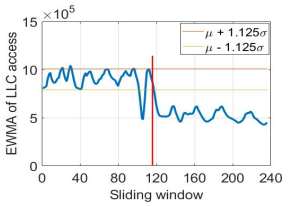


Figure 7: Detection example of k-means.

Experiments in Section 5.3 show that setting k close to 1 achieves a good tradeoff.

Procedure. We show how SDS/B works for an application. We first profile the μ_E and σ_E of the application with $W = 200$, $\Delta W = 50$ and select $k = 1.125$. If we expect to achieve 99.9% confidence level, based on Equation (4), we need to select $H_C = 30$. Take k-means application as an example, Figure 7 shows its monitored EWMA time series in real time and the normal range (i.e. $[\mu_E - 1.125\sigma_E, \mu_E + 1.125\sigma_E]$). We see that before the bus locking attack starts (red line), the EWMA values drop below the normal bound but do not reach H_C times, and hence SDS/B does not trigger the alarm. After the attack starts, the EWMA values drop below the normal bound again and this time the alarm is triggered at around window 150.

4.2.2 Periodic Applications. We propose SDS/P leveraging the property in Observation (2) described in Section 3. For periodic applications, both SDS/B and SDS/P can be used independently to detect the attacks. Our experiments in Section 5 show that SDS/B and SDS/P can both achieve high detection accuracy, low detection delay and low performance overhead. We could also use both schemes together for periodic applications to increase the detection accuracy.

Specifically, SDS/P computes the period of MA $\mathbf{M} = \{M_0, M_1, ...\}$, rather than the raw data or EWMA, since MA reduces the variations of raw data as mentioned in Section 4.1 which increases the detection accuracy, while the EWMA time series of a periodic application may not have periodic patterns.

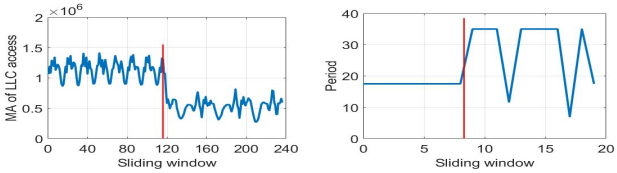
To compute the period of a time series, we could use the Discrete Fourier Transform (DFT) [13] to locate the *dominant frequency*, which is defined as the frequency that has the maximum amplitude and is equal to the reciprocal of the period. However, DFT may detect false frequencies that do not exist in the time series [18]. Auto Correlation Function (ACF), another method for detecting repeated patterns, can avoid false detection of frequencies of a time series [17], but may result in the detection of multiples of a true period [40]. For example, given a time series that has a period of 30 seconds, {60s, 90s, . . .} are also falsely considered as the periods in ACF. Therefore, solely using DFT or ACF cannot accurately determine the true frequencies in a time series. To more accurately find the period, we adopt the approach (denoted as DFT-ACF) in [40] that first generates candidate periods using DFT and then uses ACF to identify the real period of the EWMA time series.

However, we need to solve two challenges before we can use DFT-ACF in [40]. First, we need to check whether an application is periodic. As in Section 4.2.1, we can use DFT-ACF immediately after a VM is newly started or migrated to check if there exists a relatively constant period where MA patterns repeat. If yes, we declare the application to be periodic.

What size of a MA time series should be used to find the period? We define the size of a time series as the number of data points it contains. Choosing an appropriate size of a MA time series for computing the period is important. When an attack starts running, the abnormal MA values start to appear. If SDS/P uses a long MA time series to compute the period, DFT-ACF may still infer a normal period since normal MA values still dominate the time series. As a result, it may take a long time for SDS/P to detect the abnormal period, resulting in long detection delay. In addition, the DFT computation of a time series has a computation complexity of $O(N \log N)$, where N is the size of the time series. Larger N is going to result in higher computation cost.

To tackle such challenges, we propose to monitor the MA time series with a size of W_P. Each time, when ΔW_P MA values become available, we compute the period of the latest W_P MA values and check if the newly computed period is the same as the prior normal period. When H_P consecutive computed periods are not the same as the normal period of the application (20% difference), SDS/P triggers the alarm that there is an attack. Specifically, we select $W_P = 2p$, where p is the period of the cache-related statistics without attack. This is because $2p$ MA values are sufficient to determine the correct period of the applications in the case of no attack. On the other hand, ΔW_P determines the computation overhead and detection delay – intuitively, smaller ΔW_P results in higher computation overhead but smaller detection delay and vice versa. Therefore, ΔW_P should be selected to balance the tradeoff between the computation overhead and detection delay to make both acceptable. Our sensitivity analysis in Section 5.3 shows that the computation overhead is negligible and hence we can select a small ΔW_P (e.g., 10).

How fast can an attack be detected? Similar to the analysis in Section 4.2.1, the detection scheme can detect the attacks as fast as $H_P \cdot \Delta W_P \cdot \Delta W \cdot T_{PCM}$ second. Note that the computation time of DFT-ACF is negligible here.



(a) FaceNet MA time series. (b) The sequences of computed period for FaceNet.

Figure 8: Detection example of FaceNet application.

Procedure. We show how SDS/P works for a periodic application. We first profile how the cache-related statistics of the application repeats every p MA values in its MA time series using $W = 200$, $\Delta W = 50$. We then select $W_P = 2p$, $\Delta W_P = 10$, $H_P = 5$. When the PCM tool starts to monitor the cache-related statistics, the detection approach derives the MA time series and computes the period in real time. Take FaceNet application as an example, Figure 8(a) shows its MA time series in real time and Figure 8(b) shows the computed period in real time. It shows that before the attack starts, the period remains constant at around 17. However, after the attack starts, the period deviates from the normal period 5 times, which triggers the alarms.

5 PERFORMANCE EVALUATION

5.1 Implementation and Setting

We implemented prototype systems for both schemes SDS/B and SDS/P and also implemented SDS, which includes both SDS/B and SDS/P. In SDS, for non-periodic applications, only SDS/B is used to infer an attack. For periodic applications, SDS requires both SDS/B and SDS/P to detect an attack before triggering an attack alarm.

The parameters involved in SDS/B and SDS/P are shown in Table 1. The parameters were selected empirically to achieve a balance between detection accuracy and detection delay.

Table 1: Parameters in the experiment.

Parameters	Values
T_{PCM}	0.01
Window size W of raw data	200
Sliding step size ΔW	50
EWMA smooth factor α	0.2
Upper bound	$\mu + 1.125\sigma$
Lower bound	$\mu - 1.125\sigma$
Consecutive violation threshold H_C	30
Window size W_P in SDS/P	2 * period
Sliding step size ΔW_P in SDS/P	10
Consecutive period change threshold H_E	5

We ran our detection scheme on a server using the same hardware mentioned in Section 3.2. As in [49], we deployed a victim VM and 8 other VMs to share the resources on the server. Among these 8 VMs, one of them was the attack VM that performed the memory DoS attacks (bus locking attack or LLC cleansing attack), and the other 7 VMs were all benign VMs that ran normal Linux utilities such as sysstat and dstat. As to the prior-selected victim VM, it ran one of the applications introduced in Section 3.

The evaluations for SDS/B, SDS/P and SDS can be divided into three stages. We first generated the profile of an application without attack and computed the boundary range from the mean and standard deviation (Stage 1). Later we ran each application for 600 seconds. During the first 300 seconds (Stage 2), we did not launch any attacks on the attack VM. During the last 300 seconds (Stage 3), we performed the bus locking attack or LLC cleansing attack from the attack VM. For each application, the median, 10^{th}, 90^{th} percentiles of 20 runs were reported. In order to compare with KStest, we set the parameters of KStest exactly the same as the experiments in [49] listed in Section 3.2.

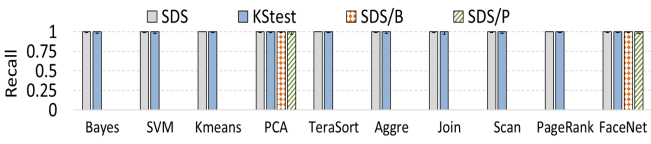
5.2 Evaluation Results

In this section, we evaluate the detection accuracy, detection latency, and performance overhead.

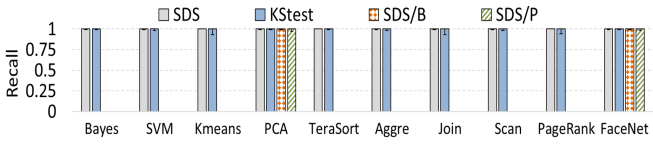
Detection accuracy. To evaluate the accuracy, we will use the following metrics.

- *Recall* is defined as $\frac{\text{true positives}}{\text{true positives+false negatives}}$, which measures the ability to detect an attack when it is present.
- *Specificity* is defined as $\frac{\text{true negatives}}{\text{true negatives+false positives}}$, which measures the ability to correctly infer no attack when the attack is absent.

Figures 9(a) and 9(b) show the recall results of bus locking attack and LLC cleansing attack, respectively. We see that the median

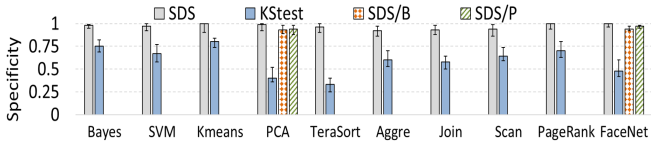


(a) Recall under bus locking attack.

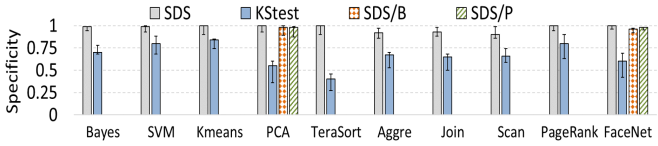


(b) Recall under LLC cleansing attack.

Figure 9: Recall results. Bars give median values and the error bars give the 10^{th} and 90^{th} percentile values (low variance here).



(a) Specificity under bus locking attack.



(b) Specificity under LLC cleansing attack.

Figure 10: Specificity results. Bars give median values and the error bars give the 10^{th} and 90^{th} percentile values.

(a) Detection delay under bus locking attack.

(b) Detection delay under LLC cleansing attack.

Figure 11: Detection delay results. Bars give median values and the error bars give the 10^{th} and 90^{th} percentile values.

recalls of both SDS and KStest are 100%, regardless of the applications or the types of attacks, indicating that there are few false negatives. Based on the 10^{th} and 90^{th} percentiles, the recall of SDS is slightly better (1-2%) than KStest. Figures 10(a) and 10(b) show the specificity results of bus locking attack and LLC cleansing attack, respectively. We see that the specificity that SDS achieves is around 90-100%, while KStest only achieves a specificity only around 30-80% due to many false positives in KStest. In summary, SDS achieves 1-2% better recall and 20-65% better in specificty than KStest, demonstrating the effectiveness of SDS in terms of reducing the number of false positives.

In addition, for periodic applications (PCA and FaceNet), we also evaluated SDS/B and SDS/P. SDS/B and SDS/P independently exploit our Observations (1) and (2) mentioned in Section 3, respectively. The figures show that SDS/B can detect the attacks with a 100% recall and 94-97% specificity, and SDS/P can detect the attacks with 100% recall and 93-94% specificity. Therefore, we see that solely using SDS/B and SDS/P can achieve high recall and specificity for periodic applications. Using both approaches, SDS improves SDS/B and SDS/P by 3-6% and 5-6% of specificity, respectively, since applying both approaches can eliminate some false positives. The results demonstrate the effectiveness of SDS, SDS/B and SDS/P in terms of detection accuracy.

Detection delay. We also evaluated the detection delay of SDS. We define the detection delay as the duration between the time when an attack is launched (which is known as we launch the attacks in the experiments) and the time when the attack is detected.

Figures 11(a) and 11(b) shows the detection delay of different applications under bus locking attack and LLC cleansing attack, respectively. We see that for both types of attacks, the detection delays of SDS for all applications are in the range of 15-30 seconds, while the detection delays of KStest were in the range of 20-50 seconds. The detection delay of SDS is 3-20 seconds (5-40%, computed by $\frac{\text{Delay}_{KStest}-\text{Delay}_{SDS}}{\text{Delay}_{KStest}}$) shorter than KStest's. Such improvement in the detection delay is significant since previous study [9] in Amazon shows that 100ms increase in the webpage loading time decreased sales by 1 percent. This demonstrates the responsiveness of SDS. SDS has faster response time because it continuously examines if data points in the time series are anomalous in real time. On the other hand, the KStest needs to perform multiple rounds of KS tests, each of which requires KStest detector to throttle the executions of other VMs while it acquires reference samples of a protected VM. Such collection cannot be too frequent for it delays the execution of all applications. This indirectly increases the detection latency of the KStest approach.

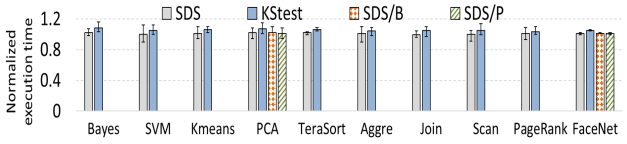


Figure 12: Performance overhead results. Bars give median values and error bars give the 10^{th} and 90^{th} percentile values.

The figures also show the detection delay of SDS/B and SDS/P for periodic applications. We see that SDS/P has larger detection delay (around 10 seconds) than SDS/B because of higher computation cost of DFT-ACF calculation. However, compared with KStest, the detection delays of both approaches are still shorter than KStest, indicating the responsiveness of SDS/B and SDS/P.

Performance overhead. In SDS, the hypervisor on each server uses PCM to measure the real-time cache-related statistics and runs a program to analyze the statistics. We measured the performance overhead of SDS on the applications running on the VMs. In this experiment, we do not launch any attacks. Figure 12 shows the normalized execution times (normalized to the execution time

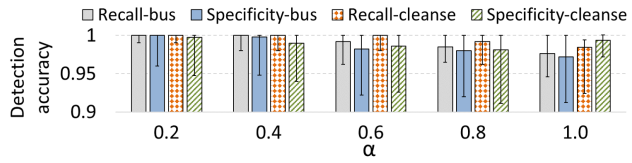
without running any detection schemes) of different applications running on the VM when the hypervisor employs different detection schemes. We see that SDS has very minimal impact on the execution times (1-2%, computed by normalized execution time minus one) of the applications running on the VMs in comparison to 3-8% of the KStest approach. This is because SDS leverages lightweight PCM and simple algorithm to detect the attacks, which generates negligible performance overhead comparing to KStest that uses throttling to collect the reference samples. The figure also shows that SDS/B and SDS/P for periodic applications also generate negligible performance overhead.

5.3 Sensitivity Analysis

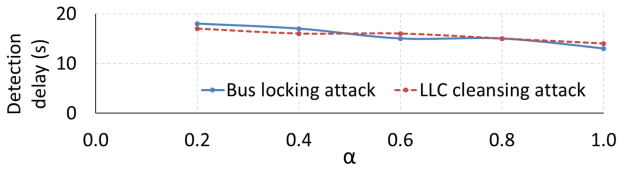
In this section, we evaluate the sensitivity of some key parameters for SDS. In the experiments below, unless the varying parameter and otherwise specified, we use the parameters in Table 1.

Due to space limitation, for the parameters regarding to the detection of periodical applications, we present the results of FaceNet only. For other parameters, we only present the results of k-means even though we have tested other applications and obtained the same conclusions. Notice that we do not present the results of performance overhead since they are not sensitive to the parameters – SDS still generates negligible performance overhead (up to 2%) even with the parameters that result in the largest performance overhead in our selection range.

Smooth factor α. In this experiment, we varied α in the range $[0.0, 1.0]$. Notice that when $\alpha = 1.0$, the EWMA time series is equivalent to the MA time series. Figure 13(a) shows the recall and specificity versus α. We see that as α increases, the recall and specificity decreases slightly. This is because a large α reduces the smoothing of EWMA values and generates a few false positives and false negatives due to the random variation. Nevertheless, the recall and specificity remain close to 1 over a wide range of α, i.e., $[0.2, 0.4]$. Figure 13(b) shows the detection delay versus α. We see that the detection delay decreases slightly as α increases. This is because after an attack is launched, the EWMA values go outside the normal range faster with larger α.



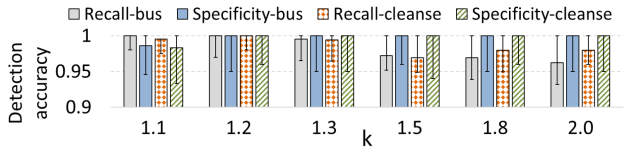
(a) Detection accuracy.



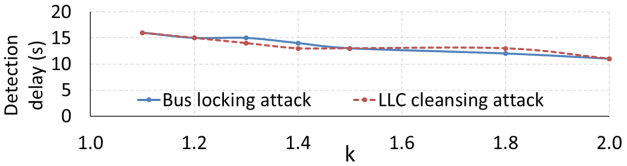
(b) Detection delay.

Figure 13: Sensitivity of α.

Boundary factor k. We varied k in the range $[1.1, 2.0]$ and the consecutive violation threshold H_C was adjusted to keep a confidence of 99.9% based on Equation (4). Figure 14(a) shows the recall and specificity versus k. We see that the specificity increases slightly as k increases, since there are fewer false positives. On the other hand, the recall decreases slightly as k increases. This is because the detection scheme sometime fails to detect the attacks due to a larger k, which results in some false negatives. Nevertheless, we see that both recall and specificity remain closed to 1 over a wide range of k values, i.e., $[1.1, 1.5]$, which allows the cloud providers to adjust k based on their desire for fewer false negatives or fewer false positives.



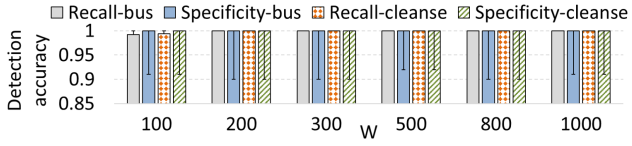
(a) Detection accuracy.



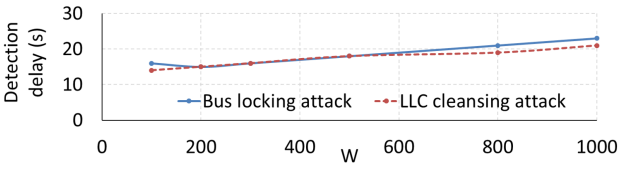
(b) Detection delay.

Figure 14: Sensitivity of k.

Figure 14(b) shows the detection delay versus k. As k increases, the consecutive violation threshold H_C decreases. Thus, the detection delay is shorter as k becomes larger. However, with a larger k, when the attack is launched, the EWMA values may drop outside the boundary range slower than that with a smaller k, which incurs a delay of a few seconds and offsets the benefit of smaller H_C.



(a) Detection accuracy.

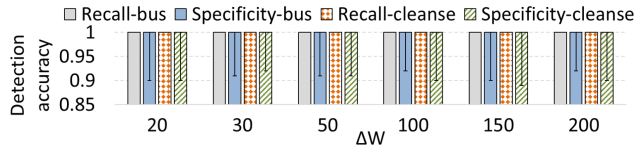


(b) Detection delay.

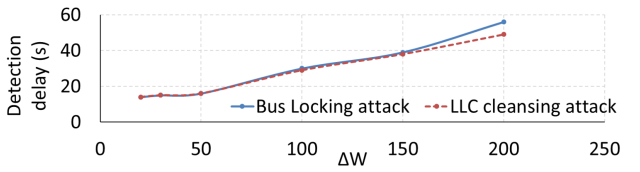
Figure 15: Sensitivity of W.

Window size W. We varied W in the range $[100, 1000]$. Figure 15(a) shows the recall and specificity versus W. We observe that varying W does not change the detection accuracy much. Only when W is 100, the recall is not 100% because W is too small to eliminate the variations in the raw data. Figure 15(b) shows the detection delay versus W. We see that the detection delay increases slightly as W increases. This is caused by the fact that the EWMA values take a longer time to go outside the normal range with a larger W value. Thus, we should select a small W but still sufficiently large

(e.g., 200) to better eliminate the variations in the raw data while minimizing the detection delay.



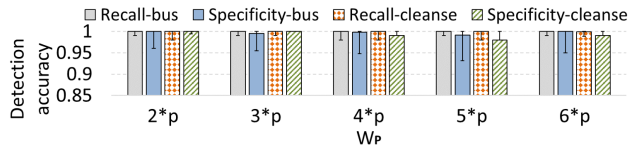
(a) Detection accuracy.



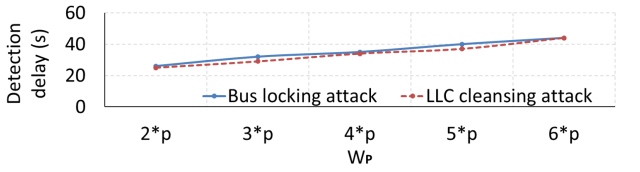
(b) Detection delay.

Figure 16: Sensitivity of ΔW.

Sliding step size ΔW. In this experiment, we varied ΔW in the range [20, 200]. Figure 16(a) shows the recall and specificity versus ΔW. We observe that varying ΔW does not change the detection accuracy. Figure 16(b) shows the detection delay versus ΔW. We see that the detection delay increases as ΔW increases. Thus, we should select a small ΔW (e.g., 10-50) in practice to reduce the detection delay.



(a) Detection accuracy.

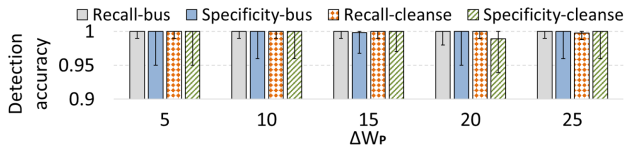


(b) Detection delay.

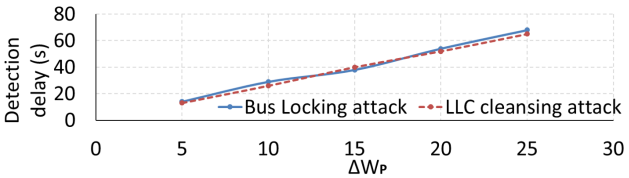
Figure 17: Sensitivity of W_P.

Window size W_P in SDS/P. In this experiment, we varied W_P in the range $[2p, 6p]$. Figure 17(a) shows the recall and specificity versus W_P. We observe that varying W_P does not change the detection accuracy. Figure 17(b) shows the detection delay versus W_P. We see that the detection delay increases as W_P increases. This is because the larger W_P, the slower the abnormal period can be detected, as mentioned Section 4.2.2. Thus, we expect W_P to be small enough but sufficiently large (e.g., $2p$) to compute the period and reduce the detection delay.

Sliding step size ΔW_P in SDS/P. We varied ΔW_P in the range [5, 25]. Figure 18(a) shows the recall and specificity versus ΔW_P. Varying ΔW_P does not change the detection accuracy. Figure 18(b) shows the detection delay versus ΔW_P. The detection delay increases as ΔW_P increases. Thus, we should select a small ΔW_P (e.g., 5-10) in practice to reduce the detection delay.



(a) Detection accuracy.



(b) Detection delay.

Figure 18: Sensitivity of ΔW_P.

6 DISCUSSION

In this paper, we focus on detecting VM based memory DoS attacks. However, in some cases, the performance of all the VMs on a PM may be affected by each other [14], even though all these VMs run benign applications. The general ideas of our work can be applied to these scenarios as well–the tenants would expect the cloud providers to detect this kind of interferences and to take proper actions (e.g., VM migrations) when they happen to prevent severe performance degradation of their applications. Moreover, as container technologies (e.g., Docker and Singularity) become popular in parallel and distributed computing, we see similar attacks on those applications and thus the broader impact of our work.

Our methods assume that we can profile the cache-related statistics of cloud applications to gather some "ground truth" beforehand and compare the real-time cache-related statistics against the ground truth to detect the attacks. This assumption is based on the fact that many cloud applications are recurring and have predictable characteristics [21]. However, an application may change dramatically under different situations (e.g., different times in a day). To address this problem, the cloud providers could allow tenants to profile the statistics under different situations, or allow tenants to request re-profiling when they notice their applications change. We plan to investigate more on how to deal with dynamic applications in the future.

7 CONCLUSION

In this paper, we have conducted a measurement study on the impact of memory DoS attacks on a variety of cloud-based applications. We observe that all the applications suffer a significant decrease in AccessNum during the bus locking attack and a significant increase in MissNum during the LLC cleansing attack. Besides, we also observe that the periodic applications show prolonged periodicity for both kind of attacks. Based on the observations, we propose two lightweight and responsive detection schemes SDS/B and SDS/P that can accurately detect the attacks. In SDS/B, we propose a boundary-based approach to infer an attack when cache-related statistics go out of bound. In SDS/P, we propose to monitor the period in real time using signal processing techniques for periodic applications. Our experiments conducted on a real server demonstrate that SDS, an implementation that includes both SDS/B

and SDS/P, can detect the attacks with 100% recall, 90-100% specificity and 15-30 seconds detection delay for a variety of applications and incurs only 1-2% performance overhead on the execution times of cloud-based applications.

ACKNOWLEDGEMENT

This research was supported in part by U.S. NSF grants NSF-1827674, CCF-1822965, OAC-1724845, and Microsoft Research Faculty Fellowship 8300751.

REFERENCES

[1] 2020. Amazon Web Service. http://aws.amazon.com/. [Accessed in June 2020].
[2] 2020. AMD architecture programmer's manual, Volume 1: Application programming. http://support.amd.com/TechDocs/24592.pdf. [Accessed in June 2020].
[3] 2020. Apache Hadoop. http://hadoop.apache.org/. [Accessed in June 2020].
[4] 2020. Apache Hive. https://hive.apache.org/. [Accessed in June 2020].
[5] 2020. HiBench Benchmark Suite. https://github.com/intel-hadoop/HiBench. [Accessed in June 2020].
[6] 2020. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System programming Guide. https://software.intel.com/en-us/articles/intel-sdm. [Accessed in June 2020].
[7] 2020. Microsoft Azure. https://azure.microsoft.com/en-us/. [Accessed in June 2020].
[8] 2020. MS-Celeb-1M: Challenge of Recognizing One Million Celebrities in the Real World. https://www.microsoft.com/en-us/research/project/ms-celeb-1m-challenge-recognizing-one-million-celebrities-real-world/. [Accessed in June 2020].
[9] 2020. Online Experiments: Lessons Learned. http://exp-platform.com/Documents/sIEEEComputer2007OnlineExperiments.pdf. [Accessed in June 2020].
[10] Jeongseob Ahn, Changdae Kim, Jaeung Han, Young-ri Choi, and Jaehyuk Huh. 2012. Dynamic Virtual Machine Scheduling in Clouds for Architectural Shared Resources.. In *Proceedings of HotCloud*.
[11] E. Arzuaga and D. R. Kaeli. 2010. Quantifying load imbalance on virtualized enterprise servers.. In *Proceedings of WOSP/SIPEW*.
[12] Ahmed Osama Fathy Atya, Zhiyun Qian, Srikanth V Krishnamurthy, Thomas La Porta, Patrick McDaniel, and Lisa Marvel. 2017. Malicious co-residency on the cloud: Attacks and defense. In *Proceedings of INFOCOM*. IEEE, 1–9.
[13] Ronald Newbold Bracewell and Ronald N Bracewell. 1986. *The Fourier transform and its applications*. Vol. 31999. McGraw-Hill New York.
[14] Liuhua Chen, Haiying Shen, and Stephen Platt. 2016. Cache contention aware Virtual Machine placement and migration in cloud datacenters. In *Prof. of ICNP*. IEEE, 1–10.
[15] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A Patterson, and Krste Asanovic. 2013. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *Proceedings of ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 308–319.
[16] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of ACM SIGPLAN Notices*, Vol. 48. ACM, 77–88.
[17] Crispin Gardiner. 2009. *Stochastic methods*. Vol. 4. springer Berlin.
[18] Fredric J Harris. 1978. On the use of windows for harmonic analysis with the discrete Fourier transform. *Proc. IEEE* 66, 1 (1978), 51–83.
[19] Wassily Hoeffding. 1963. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association* 58, 301 (1963), 13–30.
[20] CAT Intel. 2015. Improving Real-Time Performance by Utilizing Cache Allocation Technology. *Intel Corporation, April* (2015).
[21] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. 2012. Workload characterization and prediction in the cloud: A multiple time series approach. In *2012 IEEE Network Operations and Management Symposium*. IEEE, 1287–1294.
[22] G. Khanna, K. Beaty, G. Kar, and A. Kochut. 2009. Application performance management in virtualized server environments.. In *Proceedings of NOMS*.
[23] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud.. In *Proceedings of USENIX Security symposium*. 189–204.
[24] I Lawrence and Kuei Lin. 1989. A concordance correlation coefficient to evaluate reproducibility. *Biometrics* (1989), 255–268.
[25] Zhuozhao Li and Haiying Shen. 2015. Designing a hybrid scale-up/out hadoop architecture based on performance measurements for high application performance. In *2015 44th International Conference on Parallel Processing*. IEEE, 21–30.
[26] Zhuozhao Li and Haiying Shen. 2016. Performance measurement on scale-up and scale-out hadoop with remote and local file systems. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 456–463.
[27] Zhuozhao Li and Haiying Shen. 2017. Measuring scale-up and scale-out hadoop with remote and local file systems and selecting the best platform. *IEEE Transactions on Parallel and Distributed Systems* 28, 11 (2017), 3201–3214.
[28] Zhuozhao Li, Haiying Shen, Jeffrey Denton, and Walter Ligon. 2016. Comparing application performance on HPC-based Hadoop platforms with local storage and dedicated storage. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 233–242.
[29] Zhuozhao Li, Haiying Shen, Walter Ligon, and Jeffrey Denton. 2016. An exploration of designing a hybrid scale-up/out hadoop architecture based on performance measurements. *IEEE Transactions on Parallel and Distributed Systems* 28, 2 (2016), 386–400.
[30] Zhuozhao Li, Haiying Shen, and Cole Miles. 2018. PageRankVM: A pagerank based algorithm with anti-collocation constraints for virtual machine placement in cloud datacenters. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 634–644.
[31] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *Proceedings of IEEE S&P*. 605–622.
[32] L Mandel and E Wolf. 1976. Spectral coherence and the concept of cross-spectral purity. *JOSA* 66, 6 (1976), 529–535.
[33] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.
[34] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 165–178.
[35] 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 *Proceedings of CCS*. ACM, 199–212.
[36] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 815–823.
[37] A. Singh, M. R. Korupolu, and D. Mohapatra. 2008. Server-storage virtualization: integration and load balancing in data centers.. In *Proceedings of SC*.
[38] M. Tarighi, S. A. Motamedi, and S. Sharifian. 2010. A new model for virtual machine migration in virtualized cluster server based on Fuzzy Decision Making. *CoRR* (2010).
[39] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael M Swift. 2015. A Placement Vulnerability Study in Multi-Tenant Public Clouds.. In *Proceedings of USENIX Security Symposium*. 913–928.
[40] Michail Vlachos, Philip Yu, and Vittorio Castelli. 2005. On periodicity detection and structural periodic similarity. In *Proceedings of the SIAM International Conference on Data Mining*. SIAM, 449–460.
[41] Lloyd Welch. 1974. Lower bounds on the maximum cross correlation of signals (Corresp.). *IEEE Transactions on Information theory* 20, 3 (1974), 397–399.
[42] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. 2007. Black-box and Gray-box Strategies for Virtual Machine Migration.. In *Proceedings of NSDI*.
[43] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. 2009. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks* (2009).
[44] JP Wu and Shuony Wei. 1989. *Time series analysis*. Hunan Science and Technology Press, ChangSha.
[45] 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 *Proceedings of CCS workshop*. ACM, 29–40.
[46] Zhang Xu, Haining Wang, and Zhenyu Wu. 2015. A Measurement Study on Co-residence Threat inside the Cloud.. In *Proceedings of USENIX Security Symposium*. 929–944.
[47] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers. In *Proceedings of ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 607–618.
[48] Tianwei Zhang and Ruby B Lee. 2017. Host-Based DoS Attacks and Defense in the Cloud. In *Proceedings of the Hardware and Architectural Support for Security and Privacy*. ACM.
[49] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. 2017. DoS Attacks on Your Memory in Cloud. In *Proceedings of AsiaCCS*. ACM, 253–265.
[50] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *Proceedings of CCS*. ACM, 305–316.
[51] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2014. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of CCS*. ACM, 990–1003.
[52] Yunqi Zhang, Michael A Laurenzano, Jason Mars, and Lingjia Tang. 2014. Smite: Precise QoS prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 406–418.
[53] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of ACM Sigplan Notices*.