A Study on the Impact of Memory DoS Attacks on Cloud Applications and Exploring Real-Time Detection Schemes

Zhuozhao Li, Tanmoy Sen, *Graduate Student Member, IEEE*,
Haiying Shen, *Senior Member, IEEE*, *Member, ACM*, and Mooi Choo Chuah, *Fellow, IEEE*

***Abstract*— Even though memory 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 and 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. Upon SDS, we further leverage deep neural network (DNN) techniques to design a DNN-based detection scheme that is general to various types of applications and more robust to adaptive attack scenarios. Our evaluation results show that SDS/B, SDS/P and DNN outperform the state-of-the-art detection scheme, e.g., with 65% higher specificity, 40% shorter detection delay, and 7% less performance overhead. We also discuss how to use SDS and DNN-based detection schemes under different situations.**

***Index Terms*— Memory DoS attack, attack detection, cloud computing.**

I. Introduction

COMMERCIAL cloud providers (e.g., Amazon [12] and Google [19]) provide elastic Infrastructure-as-a-Service (IaaS) for tenants to deploy applications. To maximize the resource utilization, cloud providers use the virtualization techniques (e.g., hypervisors [13], [40], [45]) 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 [51], most of the underlying hardware memory resources of a PM are still shared by its VMs from different tenants.

A malicious tenant can exploit the multi-tenancy feature in the cloud to launch *memory Denial-of-Service (DoS) attacks* [50], [51], which can cause severe resource contention on the shared memory resources. There have been many reports about the impacts of *network-based DoS attacks*. For example, the cloud-based gaming services of Xbox Live and Playstation were took down on Christmas day in 2015 [10]. Amazon EC2 cloud servers suffered from the largest DoS attack ever [4]. These DoS attacks caused heavy short-term effects (e.g, service downtime), as well as long-term effects such as business losses and losses of consumers on the victims. Similar to the network-based DoS attacks, memory DoS attacks may slow down the victim services and cause severe "economic loss" on the targeted enterprises. Recent research has shown that the memory DoS attacks can be as severe as the network-base DoS attacks. For example, results in [50], [51] show that it is practical to launch memory DoS attacks and the attacks can cause severe performance degradation of distributed applications (i.e., Hadoop MapReduce) up to 3.7 times, and 38 times increase in the response time of an e-commerce website.

So far, there are two types of memory DoS attacks: i) atomic bus locking attack that keeps sending bus locking signals to prevent other VMs from using the memory buses, and ii) 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 a victim 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 [36], [41], [48].

In spite of the severity of memory DoS attacks, existing solutions that partition memory resources among VMs to enhance the performance isolation [11], [17], [49], [54],

Manuscript received 26 February 2021; revised 26 September 2021; accepted 14 January 2022; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor K. Chen. Date of publication 9 February 2022; date of current version 18 August 2022. This work was supported in part by U.S. NSF under Grant NSF-1827674, Grant CCF-1822965, and Grant OAC-1724845; and in part by the Microsoft Research Faculty Fellowship under Grant 8300751. The conference version of this paper was published in ICPP 2020 [29] [DOI: https://doi.org/10.1145/3404397.3404465]. *(Corresponding author: Haiying Shen.)*

Zhuozhao Li is with the Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: lizz@sustech.edu.cn).

Tanmoy Sen and Haiying Shen are with the Department of Computer Science, University of Virginia, Charlottesville, VA 22903 USA (e-mail: ts5xm@virginia.edu; hs6ms@virginia.edu).

Mooi Choo Chuah is with the Department of Computer Science and Engineering, Lehigh University, Bethlehem, PA 18015 USA (e-mail: chuah@cse.lehigh.edu).

Digital Object Identifier 10.1109/TNET.2022.3144895

[55] are not efficient because they either waste the memory resources or cannot defeat all types of memory DoS attacks. Some studies [11], [55] propose to monitor the performance of applications running on the VMs in a PM and then migrate the impacted VMs to other PMs when there is resource contention [13], [38], [40], [44]. However, VM migration is not sufficient to handle memory DoS attacks because a malicious tenant can easily co-locate with the VMs of the targeted services again [36], [41], [48].

Zhang *et al.* [51] 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 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 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 detection schemes, which are *robust* to different applications, *responsive* to the attacks, and *lightweight* (little performance overhead).

In this paper, 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 detect whether there is an attack.

Although SDS can provide high detection accuracy, it requires domain knowledge to profile a normal boundary for every application beforehand and to tune several statistics-based parameters, which is not general to various types of applications and adaptive attack scenarios. In this extended version, with the recent success of applying advanced machine learning to other challenging decision-making domains, we investigate whether we can leverage deep neural network (DNN) to provide an alternate to address these challenges. Specifically, we exploit the long short-term memory fully convolutional network (LSTM-FCN) [39] to design a DNN-based scheme to detect different types of memory DoS attacks.

We have implemented and evaluated *SDS* that includes both SDS/B and SDS/P, and the DNN-based detection scheme (DNN) on a real server. The evaluation demonstrates that SDS and DNN outperform previous detection scheme in [51] by up to 2% higher recall, up to 65% higher specificity, up to 40% shorter detection delay, and up to 7% less performance overhead. In addition, the evaluation shows that SDS has slightly higher detection accuracy than DNN, when the attack scenario is relatively simple. However, when dealing with adaptive attack scenario, DNN is more robust and outperforms SDS.

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 designed a DNN-based detection scheme to detect memory DoS attacks.
- We have implemented SDS and the DNN-based detection schemes on a real server and demonstrated the effectiveness of them in terms of detection accuracy, detection delay and incurred performance overhead on applications running on co-located VMs.

The rest of the paper is organized as follows. Section II presents background and related works. Section III describes our measurement study. Section IV and Section V describe the design of statistics-based and DNN-based methods. Section VI presents the experiment evaluation. Section VII discusses the broader impact of this work and when to use SDS and DNN-based detection schemes under different situations. Section VIII concludes this paper with remarks on our future work.

II. Background

A. Shared Hardware Memory Resources

We briefly introduce the shared hardware memory resources in clouds. Take the commonly used Intel processors [6] as an example. In current datacenters, 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.

B. Memory DoS Attacks

There are two types of memory DoS attacks [51].

Atomic Bus Locking Attack: In modern processors, several atomic operations temporally lock all the internal memory buses in the socket to guarantee atomicity [1], [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.

C. Related Work

VM Co-Location: Ristenpart *et al.* [36] first identified the threat of VM co-location in the cloud, which enables the malicious tenant to conduct all LLC-based attacks including memory DoS attacks. Although the cloud providers had improved their cloud management schemes since then, the works in [14], [41], [48], [51] 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], [52], [53], or to transfer information using cache operations [36], [47] 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 [13], [30], [38], [40], [44] 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 [14], [41], [48], [51].

Performance Isolation: Many studies [11], [16], [17], [22], [25], [31], [49], [54], [55] focused on enhancing the 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: To the best of our knowledge, only one previous work [51] proposed a scheme to detect memory DoS attacks. They used the two-sample Kolmogorov-Smirnov (KS) test [34] 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 [51], SDS uses low complexity statistical-based methods to provide accurate, responsive and lightweight detection of memory DoS attacks.

III. 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. Currently, there are many types of applications running in the cloud, including database, machine learning, deep learning, data-intensive, web search, etc. Thus, we select some representative applications in different categories to study the impacts of memory DoS attacks on them.

A. 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 [3] queries (Aggregation, Join, and Scan) performing typical OLAP transactions described in [35]. 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 [2]. Its input data is generated by the Hadoop TeraGen program.

Web Search Application: We select a 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 [37]. The input data is the face recognition training dataset provided by Microsoft [7].

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 the bus locking attack, we measure AccessNum because such an attack prevents the victim VM from using the memory buses to access memory.

The number of LLC misses every T_{PCM} time (denoted as *MissNum*). For the LLC cleansing attack, we measure MissNum because such an attack frequently evicts the data

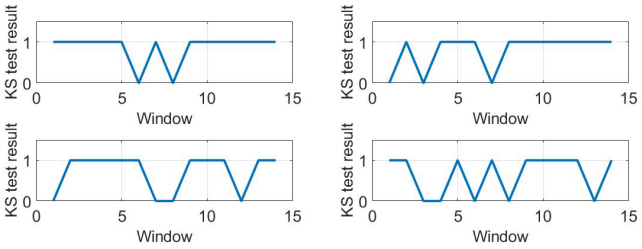


Fig. 1. KStest results of TeraSort – no attack launched.

of the 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.

B. Insufficiencies of KStest

In [51], 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.

In [51], 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 [51], 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.

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, and 55% in FaceNet when the attack is absent.

C. 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 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*.

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

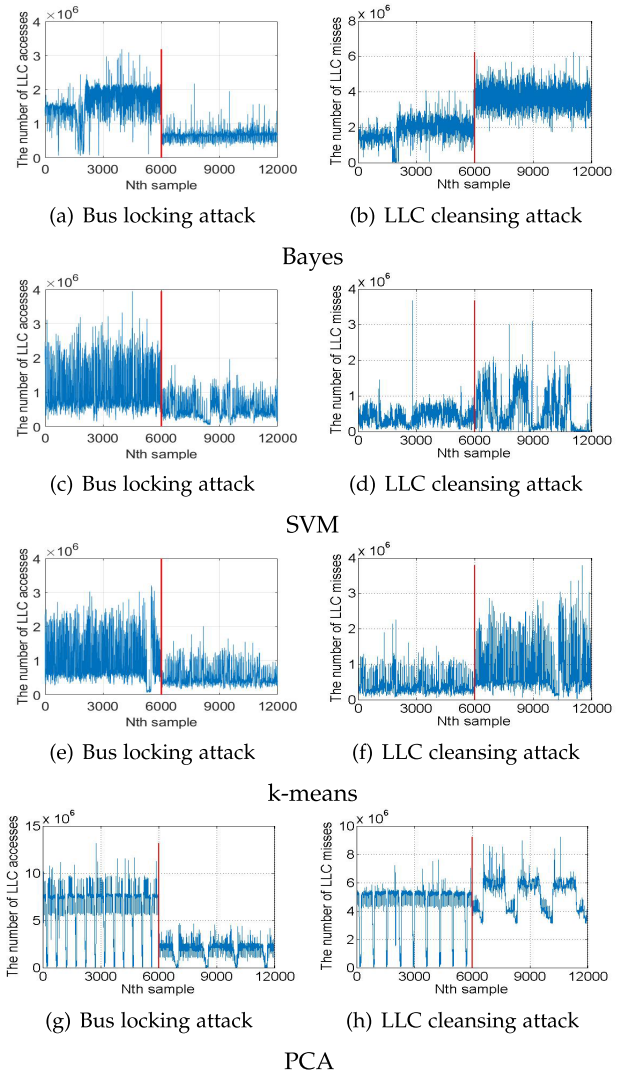


Fig. 2. Machine learning applications.

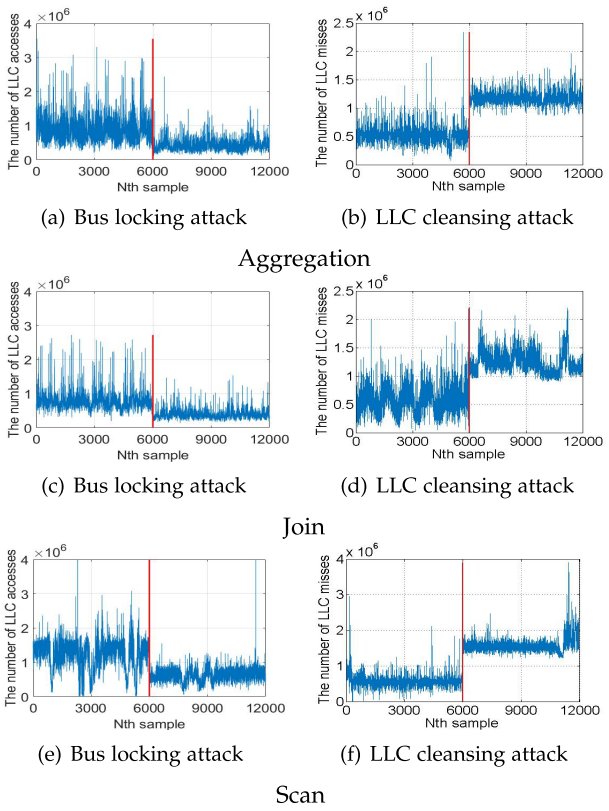


Fig. 3. Database applications.

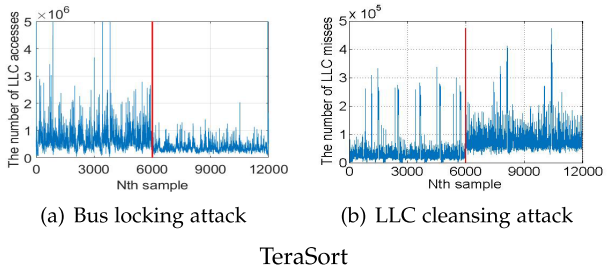


Fig. 4. Data-intensive applications.

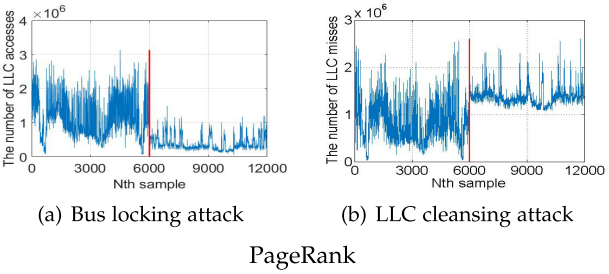


Fig. 5. Web search applications.

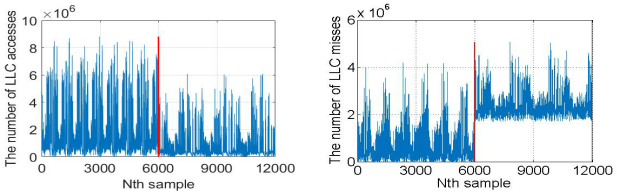
phenomenon occurs because the cleansing attack continuously cleanses the LLC, which increases the MissNum. In addition, the period of the periodic applications also increases (shown in 2(h) and 6(b)) after the LLC cleansing attack starts since more time is needed to process the same batches of input data.

Exploration of Viable Solutions: In order 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 [33], cross-correlation [43] and Pearson correlation [28] approaches on many applications including Bayes, SVM, k-means, PCA, Aggregation, Join, Scan, TeraSort, PageRank and FaceNet. Unfortunately, we have not found a viable scheme to leverage these approaches to detect the memory DoS attacks.

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 propose SDS/P for



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

FaceNet

Fig. 6. Deep learning applications.

periodic applications that leverages the change in their periods to detect the attacks.

IV. Design of SDS

In this section, we introduce the design of SDS to detect the memory DoS attacks in the cloud.

A. Data Preprocessing

To detect the attacks, the cloud providers run the PCM tool on the hypervisor of each server to collect raw cache-related statistics. For ease of presentation, we use $\{A_1, A_2, \ldots\}$ 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) [46]. 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 $\{A_{1+n*\Delta W}, A_{2+n*\Delta W}, \ldots, A_{W+n*\Delta W}\}$ in the n^{th} window, i.e.,

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

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

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.,

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

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.

B. Detection Design

In this section, we describe SDS/B for arbitrary applications, and SDS/P for periodic applications.

1) SDS/B: 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, which is based on the fact 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 and the standard deviation of the EWMA time series as μ_E and σ_E, respectively. 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

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

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 some time to co-locate her/his VM with the VM. Thus, the cloud 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

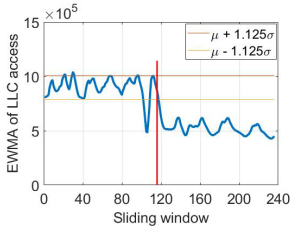


Fig. 7. Detection example of k-means.

summarize all applications. Thus, we leverage the *Chebyshev's inequality* [21] 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

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

Based on Chebyshev's inequality, one can argue that 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 the 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. Experiments in Section VI-C 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.

2) SDS/P: We propose SDS/P to leverage Observation (2) described in Section III. For periodic applications, both SDS/B and SDS/P can be used independently to detect the attacks. Our experiments in Section VI 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.

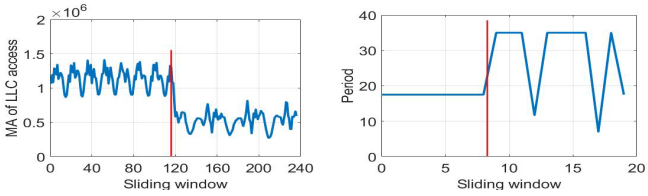
Specifically, SDS/P computes the period of MA $\mathbf{M} = \{M_0, M_1, \ldots\}$, rather than the raw data or EWMA, since MA reduces the variations of raw data as mentioned in Section IV-A which increases the detection accuracy, while the EWMA time series of a periodic application may not have periodic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

To compute the period of a time series, we could use the Discrete Fourier Transform (DFT) [15] 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 [20]. Auto Correlation Function (ACF), another method for detecting repeated patterns, can avoid false detection of frequencies of a time series [18], but may result in the detection of multiples of a true period [42]. For example, given a time series that has a period of 30 seconds, $\{60s, 90s, \ldots\}$ 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 [42] 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 [42]. First, we need to check whether an application is periodic. As in Section IV-B.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, 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



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

Fig. 8. Detection example of FaceNet application.

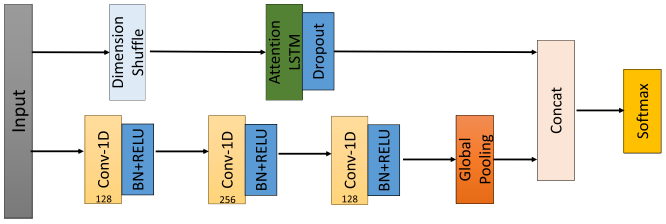


Fig. 9. Architecture of LSTM-FCN.

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 VI-C 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 IV-B.1, the detection scheme can detect the attacks as fast as $H_P \cdot \Delta W_P \cdot \Delta W \cdot T_{PCM}$ second.

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.

C. Implementation

We implement a prototype system that combines both SDS/B and SDS/P, namely SDS. In our implementation, we use PCM [9] tools to measure the real-time cache-related statistics every T_{PCM} seconds (configurable). SDS first uses the period detection method in SDS/P to check whether an application VM has periodic patterns or not. If the application VM has a periodic pattern, SDS uses both SDS/B and SDS/P to monitor the application VM to achieve a higher accuracy; otherwise, SDS leverages SDS/B for attack detection. SDS is deployed on the hypervisor on each physical machine and can be provided by the cloud providers as a service.

V. DNN-Based Detection Scheme

While we have demonstrated in the evaluation (Section VI) that SDS is highly effective in detecting memory DoS attacks, SDS falls short in two aspects. First, SDS involves tunings of several parameters based on the system requirements to balance the tradeoff between accuracy and detection delay, although these parameters are robust and it is not necessary to change the parameters frequently. Second, SDS requires to profile a normal boundary for every application based on its mean μ_E and standard deviation σ_E, and hence SDS is not general to all applications.

In this section, we aim to explore a new detection scheme that i) does not have many parameters to tune and ii) is general to all applications. With the recent success of advanced machine learning techniques to decision-making domains, we are inspired to investigate whether Deep Neural Network (DNN) can provide a viable alternative to memory DoS attack detection.

A. LSTM-Based Design

Since we would like to design a detection scheme that is general to all types of applications, the scheme needs to be capable to i) determine the type of an application from a series of cache-related statistics and ii) determine whether the statistics are normal or not. Indeed, both of these two capabilities are classification problem. We notice that Long Short Term Memory Fully Convolutional Network (LSTM-FCN) [23], [39] has been proposed to classify the time series data. Thus, we propose to exploit the LSTM-FCN (as shown in Figure 9) to design a DNN-based detection scheme for memory DoS attacks in this paper.

LSTM-FCN consists of a fully convolutional network (FCN) block and a long short-term memory (LSTM) block, as shown in Figure 9. FCN consists of three temporal convolutional blocks which are typically used as feature extractors. Then a global average pooling is applied following the final convolution block to reduce the number of parameters in the model prior to classification. The filter sizes of three stacked temporal convolutional blocks are 128, 256 and 128, respectively. For LSTM, it is used to extract temporal dependencies among input data. Specifically, the input is first conveyed into a dimension shuffle layer. The transformed time series from the dimension shuffle is then passed into the LSTM block. Each LSTM block comprises of an attention LSTM layer with 256 cells, followed by a dropout. The attention mechanism is more accurate since it allows the network to learn where to pay attention in the input sequence for each item in the output sequence, which helps to improve the accuracy of the classification for a long sequence of input. Finally, the outputs of the global pooling layer and the LSTM block are concatenated and passed onto a softmax classification layer.

We propose to use two such cascaded LSTM-FCNs to form the DNN-based detection scheme, as shown in Figure 10. The first LSTM-FCN aims to categorize the application, while the second one is responsible for detecting the type of attack. For the first LSTM-FCN, the input is a sliding window of the

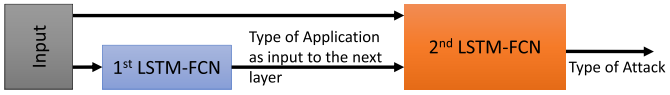


Fig. 10. Architecture of DNN-based detection scheme.

cache-related statistics (i.e., AccessNum and MissNum), and the output is the type of application. The output of the first LSTM-FCN is then sent as input to the second one to decrease the search space.

B. Implementation

To prepare the training dataset for our DNN model, we run different types of applications with attack and without attack, and collect the cache-related statistics (including AccessNum and MissNum) for them. As in Section IV-A, the data is pre-processed using the sliding window method with a window size of W and a sliding step size of ΔW. LSTM considers a window of W raw data points as a single time step and the convolution layer analyzes each data point of the time series individually throughout the window. This facilitates that fully convolutional block and LSTM block perceive the same input sliding window from two different views. The fully convolutional block views the input as a univariate time series with multiple time steps. On the other hand, LSTM considers the input as a multivariate time series and processes each window as a different time step. For example, if the input is a sliding window of length 200, the fully convolutional block receives the data in 200-time steps, while the LSTM block in the proposed architecture receives the input as a multivariate time series and processes it as a single time step.

The sample size we use for training is 20137. Similar to the setting in [23], the number of epochs is set to 3000 in this paper, and the DNN model is trained via the Adam optimizer [26], with an initial learning rate of 10^{-3} and final learning rate of 10^{-4}. The learning rate is reduced by a factor of $1/\sqrt[3]{2}$ for every 150 epochs of no improvement on validation score. We note here that we only aim to show that it is feasible to use DNN to detect the memory DoS attacks, and in practice a cloud provider can collect more training data from many applications to train its own model for each tenant. As SDS, the DNN-based detection scheme triggers an alarm after H_D consecutive anomaly windows.

VI. Performance Evaluation

In this section, we present the evaluation of the SDS and DNN-based schemes.

A. Experimental Setup

1) Environmental Setup: We ran each of our detection schemes on a server using the same hardware mentioned in Section III-B. As in [51], 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. The pre-selected victim VM ran one of the applications introduced in Section III.

TABLE I
Parameters in the Experiment

Parameter	Value
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 (SDS) 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
Consecutive violation threshold (DNN) H_D	5

TABLE II
Abbreviations of Applications

Application	Abbreviation
Bayes	BA
SVM	SVM
Kmeans	KM
PCA	PCA
TeraSort	TS
Aggregation	Aggre
Join	Join
Scan	Scan
PageRank	PR
FaceNet	FN

2) Parameters: The parameters involved in the SDS- and DNN-based schemes are shown in Table I. The parameters were selected empirically to achieve a balance between detection accuracy and detection delay.

3) Attack Scenarios and Baseline: We consider two scenarios in this evaluation.

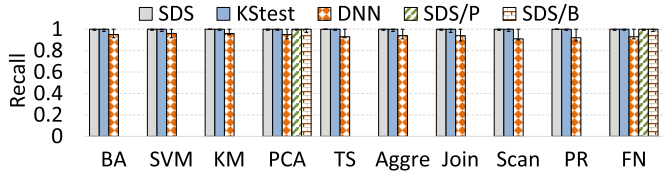
Scenario 1: We ran each benign application on the victim VM for 600 seconds. The attack VM has only two attack states: *enabled* and *disabled*. During the first 300 seconds, we did not launch any attacks on the attack VM (Stage 1). In the next 300 seconds, we performed the bus locking attack or LLC cleansing attack from the attack VM (Stage 2).

Scenario 2: Different from Scenario 1 that has only two stages, Scenario 2 consists of many stages—the attack VM keeps enabling and disabling the attacks for random durations in a round-robin manner. Specifically, we ran each benign application on the victim VM for 600 seconds, and the durations of the two attack states follow a uniform distribution in the range of $[10, 50]$ seconds. Scenario 2 is more adaptive than Scenario 1 and we aim to use Scenario 2 to simulate the case where the attack VM may attempt to use adaptive approaches to evade the detection schemes.

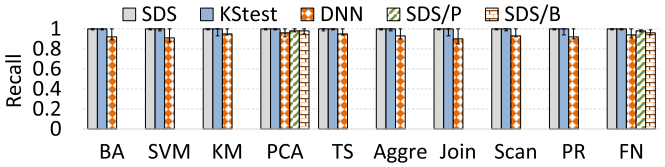
B. Evaluation Results

In this section, we evaluate the detection accuracy, detection latency, and performance overhead of SDS and DNN over various applications. Table II summarizes the abbreviations of the applications in the results.

1) Scenario 1: Detection Accuracy: To evaluate the accuracy, we will use the following metrics.

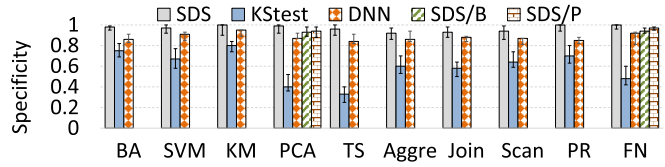


(a) Recall under bus locking attack.

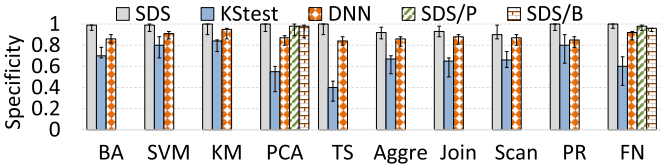


(b) Recall under LLC cleansing attack.

Fig. 11. Recall results under Scenario 1.



(a) Specificity under bus locking attack.



(b) Specificity under LLC cleansing attack.

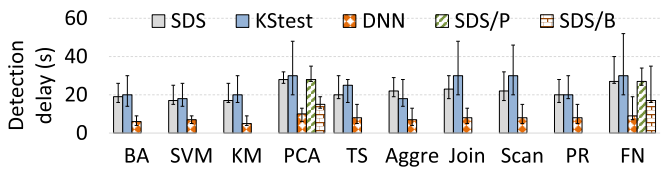
Fig. 12. Specificity results under Scenario 1.

- *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.

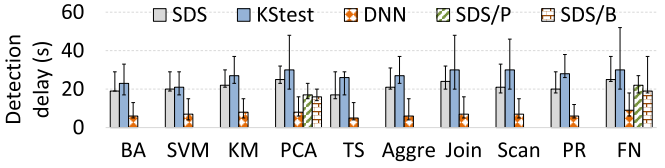
Figures 11(a) and 11(b) show the recall results of bus locking attack and LLC cleansing attack, respectively. We see that the median recalls of both SDS and KStest are 100%, regardless of the applications or the types of attacks, indicating that there are few false negatives. Based on the 10^{th} and 90^{th} percentiles, the recall of SDS is slightly better (1-2%) than KStest. The recall of DNN for all applications are around 90 - 95%, which is slightly lower than that of SDS and KStest.

Figures 12(a) and 12(b) show the specificity results of bus locking attack and LLC cleansing attack, respectively. SDS achieves a specificity around 90-100% and DNN achieves a specificity around 85-95%, while KStest only achieves a specificity around 30-80% due to many false positives. This demonstrates the outstanding performance of SDS and DNN 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



(a) Detection delay under bus locking attack.



(b) Detection delay under LLC cleansing attack.

Fig. 13. Detection delay results.

in Section III, 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 and the time when the attack is detected.

Figures 13(a) and 13(b) shows the detection delay of different applications under bus locking attack and LLC cleansing attack, respectively. For both types of attacks, the detection delays of SDS and DNN are in the range of 15-30 seconds and 5-10 seconds, respectively, while the detection delays of KStest were in the range of 20-50 seconds. The detection delays of SDS and DNN are at least 5-20 seconds shorter than KStest's. Such improvement in the detection delay is significant since previous study [8] in Amazon shows that 100 ms increase in the webpage loading time decreased sales by 1 percent. Both SDS and DNN have faster response time than KStest, because they continuously examines if the time series is anomalous in real time. On the contrary, 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 as it delays the execution of all applications, which indirectly increases the detection delay of the KStest approach.

DNN is more responsive than SDS, since DNN can extract more features from all over the time series, while SDS only creates a normal range based on the mean and standard deviation of the time series and loses much information about the time series. Thus, DNN knows more features of the time series and can response much quicker to the anomaly cache-related statistics.

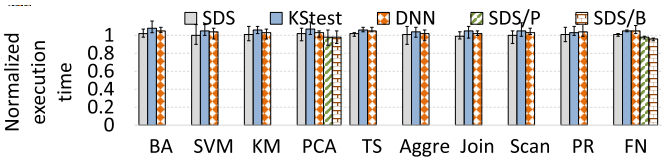
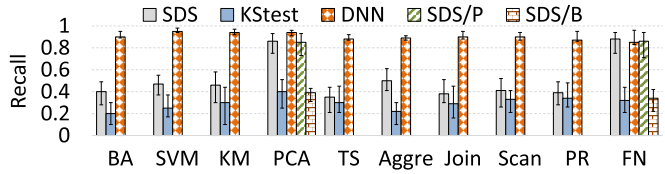
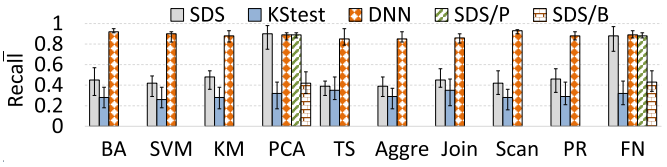


Fig. 14. Performance overhead results.

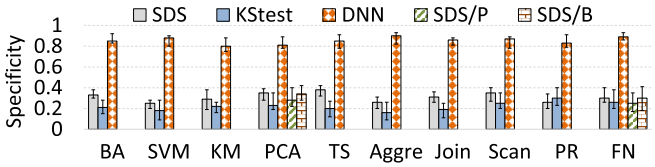


(a) Recall under bus locking attack.

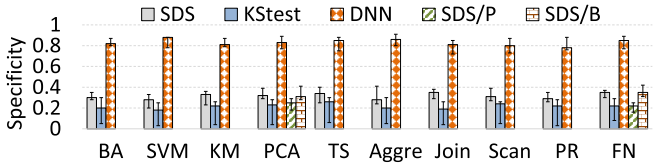


(b) Recall under LLC cleansing attack.

Fig. 15. Recall results under Scenario 2.



(a) Specificity under bus locking attack.



(b) Specificity under LLC cleansing attack.

Fig. 16. Specificity results under Scenario 2.

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 14 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 2-5% of the DNN and 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 generate negligible performance overhead. DNN results in slightly higher overhead than SDS due to the high computation cost of DNN algorithm.

2) Scenario 2: Note that we only show the results of detection accuracy in Scenario 2, since the results of performance overhead and detection delay are similar to those in Scenario 1.

Figures 15(a) and 15(b) show the recall results under Scenario 2. Figures 16(a) and 16(b) show the specificity results under Scenario 2. DNN achieves a detection accuracy around 80%-95% and performs much better than SDS and KStest in terms of both recall and specificity. This is mainly because the attacks in Scenario 2 were launched for random durations in the range of $[10, 50]$. In this adaptive attack scenario, DNN benefits from its faster response to memory DoS attacks and thus can detect most of the attacks accurately, while SDS and KStest cannot achieve so when the attacks last for relatively short durations.

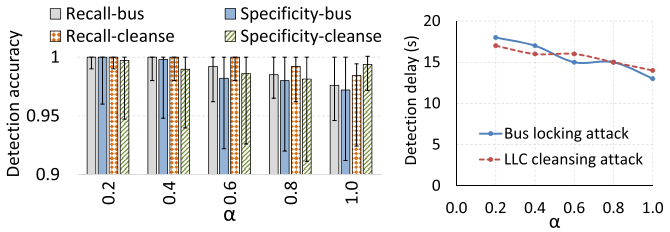
C. Sensitivity Analysis

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 I.

Due to space limitation, for the parameters regarding to the detection of periodic 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. All the experiments were conducted under Scenario 1.

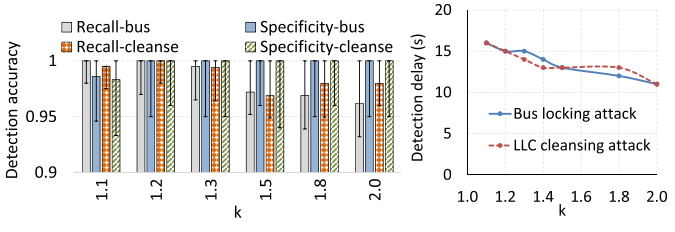
Smooth Factor α: In this experiment, we varied α in the range $[0.0, 1.0]$. Notice that when $\alpha = 1.0$, the EWMA time series is equivalent to the MA time series. Figure 17(a) shows the recall and specificity versus α. We see that as α increases, the recall and specificity decreases slightly. This 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 17(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.

Boundary Factor k: In this experiment, 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 18(a) shows the recall and specificity



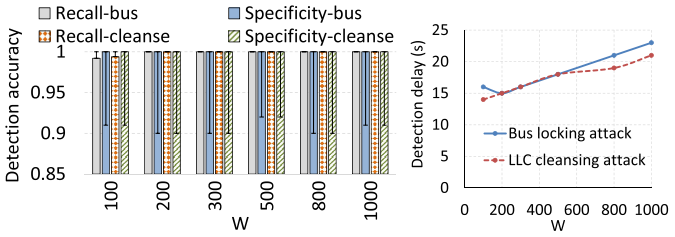
(a) Detection accuracy. (b) Detection delay.

Fig. 17. Sensitivity of α.



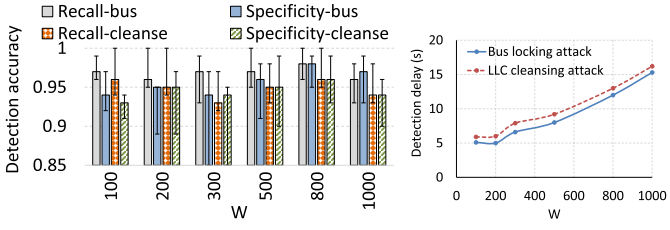
(a) Detection accuracy. (b) Detection delay.

Fig. 18. Sensitivity of k.



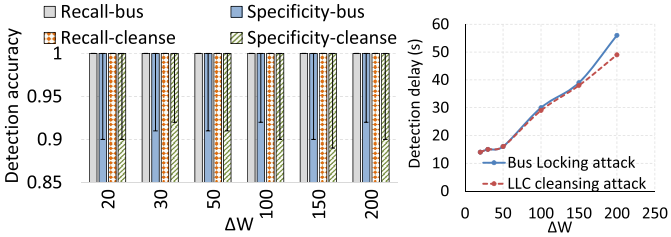
(a) Detection accuracy. (b) Detection delay.

Fig. 19. Sensitivity of W for SDS.



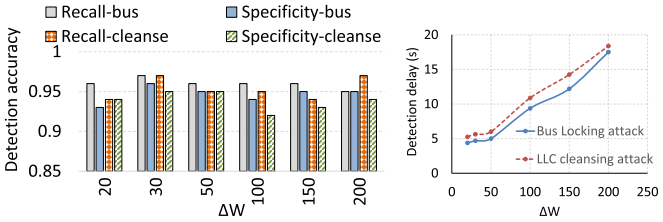
(a) Detection accuracy. (b) Detection delay.

Fig. 20. Sensitivity of W for DNN.



(a) Detection accuracy. (b) Detection delay.

Fig. 21. Sensitivity of ΔW for SDS.



(a) Detection accuracy. (b) Detection delay.

Fig. 22. Sensitivity of ΔW for DNN.

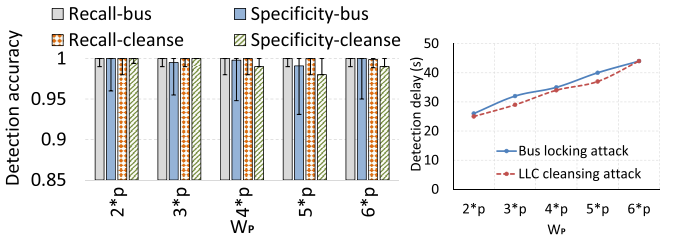
versus k. We see that the specificity increases slightly as k increases, since there are fewer false positives. On the other hand, the recall decreases slightly as k increases. This is because the detection scheme sometime fails to detect the attacks due to a larger k, which results in some false negatives. 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.

Figure 18(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.

Window Size W: In this experiment, we varied W in the range $[100, 1000]$. Figure 19(a) shows the recall and specificity versus W for the SDS detection scheme. 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 19(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.

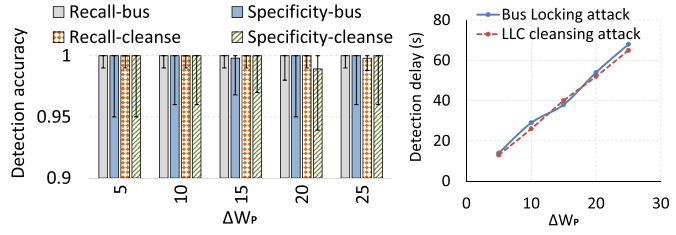
Figures 20(a) and 20(b) show the accuracy and delay versus W for the DNN detection scheme. Varying the window size W demonstrates similar impacts to DNN as SDS. First, varying W does not change the detection accuracy of DNN much. Second, the detection delay of DNN increases slightly as W increases, since it also takes longer time for DNN to detect a change on the time series as W increases. To summarize, 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.

Sliding Step Size ΔW: In this experiment, we varied ΔW in the range $[20, 200]$. Figures 21(a) and 21(b) show the accuracy and detection delay versus ΔW for SDS. Figures 22(a) and 22(b) show the accuracy and detection delay versus ΔW for DNN. We see that for both SDS and DNN, varying ΔW has similar impacts. It does not change the detection accuracy. However, the detection delay increases as ΔW increases. To summarize, we should select a small ΔW (e.g., 10-50) in practice to reduce the detection delay.



(a) Detection accuracy. (b) Detection delay.

Fig. 23. Sensitivity of W_P.



(a) Detection accuracy. (b) Detection delay.

Fig. 24. Sensitivity of ΔW_P.

Window Size W_P in SDS/P: In this experiment, we varied W_P in the range $[2p, 6p]$. Figure 23(a) shows the recall and specificity versus W_P. We observe that varying W_P does not change the detection accuracy. Figure 23(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 IV-B.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: In this experiment, we varied ΔW_P in the range $[5, 25]$. Figure 24(a) shows the recall and specificity versus ΔW_P. We observe that varying ΔW_P does not change the detection accuracy. Figure 24(b) shows the detection delay versus ΔW_P. We see that 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.

VII. Discussion

In this section, we discuss the broader impact of this work, as well as when to apply SDS and DNN under different situations.

Broader Impact: In practice, the performance of the applications running on co-located VMs may be impacted by each other *occasionally and unintentionally* [27], even though these co-located VMs are from non-malicious tenants and only run benign applications. From the tenants' perspectives, they expect cloud providers to be able to detect the interferences and to take proper actions (e.g., VM migrations) when the interferences occur to prevent severe performance degradation of their applications. The general ideas of our work can be applied here to detect the performance interferences among co-located VMs. Moreover, as container technologies (e.g., Docker and Singularity) become popular in parallel and distributed computing, we see the broader impact of our work on these areas.

When to Use SDS and DNN-Based Detection Schemes? As demonstrated in Section VI, both SDS and DNN schemes can detect the memory DoS attacks accurately and responsively, with low performance overhead. SDS and DNN have different pros and cons. When the attack scenario is relatively simple (i.e., Scenario 1), SDS has slightly higher detection accuracy than DNN. However, when dealing with the adaptive scenario (i.e., Scenario 2), DNN is more robust and can provide higher detection accuracy than SDS.

Based on these facts, we argue that it is more appropriate to use SDS to detect the performance interferences among co-located VMs under the normal scenario, where all the application VMs run normally. In this case, the tenants would expect higher accuracy and hence it is straightforward to apply SDS for the performance interference detection. On the other hand, DNN is more suitable for detecting the memory DoS attacks in practice, as DNN is more robust to the adaptive attack scenarios and can provide faster response, even the attackers adapt the attack launching patterns intentionally.

Assumptions: It is worth mentioning that our proposed detection schemes assume that we can profile the cache-related statistics of cloud applications to gather some "ground truth" beforehand. This assumption is based on the fact that many cloud applications are recurring and have predictable characteristics [24], [32].

Besides, 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 in a reasonable frequency when the tenants change their applications.

VIII. 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 i) 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, and ii) the periodic applications show prolonged periodicity for both kind of attacks. Based on the two observations, we propose two lightweight and responsive detection schemes SDS/B and SDS/P that can accurately detect the attacks. In this extended version, we exploit the LSTM-FCN to design a DNN-based detection scheme that is general to all applications and is more responsive and robust to the adaptive attack scenario. Our experiments conducted on a real server demonstrate that SDS (an implementation that includes both SDS/B and SDS/P) and DNN can detect the attacks with up to 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.

In the future, we plan to extend this work by i) exploring whether we can co-relate the resource utilizations (e.g., CPU

and memory) with the cache-related statistics to design a new detection scheme for dynamic applications; and ii) studying the memory DoS attacks in the container-based services and systems such as AWS Lambda and Kubernetes.

ACKNOWLEDGMENT

The authors would like to thank Dr. Yinqian Zhang and Dr. Tianwei Zhang for providing the source code of two memory DoS attacks and their insightful suggestions.

REFERENCES

[1] *AMD Architecture Programmer's Manual, Volume 1: Application Programming*. Accessed: Sep. 2021. [Online]. Available: http://support.amd.com/TechDocs/24592.pdf

[2] *Apache Hadoop*. Accessed: Sep. 2021. [Online]. Available: http://hadoop.apache.org/

[3] *Apache Hive*. Accessed: Sep. 2021. [Online]. Available: https://hive.apache.org/

[4] *AWS Said it Mitigated a 2.3 TBPS DDOS Attack, the Largest Ever*. Accessed: Sep. 2021. [Online]. Available: https://www.zdnet.com/article/aws-said-it-mitigated-a-2-3-tbps-ddos-attack-the-largest-ever/

[5] *HiBench Benchmark Suite*. Accessed: Sep. 2021. [Online]. Available: https://github.com/intel-hadoop/HiBench

[6] *Intel R 64 and ia-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide*. Accessed: Sep. 2021. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm

[7] *Ms-Celeb-1M: Challenge of Recognizing One Million Celebrities in the Real World*. Accessed: Sep. 2021. [Online]. Available: https://www.microsoft.com/en-us/research/project/ms-celeb-1m-challenge-recognizing-one-million-celebrities-real-world/

[8] *Online Experiments: Lessons Learned*. Accessed: Sep. 2021. [Online]. Available: http://exp-platform.com/Documents/IEEEComputer2007OnlineExperiments.pdf

[9] *Processor Counter Monitor (PCM)*. Accessed: Sep. 2021. [Online]. Available: https://github.com/opcm/pcm

[10] *Xbox Live and Playstation Attack: Christmas Ruined for Millions of Gamers*. Accessed: Sep. 2021. [Online]. Available: https://www.theguardian.com/technology/2014/dec/26/xbox-live-and-psn-attack-christmas-ruined-for-millions-of-gamers

[11] J. Ahn, C. Kim, J. Han, Y.-R. Choi, and J. Huh, "Dynamic virtual machine scheduling in clouds for architectural shared resources," in *Proc. HotCloud*, 2012, pp. 1–5.

[12] *Amazon Web Service*. Accessed: Sep. 2021. [Online]. Available: http://aws.amazon.com/

[13] E. Arzuaga and D. R. Kaeli, "Quantifying load imbalance on virtualized enterprise servers," in *Proc. WOSP/SIPEW*, 2010, pp. 235–242.

[14] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. La Porta, P. McDaniel, and L. Marvel, "Malicious co-residency on the cloud: Attacks and defense," in *Proc. INFOCOM*, 2017, pp. 1–9.

[15] R. N. Bracewell and R. N. Bracewell, *The Fourier Transform and its Applications*, vol. 31999. New York, NY, USA: McGraw-Hill, 1986.

[16] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *Proc. ACM SIGARCH Comput. Archit. News*, vol. 41, 2013, pp. 308–319.

[17] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.

[18] C. Gardiner, *Stochastic Methods*, vol. 4. Berlin, Germany: Springer, 2009.

[19] *Google Cloud Platform*. Accessed: Sep. 2021. [Online]. Available: https://cloud.google.com/compute/

[20] F. J. Harris, "On the use of windows for harmonic analysis with the discrete Fourier transform," *Proc. IEEE*, vol. 66, no. 1, pp. 51–83, Jan. 1978.

[21] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *J. Amer. Stat. Assoc.*, vol. 58, no. 301, pp. 13–30, 1963.

[22] *Improving Real-Time Performance by Utilizing Cache Allocation Technology*, Intel, Mountain View, CA, USA, 2015.

[23] F. Karim, S. Majumdar, H. Darabi, and S. Chen, "LSTM fully convolutional networks for time series classification," *IEEE Access*, vol. 6, pp. 1662–1669, 2017.

[24] A. Khan, X. Yan, S. Tao, and N. Anerousis, "Workload characterization and prediction in the cloud: A multiple time series approach," in *Proc. IEEE Netw. Oper. Manage. Symp.*, Apr. 2012, pp. 1287–1294.

[25] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud," in *Proc. USENIX Secur. Symp.*, 2012, pp. 189–204.

[26] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.

[27] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An analysis of performance interference effects in virtual environments," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Apr. 2007, pp. 200–209.

[28] I. Lawrence and K. Lin, "A concordance correlation coefficient to evaluate reproducibility," *Biometrics*, vol. 4, pp. 255–268, Mar. 1989.

[29] Z. Li, T. Sen, H. Shen, and M. C. Chuah, "Impact of memory DoS attacks on cloud applications and real-time detection schemes," in *Proc. 49th Int. Conf. Parallel Process.*, Aug. 2020, pp. 1–11.

[30] Z. Li, H. Shen, and C. Miles, "PageRankVM: A pagerank based algorithm with anti-collocation constraints for virtual machine placement in cloud datacenters," in *Proc. ICDCS*, 2018, pp. 634–644.

[31] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. IEEE S&P*, Oct. 2015, pp. 605–622.

[32] A. Mahgoub *et al.*, "Optimuscloud: Heterogeneous configuration optimization for distributed databases in the cloud," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2020, pp. 189–203.

[33] L. Mandel and E. Wolf, "Spectral coherence and the concept of cross-spectral purity," *JOSA*, vol. 66, no. 6, pp. 529–535, 1976.

[34] F. J. Massey, Jr., "The Kolmogorov-Smirnov test for goodness of fit," *J. Amer. Stat. Assoc.*, vol. 46, no. 253, pp. 68–78, 1951.

[35] A. Pavlo *et al.*, "A comparison of approaches to large-scale data analysis," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2009, pp. 165–178.

[36] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proc. CCS*, 2009, pp. 199–212.

[37] F. Schroff, D. Kalenichenko, and J. Philbin, "FaceNet: A unified embedding for face recognition and clustering," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, May 2015, pp. 815–823.

[38] A. Singh, M. R. Korupolu, and D. Mohapatra, "Server-storage virtualization: Integration and load balancing in data centers," in *Proc. SC*, 2008, pp. 1–12.

[39] N. Strodthoff and C. Strodthoff, "Detecting and interpreting myocardial infarction using fully convolutional neural networks," 2018, *arXiv:1806.07385*.

[40] M. Tarighi, S. A. Motamedi, and S. Sharifian, "A new model for virtual machine migration in virtualized cluster server based on fuzzy decision making," *CoRR*, vol. 2020, pp. 1–12, Feb. 2020.

[41] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. M. Swift, "A placement vulnerability study in multi-tenant public clouds," in *Proc. USENIX Secur. Symp.*, 2015, pp. 913–928.

[42] M. Vlachos, P. Yu, and V. Castelli, "On periodicity detection and structural periodic similarity," in *Proc. SIAM Int. Conf. Data Mining*, Apr. 2005, pp. 449–460.

[43] L. Welch, "Lower bounds on the maximum cross correlation of signals (corresp.)," *IEEE Trans. Inf. Theory*, vol. IT-20, no. 3, pp. 397–399, May 1974.

[44] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif, "Black-box and gray-box strategies for virtual machine migration," in *Proc. NSDI*, 2007, pp. 1–17.

[45] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, "Sandpiper: Black-box and gray-box resource management for virtual machines," *Comput. Netw.*, vol. 53, no. 17, pp. 2923–2938, Dec. 2009.

[46] J. Wu and S. Wei, *Time Series Analysis*. ChangSha, China: Hunan Science and Technology Press, 1989.

[47] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of l2 cache covert channels in virtualized environments," in *Proc. 3rd ACM workshop Cloud Comput. Secur. Workshop*, 2011, pp. 29–40.

[48] Z. Xu, H. Wang, and Z. Wu, "A measurement study on co-residence threat inside the cloud," in *Proc. USENIX Secur. Symp.*, 2015, pp. 929–944.

[49] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers," in *Proc. ACM SIGARCH Comput. Archit. News*, vol. 41, 2013, pp. 607–618.

[50] T. Zhang and R. B. Lee, "Host-based DoS attacks and defense in the cloud," in *Proc. Hardw. Archit. Support Secur. Privacy*, 2017, pp. 1–8.
[51] T. Zhang, Y. Zhang, and R. B. Lee, "DoS attacks on your memory in cloud," in *Proc. AsiaCCS*, 2017, pp. 253–265.
[52] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proc. CCS*, 201, pp. 305–3162.
[53] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PAAS clouds," in *Proc. CCS*, 2014, pp. 990–1003.
[54] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "Smite: Precise QoS prediction on real-system SMT processors to improve utilization in warehouse scale computers," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2014, pp. 406–418.
[55] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," *ACM SIGPLAN Notices*, vol. 45, no. 2, pp. 129–142, 2010.

Zhuozhao Li received the Ph.D. degree in computer science from the University of Virginia. He was a Post-Doctoral Scholar at The University of Chicago. He is currently an Assistant Professor with the Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology. His research interests include distributed systems, cloud computing, and high-performance computing, with an emphasis on developing working prototypes for real-world problems and designing the foundation methodologies to optimize system performance for efficient computing.

Tanmoy Sen (Graduate Student Member, IEEE) received the B.Sc. degree from the Bangladesh University of Engineering and Technology in 2014. He is currently pursuing the Ph.D. degree with the Department of Computer Science, University of Virginia. His research interests include edge/cloud computing, the IoT, and machine learning.

Haiying Shen (Senior Member, IEEE) received the B.S. degree in computer science and engineering from Tongji University, China, in 2000, and the M.S. and Ph.D. degrees in computer engineering from Wayne State University in 2004 and 2006, respectively. She is currently an Associate Professor with the Department of Computer Science, University of Virginia. Her research interests include distributed computer systems and networks, cloud computing, edge computing, distributed machine learning, and cyber-physical systems. She is a Microsoft Faculty Fellow of 2010 and a member of the ACM.

Mooi Choo Chuah (Fellow, IEEE) received the Ph.D. degree in electrical engineering from the University of California San Deigo. She is currently a Professor with the Department of Computer Science and Engineering, Lehigh University. Prior to joining Lehigh University, she was a Distinguished Member of technical staff and a Technical Manager at Lucent Bell Laboratories, Murray Hill, NJ, USA. Based on her research work at Bell Laboratories, she has been awarded 63 U.S. patents and 15 international patents related to mobility management, 3G, and next generation wireless system design. Her research interests include designing next generation networks, mobile computing, mobile healthcare, network security, and secure cyber-physical systems. She is a NAI Fellow. She has served as a Technical Co-Chair for IEEE INFOCOM in 2010 and a Symposium Co-Chair for the IEEE Globecom Next Generation Networking Symposium in 2013. She has been an Associate Editor of the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, the IEEE TRANSACTIONS ON MOBILE COMPUTING, and *Computer Networks*.