IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 16, 2021, accepted April 30, 2021, date of publication May 4, 2021, date of current version May 13, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3077498

Recurrent Neural Networks Based Online Behavioural Malware Detection Techniques for Cloud Infrastructure

JEFFREY C. KIMMELL[1], ANDREW D. MCDOLE[1], MAHMOUD ABDELSALAM[2], MAANAK GUPTA[1], (Member, IEEE), AND RAVI SANDHU[3], (Fellow, IEEE)

[1]Department of Computer Science, Tennessee Technological University, Cookeville, TN 38505, USA
[2]Department of Computer Science, Manhattan College, New York, NY 10471, USA
[3]Department of Computer Science, The University of Texas at San Antonio, San Antonio, TX 78249, USA

Corresponding author: Maanak Gupta (mgupta@tntech.edu)

This work was supported in part by the National Science Foundation under Grant 1565562, Grant 2025682, and Grant 2025686; in part by the Faculty Research Grant Program at Tennessee Technological University; and in part by the NSF CREST Center at University of Texas at San Antonio (UTSA) under Grant HRD-1736209.

ABSTRACT Several organizations are utilizing cloud technologies and resources to run a range of applications. These services help businesses save on hardware management, scalability and maintainability concerns of underlying infrastructure. Key cloud service providers (CSPs) like Amazon, Microsoft and Google offer Infrastructure as a Service (IaaS) to meet the growing demand of such enterprises. This increased utilization of cloud platforms has made it an attractive target to the attackers, thereby, making the security of cloud services a top priority for CSPs. In this respect, malware has been recognized as one of the most dangerous and destructive threats to cloud infrastructure (IaaS). In this paper, we study the effectiveness of Recurrent Neural Networks (RNNs) based deep learning techniques for detecting malware in cloud Virtual Machines (VMs). We focus on two major RNN architectures: Long Short Term Memory RNNs (LSTMs) and Bidirectional RNNs (BIDIs). These models learn the behavior of malware over time based on run-time fine-grained processes system features such as CPU, memory, and disk utilization. We evaluate our approach on a dataset of 40,680 malicious and benign samples. The process level features were collected using real malware running in an open online cloud environment with no restrictions, which is important to emulate practical cloud provider settings and also capture the true behaviour of stealth and sophisticated malware. Both our LSTM and BIDI models achieve high detection rates over 99% for different evaluation metrics. In addition, an analysis study is conducted to understand the significance of input data representations. Our results suggest that in particular cases, input ordering does have some affect on the performance of the trained RNN models.

INDEX TERMS Security, deep learning, recurrent neural network, cloud IaaS, online malware detection, long short term memory RNNs, bidirectional RNNs.

I. INTRODUCTION AND MOTIVATION

A heterogeneous cloud is a complex platform requiring substantial security infrastructure. According to the NIST [1], a cloud platform should have essential characteristics not limited to on-demand self service, broad network access, and resource pooling. These features have helped forging cloud computing into a standard for both private and public sectors. As such, many organizations are utilizing the cloud computational power for different tasks to meet growing business needs. Typically, a cloud service provider (CSP) offers Infrastructure as a Service (IaaS) where clients are allowed to 'rent' *space* in the form of virtual machines (VMs) within a data center to facilitate different operational jobs. Clients have the ability to spawn many of these virtual machines on-demand. Such a convenient way of utilizing computational resources is derived from the defined cloud essential characteristics. Recently, the amount of cloud services, in particular VMs, being offered as well as the number of clients demanding the use of these services has increased dramatically.

The associate editor coordinating the review of this manuscript and approving it for publication was Pedro R. M. Inácio.

This increase has made the cloud a very desirable target for attackers since these resources, if exploited, can be recruited to launch large scale cybersecurity attacks [2]–[5].

Cloud malware is one of the most common and growing threats where a malicious software is purposely designed to attack VMs running on a cloud IaaS. Although malware is a well researched challenge [6], [7], it's impact magnifies in cloud settings due to several underlying reasons: (i) the high demand of cloud resources usage as well as the increase in the number of clients significantly *broaden* the attack vector, (ii) several clients *lack the ability* to properly secure their acquired resources, and (iii) the *rise of automated* configuration tools (e.g., Puppet,[1] Chef,[2] etc.) further adds to the list of security vulnerabilities. If a VM is spawned using a script that contains a configuration vulnerability (a flaw in security settings, like failing to auto-encrypt files or change a default image root password) it could be left prone to attacks. Further, any VM spawned using the same script will most likely have the same weakness. This is particularly true in cases where a client is deploying a large-scale system on the cloud. For example, deploying a Web Service used by millions of users will typically include multiple web, application, and database servers, which in most cases will all be deployed using the same configuration script. The redundant use of configuration scripts across the servers that make up a web service could allow malware to easily propagate to each server in the web service. Consequentially, detecting cloud malware in a real time, online, and effective manner is an essential task for CSPs.

To address these challenges, numerous malware detection approaches have been proposed [8], [9] and are mostly categorized into *static analysis* [10], *dynamic analysis* [7], [8] and *online malware detection* [9], [11]. Static analysis works via analysing executables by code examination and creating a signature for the executable if it is flagged as a malware, whereas, dynamic analysis works by running an executable in a closed environment (e.g., sandbox) and monitoring its behavior. Online malware detection methods focus on constantly monitoring hosts by analyzing normal and malicious behaviors at all times. Static and dynamic analysis methods are well understood in literature and both have their shortcomings [10], [12]. Static approach falls short against polymorphic malware, which constantly changes its identifiable features, and zero-day malware. Such sophisticated malware can evade detection by applying packing and crypting methods to change the way it looks. Dynamic analysis can mitigate the limitations of static analysis since it is based on the behavior of the malware during execution; however, *smart* malware can detect the presence of sandboxes and cease malicious activities to avoid detection. Additionally, static and dynamic analysis share a fundamental drawback due to the fact that they aim to detect malware executables before they run on a host. This is not usually the case since malware can get into a host without passing through the static/dynamic detection system. To mitigate the aforementioned drawbacks, *online* malware detection is used by defining a set of host-wide features to capture benign and malicious behaviors.

Detecting malware in a rapid and effective manner has become a necessity. As such, researchers have utilized machine learning (ML) as a mature and reliable way for static, dynamic and online malware detection. In this paper, we introduce an approach of online cloud malware detection using deep learning (DL). In particular, we demonstrate the effectiveness of using Recurrent Neural Networks (RNNs) for online malware detection by utilizing processes system features of VMs in cloud IaaS environments. Our work is driven by the assumption that many VMs running on the cloud are automatically provisioned to do a specific task. In turn, such VMs will contain a fixed set of processes to achieve this task. Note that processes are dynamic in nature, so other unexpected processes will always be created and deleted. However, a large number of the running processes belong to the fixed set. For example, a single VM configured to host a web service will typically have web server processes (e.g, Apache), database processes (e.g. MySQL), etc. that can be represented as a sequence. Each process in this sequence is represented as a vector of the utilized system features. Towards this end, we use RNN to learn the sequence of processes running in a VM and how the presence of malware can disrupt this sequence.

We conducted an analysis for the malware samples which showed that the majority of the malware was able to change their process names to a legitimate system process. Malware was also capable of attaching itself to a legitimate process and, because of these two reasons, typical whitelisting methods are not effective, hence more sophisticated methods are needed. In our previous work [13], we used simple shallow CNN model which proved effective but with a limited detection accuracy. This was used as a baseline for our more sophisticated RNN approach.

The *main contributions* in this paper are as follows:

- We introduce a novel approach of detecting cloud malware using RNNs by utilizing processes system features. We demonstrate that the set of processes running in a VM can be represented as a sequence of system features. Further, we highlight that RNNs can effectively detect the presence of malware processes within the benign processes sequence.
- We provide a comparative analysis of Long Short Term Memory (LSTM) and Bidirectional (BIDI) models in terms of evaluation metrics, along with training and detection time.
- We provide an analysis on the effect of using different input representations. Our experiments suggest that both LSTM and BIDI models achieved high performance regardless of the order of system features, whereas, the order of processes within the input sequences impacted the performance by a range of 1-2%.

[1]Puppet. https://puppet.com/
[2]Chef. https://www.chef.io/

TABLE 1. Classification of Malware detection approaches.

Research Paper	Focus		Machine Learning Techniques				Domain	Features				
				Deep Learning Based								
	Dynamic Malware Detection	Online Malware Detection	Traditional ML	CNN	RNN	Others	Cloud-Specific	API Calls	Performance Metrics	System Calls	Performance Counters	Memory Features
Firdausi et al. 2010 [14]	✓		✓							✓		
Pirscoveanu et al. 2015 [15]	✓		✓						✓			
Luckett et al. 2016 [16]	✓					✓				✓		
Fan et al. 2016 [17]	✓		✓					✓				
Joshi et al. 2018 [18]	✓		✓						✓			
Tobiyama et al. 2016 [8]	✓			✓	✓			✓				
Pascanu et al. 2015 [19]	✓				✓			✓				
Kolosnjaji et al. 2016 [20]	✓				✓			✓				
Halim et al. 2019 [21]	✓				✓					✓		
Xiao et al. 2019 [22]	✓				✓					✓		
Xie et al. 2020 [23]	✓				✓					✓		
Mishra et al. 2019 [24]	✓				✓		✓			✓		
Demme et al. 2013 [25]		✓	✓								✓	
Ozsoy et al. 2015 [26]		✓	✓								✓	
Xu et al. 2017 [27]		✓	✓									✓
Azmandian et al. 2011 [28]		✓	✓				✓			✓		
Guan et al. 2012 [29]		✓	✓				✓		✓			
Watson et al. 2015 [2]		✓	✓				✓		✓			
Abdelsalam et al. 2017[30]		✓	✓				✓		✓			
Dawson et al. 2018 [31]		✓	✓			✓	✓			✓		
Abdelsalam et al. 2018 [13]		✓		✓			✓		✓			
McDole et al. 2020 [32]		✓		✓			✓		✓			
Fei Xiao et al. 2019 [33]		✓				✓	✓	✓				
Our Approach		✓			✓		✓		✓			

The remainder of this paper is as follows. Section II, discusses other related works regarding RNNs, malware detection, and cloud computing. Section III describes the approach and methodology to our experiments. Section IV discusses the experimental cloud set up and the results from the RNN models. Section V elaborates on the RNN sensitivity to different input representations whereas cost analysis is described in Section VI. Section VII focuses on discussion and highlights some limitations, Section VIII summarizes the findings and concludes with possible future directions.

II. RELATED WORK

Behavioral machine learning based malware detection approaches can be divided into *dynamic malware detection* and *online malware detection*. An important distinction between the two approaches is that, in dynamic malware detection, executable (malware or benign) is run in a sandbox and its behavior is captured, whereas in online malware detection, the behavior of the entire system is captured with particular times being labeled as malicious if a malware is running. In this section, we discuss some of the related dynamic and online malware detection works. Further, we sub-categorize these works based on several aspects including traditional versus deep learning based approaches and whether the work is cloud-specific, as shown in Table 1.

A. DYNAMIC MALWARE DETECTION

There have been several works on dynamic malware detection using traditional machine learning approaches. The works in [14], [16] focused on using system calls as features. Firdausi *et al.* [14] employed traditional machine learning algorithms such as KNN, Naive Bayes, decision trees and SVM, where as Lucket *et al.* [16] used neural networks. The works in [15], [18] rely on system performance metrics and traditional ML algorithms for malware detection. In addition, Fan *et al.* [17] built a framework using sequence mining techniques that effectively discover malicious patterns in malware. This work utilizes a Nearest Neighbor classifier to identify previously unknown malware.

Recently, it has become clear that more sophisticated approaches for malware detection are needed. This is mainly because of the fact that traditional ML approaches require extensive pre-processing and rigorous feature engineering and representation. As such, recent research efforts have moved towards employing end-to-end deep learning techniques to bypass the feature engineering step. Many research works [8], [19]–[24] aimed to overcome the limitations of traditional ML approaches and employed DL algorithms. The works in [21]–[24] provide malware detection methods based on system calls and RNN. Others [8], [19], [20] have also used Recurrent Neural Networks (RNN) and Convolutional Neural Network (CNN) but, instead focused on API calls.

However, dynamic analysis has some limitations due to controlled environment where the malware run. In many cases, it cannot be analyzed completely due to limited access of Internet. Sophisticated malware can detect the presence of a sandbox and immediately terminate any malicious behavior. In addition, most of the dynamic detection target traditional host-based systems and not specific to cloud infrastructures (e.g., VMs). Consequentially, the need for online malware detection approaches is necessary.

B. ONLINE MALWARE DETECTION

The advantages of online malware detection approaches are: (1) they don't rely on a closed environment, (2) they continuously monitor the VMs, as opposed to dynamic analysis approaches where once an executable is deemed benign it freely runs on the system, and (3) they consider the entire VM behavior as opposed to just an executable behavior.

The authors in [25], [26] utilize performance counters for online malware detection, whereas [27] proposed the use of memory features; however, these works used traditional ML algorithms and targeted traditional host-based systems. In order to enhance the accuracy of malware detection in cloud, more cloud-specific techniques are proposed. Guan *et al.* [29] proposed an anomaly detection for VMs in cloud environment using system calls. They used an ensemble of Bayesian predictors and decision trees. Similarly, Azmandian *et al.* [28] proposed an intrusion detection system using system calls and used traditional ML algorithms including KNN and clustering. Further, Dawson *et al.* [31] used API calls captured through the hypervisor and used a non linear phase-space algorithm to detect anomalous behavior.

Other works have focused on using features that can only be fetched through the hypervisor. Given that many experimental setups are run within the context of a hypervisor, it is common to see features collected from the hypervisor. Also, such techniques are suitable to be implemented by the CSP since they do not require inside visibility to the VMs. Watson *et al.* [2] utilized performance metrics that can be fetched from the hypervisor in order to detect malware. This paper utilized a one class SVM for malware detection; however, they focused on malware that is known-to-be as highly-active malware. Similarly, Abdelsalam *et al.* [30] demonstrated a black box based approach to detect malware. This work uses VM-level system and resource utilization features. This worked well in detecting highly active malware with high resource utilization features but was not as effective in detecting malware that hide itself with low utilization.

Beside the works that used traditional ML algorithms, others [9], [13], [32], [33] focused on using deep learning algorithms for online malware detection. The authors in [13] extended their work in [30] and introduced a detection method which uses a CNN model with the goal of identifying low profile malware. This method achieved ≈90% accuracy using resource utilization features and was able to identify multiple low-profile malware since it focused on per-process level performance metrics. One limitation of this work is that the authors used a shallow CNN model and didn't provide an analysis on using various CNN models. In this regards, McDole *et al.* [32] provided a baseline analysis of using state-of-the-art CNN models including multiple ResNet [34] and DenseNet [35] models. We extend this work by providing an analysis on employing RNN.

In this paper, we primarily focus on online malware detection using RNN in cloud infrastructures. To the best of our knowledge, this is the first work that uses *RNN* based malware detection approach using *performance metrics* in *online cloud* environment. We provide a novel way of representing a VM's behavior as a sequence of processes performance metrics as discussed in the next section. Additionally, our work provides an insightful analysis on the RNN sensitivity to different input representations for malware detection.

III. METHODOLOGY

In this section, we explain the methodology used for malware detection in VMs in cloud infrastructure.

A. LSTM MODELS

RNN is a category of deep learning that can process sequential information such as language translation [36], speech recognition [37], and time series prediction [38]. However, it suffers from two problems. First, RNN struggles with short term memory; this means that long inputs can cause the RNN model to forget earlier information. Second, RNN models are subject to vanishing gradients. This is where the gradient value becomes diminished as the model backpropagates, which leads to the model not learning properly. LSTM was created to resolve these problems [39]. LSTM units contain *input*, *forget*, and *output* gates which control how the information flows into and out of the cell. This allows them to preserve important information and discard any unnecessary data. As shown in Figure 1, LSTM contains `sigmoid` and `tanh` activations. `Sigmoid` activations force inputs to a value between 0 and 1. This is where information is either retained or discarded. The closer the value is to 0, the less important it is. The `tanh` activations keep values between -1 and 1 to ensure that the output is regulated.

All of these gates help LSTM layers create a reliable model that can leverage all of our sequential data without the worry of losing data or having inaccurate gradients. The

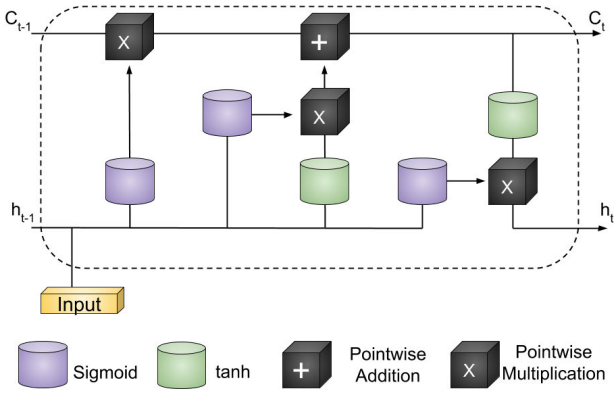

FIGURE 1. Architecture of a LSTM.

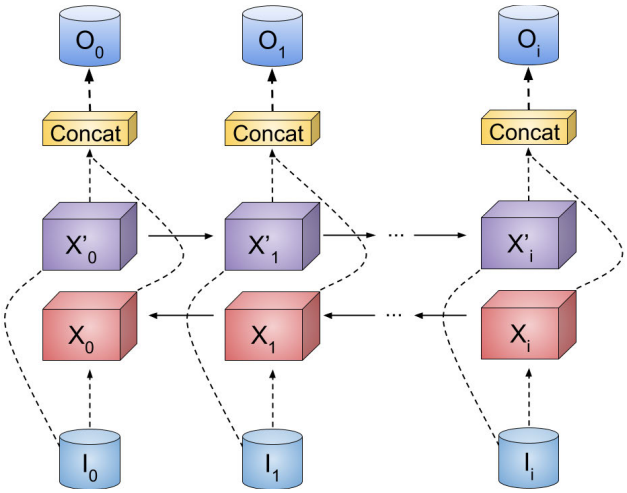

FIGURE 2. Architecture of a bidirectional (BIDI) layer.

first step in the LSTM unit is the *forget* gate. Data from the previous hidden state (h_{t-1}) and data from the current input is pushed through the `sigmoid` function, which as stated earlier, forces these inputs to be changed to a value between 0 and 1. This is where the LSTM decides what information to keep or discard. The next gate is the *input* gate; here, data from the previous hidden state and the current input are once again passed through the `sigmoid` function, but this time they are also passed through a tanh function which regulates the output by forcing the values between -1 and 1. The outputs from the `tanh` function and the `sigmoid` function are combined by using pointwise multiplication. Now we have output from the forget gate and the input gate, so we can now calculate the cell state to be passed on. This is done by using pointwise multiplication on the previous cell state and the forget gate. Pointwise addition is then used to add the output from the input gate to the value obtained from the previous step. Finally we have the *output* gate. This is where data from the previous hidden state and the current input are once again pushed through a `sigmoid` function, then the current cell state is pushed through a `tanh` function. Pointwise multiplication is then used to multiply these numbers to produce the new hidden state. The new hidden state (h_t) and the new cell state (c_t) is what is passed to the next LSTM unit.

B. BIDIRECTIONAL MODELS

Bidirectional LSTM models are able to process input in a forward and backwards manner [40]. Instead of the layer only processing the input normally by using one LSTM layer, past to future, another LSTM layer is added that processes the input starting at the last object of the input and working its way backwards, i.e. future to past. Just like in a normal LSTM layer, each bidirectional LSTM layer is assigned a number of units. This bidirectional methodology allows the model to learn more by analyzing the data from both directions and applying information from future inputs towards its predictions. Once these two layers process their respective data, the output from these layers is then concatenated together after each timestep. This type of model is useful when extra context might be needed in order to make accurate predictions. In our case, the bidirectional model can analyze future processes and use that information to determine what might be happening at a current process. This creates a model that is well suited to determine if a machine is infected with malware or not by analyzing how the machine will behave in the future. Figure 2, depicts the architecture of a bidirectional LSTM layer within an RNN model.

C. SYSTEM FEATURES

The system features in Table 2 are the features used to define processes behavior. The values are an example of the raw data collected about a single process taken at a certain time. Further processing of the data is required such as encoding the strings using one-hot-encoding, and the data must be flattened to a 1-dimensional vector before it can be used in an RNN. Most of these features can be obtained by using Virtual Machine Introspection (VMI) tools such as LibVMI[3] to capture snapshots of VMs memory and, in turn, extract the required information by using memory forensics tools such as Volatility.[4] This set of system features are intended for the sole purpose of demonstrating the validity of our approach, but more features can further enhance the accuracy.

D. UNIQUE PROCESSES AND RNN INPUT

System features are collected from all processes running in a VM at certain time. With many short lived processes (i.e. being created and destroyed quickly within each VM) as well as having their IDs reassigned by the operating system, it can be misleading and difficult to learn their behavior. As such, we define "unique processes" (as introduced in [13]) to reduce such dynamism. Unlike traditional operating system process which is identified by a "pid", a unique process is more concerned about the behavior of a process and is identified by a tuple of two elements *process name* and *the*

[3]LibVMI. http://libvmi.com/
[4]Volatility. https://www.volatilityfoundation.org/

TABLE 2. System features sample.

Metric	Value	Metric	Value	Metric	Value
mem_swap	0	kb_received	0	mem_lib	0
kb_sent	0	mem_text	217088	num_threads	1
mem_uss	1105920	ionice_ioclass	0	ionice_value	0
io_write_bytes	0	gid_effective	111	mem_shared	3334144
io_write_chars	76	num_fds	14	mem_data	585728
io_write_count	9	cpu_children_sys	0	mem_vms	43921408
io_read_bytes	958464	cpu_children_user	0	mem_rss	3751936
io_read_chars	61088	cpu_user	0.01	mem_dirty	0
io_read_count	77	cpu_sys	0	status	sleeping
ctx_switches_voluntary	0	cpu_num	0	name	dbus-daemon
ctx_switches_involuntary	43	cpu_percent	182	nice	0

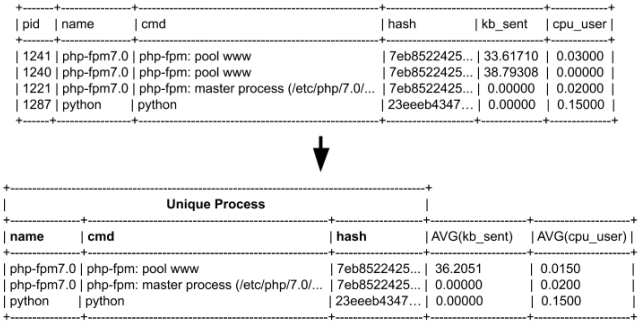

FIGURE 3. Operating system processes and unique processes.

command used to run the process. Figure 3, shows an example of operating system processes converted to unique processes. Processes sharing the same 2-tuple (e.g., forked processes) are aggregated by taking the average of their measures. This approach also helps in reducing the number of processes in a single sample.

The collected unique processes' features will be represented as data samples to be used as input to the RNN models, where each data sample is a sequence of unique processes. We represent a sample X recorded at time t collecting n features f for m unique processes up as follows:

$$X_t = up_1 \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \rightarrow up_2 \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \cdots \rightarrow up_m \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}$$

Typically, a malware infects a VM and creates one or more processes which will disrupt the benign sequence of processes. Depending on the malware, it can attach itself to another process and cease its own main process to avoid detection which may turn some existing unique processes behavior to malicious. As such, a malicious sample includes some malicious processes interspersed between the benign sequence and can be represented as follows (mp_k denotes malicious processes):

$$X_t = up_1 \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \rightarrow mp_1 \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \cdots \rightarrow up_m \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \rightarrow mp_k \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix}$$

A malware process can hide within the large number of running processes by renaming its process to some commonly used names. However, using the concept of unique process makes it harder for the malware process to hide because the number of unique processes is substantially smaller. Further, a malware process will be more visible since it will be considered a unique process. Our aim is to learn from the sequence of processes (including benign processes that a malware attached to) in a given sample and to identify it as malicious or benign.

IV. EXPERIMENTAL SETUP AND RESULTS

A. LSTM AND BIDI MODELS ARCHITECTURE

Our first model is based on LSTM and consists of eight layers. The first three LSTM layers consist of 256, 128, and 64 units, respectively. Each of our LSTM layers is followed by a dropout layer of 10% in order to prevent over fitting. The final layer is an output layer with softmax activation. Since we are using binary classification (i.e. malicious or benign) we only need two output units. Our second RNN model is bidirectional LSTM. This model consists of four bidirectional LSTM layers. The four layers are comprised of 512, 256, 128 neurons, and 64 neurons, respectively. Each of these layers is followed by a dropout layer of 10%. The output layer for this model consists of two output units and uses a softmax activation. Both of these architectures were chosen due to their simplicity which allows for faster training times. Despite the models' simplicity, they are still able to perform at a high level.

These models are trained, validated and tested with a data set that consists of 113 experiments, split by 60% for training, 20% for validation, and 20% for testing. To obtain optimal models, a grid search method was used for hyperparameters optimization, mostly, with respect to batch sizes (16, 32, and 64) and learning rates (.0001, .00001, .000001).

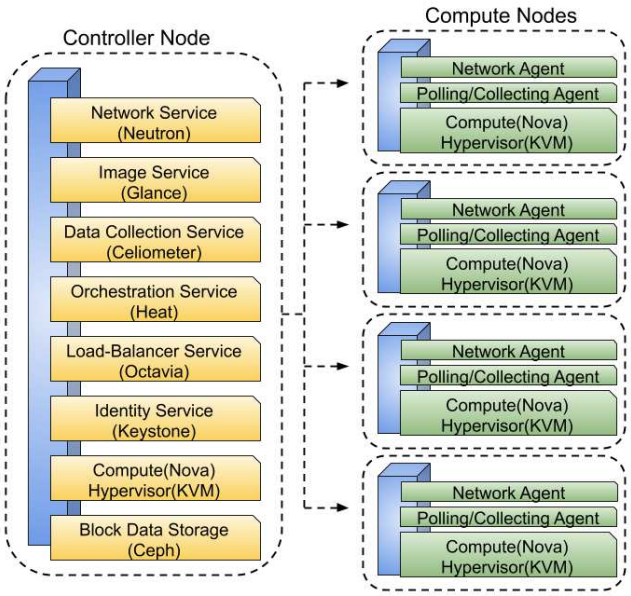

FIGURE 4. Experimental openstack cloud testbed.

B. EXPERIMENTAL SETUP

1) CLOUD TESTBED

Getting accurate system features from the malware experiments is imperative for showcasing near real world performance. To accomplish this, a cloud testbed running an actual application was used and multiple measures were taken to ensure that the malware shows its true behavior. As shown in Figure 4, the cloud testbed utilized OpenStack,[5] a popular open source cloud platform and consists of one control node and four compute nodes. The control node handles tasks such as the dashboard, storage, network, identity, and computing. The compute nodes only handle computing services. Each compute node is also supplied with agents for networking, polling, and collecting.

To avoid hindering the malware and allow it to exhibit its true malicious behavior, all of our experiments were conducted in the wild where all the VMs were connected to the Internet. This is because (i) sophisticated malware typically has the ability to detect the presence of a closed restricted environment (e.g., sandbox) and (ii) many malware, which are controlled by a command and control server (C&C), cease malicious activities upon failing to communicate with its C&C. Also, all antivirus tools and firewalls were disabled.

2) MALWARE SAMPLES

In total, 113 linux malware executable were obtained from VirusTotal.[6] To avoid biased results towards certain malware families, the malware was chosen randomly from various categories (according to VirusTotal) including DoS, DDoS, Backdoor, Trojan, Virus, Worm, among others.

[5]OpenStack. https://www.openstack.org/
[6]https://www.virustotal.com/

3) EXPERIMENTS DEPLOYMENT

Figure 5 shows an overview of the experiments deployment. The upper dotted box depicts the deployment of a single experiment stack. To simulate a real world scenario, a commonly used 3-tier web architecture, consisting of webservers, application servers, and a database, was deployed. A front load balancer is deployed to handle and distribute clients requests to appropriate web servers. Web servers are connected to application servers via an internal load balancer to distributed the requests among the application servers. For simplicity, application servers are all connected to a single powerful database server. Further, an auto-scaling policy was implemented based on CPU usage. The same scalability policy is applied to both web and application servers independently. If the average CPU utilization of all VMs belonging to the web or application tier exceeded 70%, new VMs are spawned and attached to the corresponding load balancer to meet demand. If the CPU utilization fell below 40%, VMs are deleted to reduce resource usage. In our experiments, based on the traffic load, between 2 to 10 servers were spawned in each tier. Random GET/POST requests, denoting clients, were sent to the front load balancer using a multi process python script running on a dedicated VM. For integrity of experiments, the traffic/requests were generated based on an ON/OFF Pareto distribution. This deployment is intended to reflect the real world *dynamic* behavior of cloud infrastructures to satisfy changing tenants resource requirements.

The lower part of Figure 5 consists of a main control VM and a data collection VM. The main control VM is responsible for (i) keeping the malware executables in a database, (ii) injecting a single malware in one of the application servers at a certain point of time, and (iii) deploy/destroy an experiment stack. We utilized OpenStack Heat orchestration service to easily deploy/destroy an experiment stack using `yaml` scripts. The data collection VM is responsible for collecting data from the infected VM. The process for collecting data is shown in Figure 6. Each experiment lasted a total of 1 hour. The first 30 minutes is referred to as the benign phase, where there is no malware running. A single malware is injected at a random time between minute 30 and 40 in one of the application servers. Varying the malware injection/execution time introduces more dynamism into the experiments and ensuring that consistency of injecting malware at the same point in time would not affect the results. Starting minute 40 is referred to as malicious phase, where a malware is running. Data samples are collected every 10 seconds (resulting in 360 samples in total for each experiment) using a host based built in python script, and are stored into a database. 113 experiments were conducted, each using a different malware executable. Each of these 113 experiments generated 360 samples for each individual experiment. This results in a total of 40,680 total samples. The main control VM destroys the entire experiment stack (upper box) after each experiment to prevent data contamination of subsequent experiments/runs.

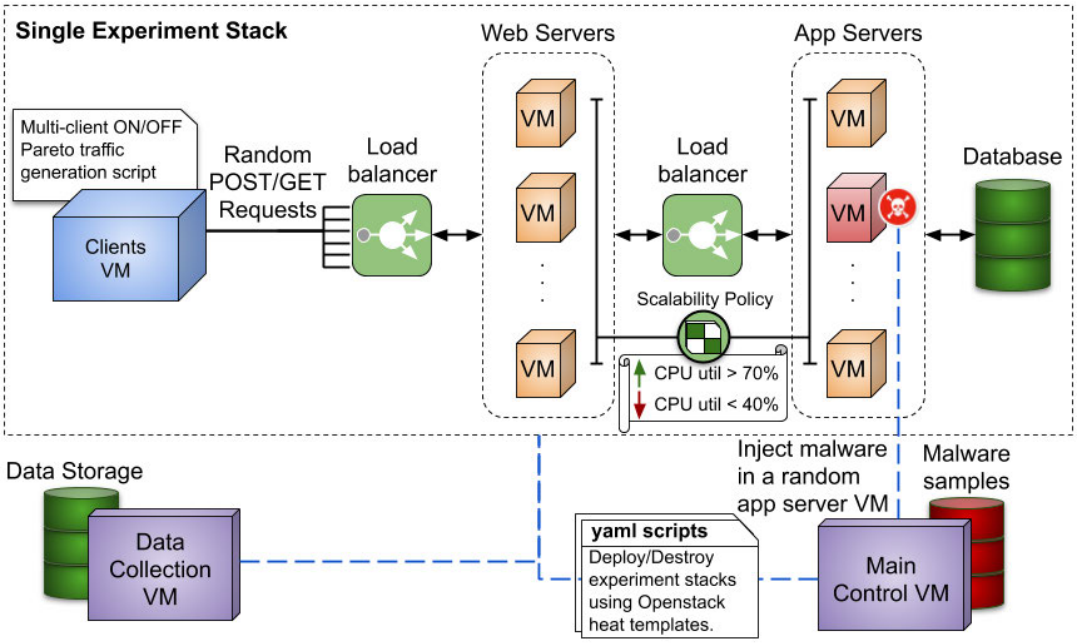

FIGURE 5. Overview of experiments deployment.

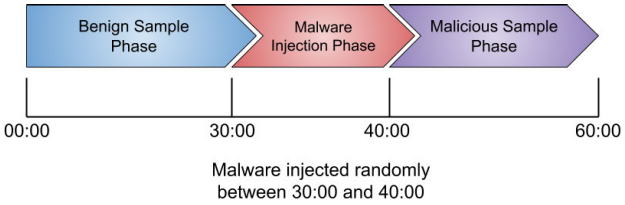

FIGURE 6. Data collection phases.

4) RNN MODELS TRAINING

All experiments resulted in 40,680 data samples collected. This is because the data we are collecting represent the behavior of all processes in the virtual machine, not just the actual malware executables. Models training was performed on a high performance computing center (HPC) with four Dell PowerEdge R730 servers, each with one NVIDIA Tesla K80 GPU. The RNN models were built and tested by Python scripts using Keras[7] API which is built on top of Tensorflow.[8]

5) RNN INPUT

As stated in Section III-D, the input to the RNN models is a sequence of vectors, each denoting the features for a particular unique process. In our experiments, the maximum number of unique processes in any experiment is 120, hence, all sequences are padded to be of the same length. The system features (Table 2) collected for each unique process are preprocessed by converting categorical string features to one-hot vectors and standardizing the data values.

[7]https://keras.io/
[8]https://www.tensorflow.org/

C. EVALUATION

The performance of our models is measured by five evaluation metrics, accuracy, precision, recall, and F1 score.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$
$$Precision = \frac{TP}{TP + FP}$$
$$Recall = \frac{TP}{TP + FN}$$
$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

True positives (TP) is the number of correctly classified malicious samples. True negative (TN) is the number of correctly classified benign samples. False positive (FP) is the number of benign samples that were incorrectly classified as malicious. False negative (FN) is the number of malicious samples that were incorrectly classified as benign. The accuracy metric is the measure of correct classification. Precision is the measure of correct positive classifications over the total number of positive classifications. Recall is a measure of correctly classified malicious samples over the actual number of malicious samples. F1 score is the balance between precision and recall.

D. RESULTS

As stated in Section IV-A, a different malware is used in each of the 113 experiments and the dataset collected were divided into 60% training, 20% validation and 20% testing. In order to emphasize the ability of our models to detect **zero-day malware**, the dataset were split on the number of experiments

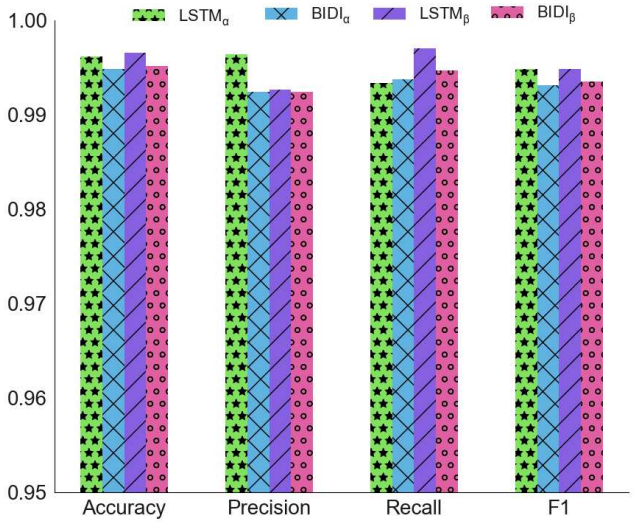

FIGURE 7. Results of LSTM and BIDI (α and β) models.

TABLE 3. Detailed results for LSTM and BIDI (α and β) models.

Model	Accuracy	Precision	Recall	F1
LSTM$_\alpha$	99.61%	99.64%	99.33%	99.48%
BIDI$_\alpha$	99.48%	99.24%	99.37%	99.31%
LSTM$_\beta$	99.65%	99.26%	99.7%	99.48%
BIDI$_\beta$	99.51%	99.2%	99.51%	99.35%

(i.e. 67 training, 23 validation and 23 testing). This ensures that the data samples collected from the 23 experiments (based on 23 unseen malware) for testing were completely unseen to the RNN models. The training dataset is used to train the RNN models, the validation dataset is used as a way to tune the hyperparameters (e.g., learning rate, batch-size, etc.) to get optimal models, and the testing dataset is used to measure the detection ability of the optimized LSTM and BIDI models.

To ensure the validity of our results, both LSTM and BIDI models were trained twice (i.e. LSTM$_\alpha$, LSTM$_\beta$, BIDI$_\alpha$ and BIDI$_\beta$). The order of input sequences to the α and β models is slightly different to introduce dynamism. All RNN models were trained for 40 epochs since there was no decrease in the loss afterwards.

Figure 7 depicts the results of our experiment where the bars shown were produced by calculating different evaluation metrics for each the LSTM and BIDI optimal models. In our case, the optimal models are identified by hyperparameters of *batchsize* $= 32$ and *learningrate* $= 1e{-}5$. Both of the α and β models were able to detect newly seen malware with high accuracy in all metrics exceeding 99% (exact numbers are given in Table 3). The α models achieved higher precision scores than the β models, whereas the β models achieved higher recall scores than the α models. Even though the difference in recall and precision scores between the α and β models is very minuscule, these results suggests that the order of processes within input sequences might affect the models learning ability (more details are discussed in Section V).

Figure 8 shows the training and validation *mean cross entropy loss* during the models' learning progression. Training loss is recorded after each iteration, whereas validation loss is recorded after each epoch. The figure shows that the models were able to properly generalize and learn from the given datasets. The red line indicates the epoch where a particular model scored the highest validation accuracy during the 40 epochs training phase.

V. RNN SENSITIVITY TO DIFFERENT INPUT REPRESENTATIONS

In Section III-D, we described how we construct the samples that are used in our experiments. Each sample consists of a sequence of unique processes. However, it is not clear whether the order of unique processes and features in a single sample would affect the RNN models' ability to learn and generalize effectively. Altering the ordering of the input data can often reveal insights as to how to best train certain models. For instance, the authors in [41] provided an analysis on the effects of input ordering when using CNN models. They used similar process system features for malware detection using CNN models and studied the effects of processes and features ordering in the input, represented as an image (denoting processes $\times$ features). The authors performed experiments with CNN models by generating different sets of the same input data with different orderings. In this study, the authors were able to enhance the accuracy of detecting malware from 90% to 98%. As such, it was concluded that certain orderings of input data can in fact improve the performance of the models and must be constructed properly.

In this section, we provide an analysis on whether the order of sequence in a single sample (denoted by *row* models) as well as the order of features (denoted by *col* models) would affect the results of the RNN models. The key intuition of this analysis lies in the fact that some unique processes might be closely related, and including them in close proximity in the input sequences might help the models to easily draw and learn such correlations. For example, consider two unique processes of the FastCGI Process Manager (FPM) *php-fpm: master* and the its forked pool of processes *php-fpm: pool* (see Figure 3). Similarly, some system features might be related. For example, features of cpu usage such as *cpu_user, cpu_sys, cpu_num, and cpu_percent* are closely related. As shown in Figure 9, different row orderings are created by randomly changing the sequence of unique processes for all samples. Similarly, different column orderings are created by randomly changing the order of the features that belong to each unique process. This increases the odds of preventing related processes or features from appearing in close positions in a given sample.

A. RANDOM ORDERING RESULTS

In our experiments, we trained four LSTM (i.e. LSTM colα, colβ, colγ, and colδ) and BIDI models (i.e. BIDI colα,

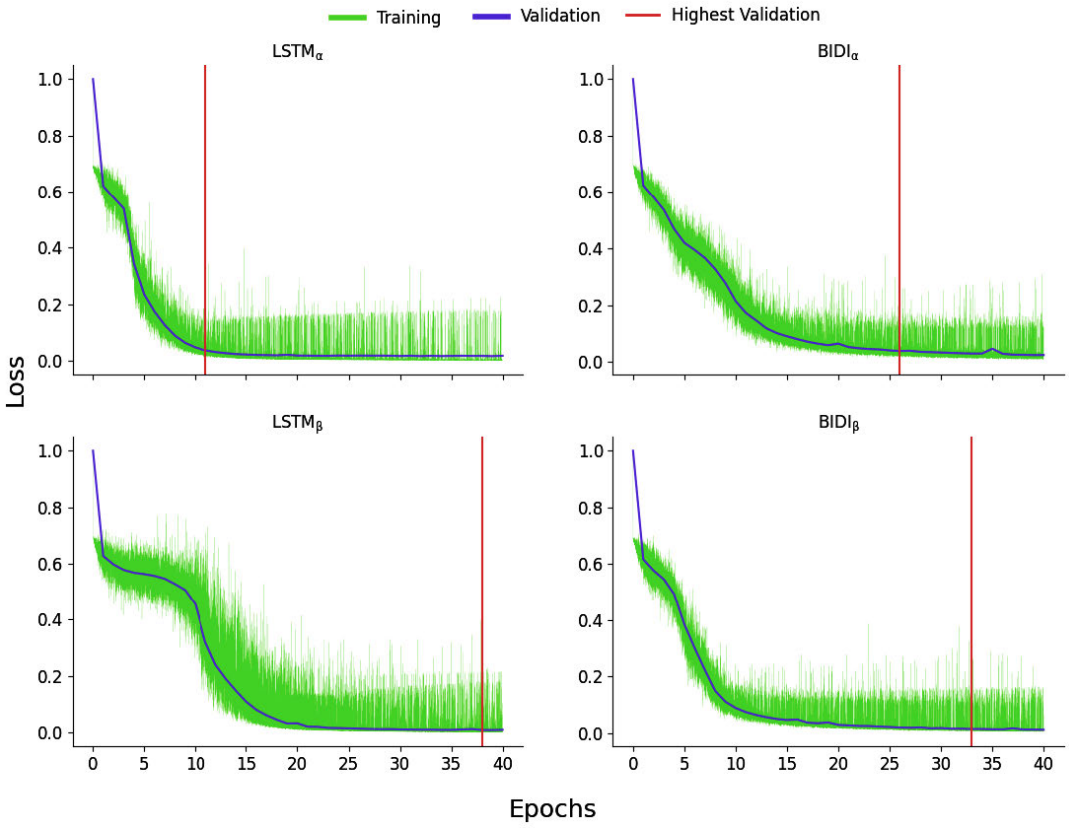

FIGURE 8. Mean cross entropy loss for LSTM and BIDI (α and β) models.

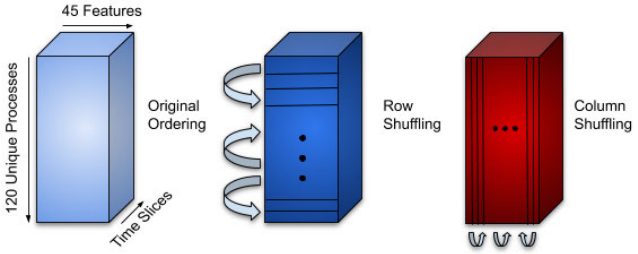

FIGURE 9. Input samples random ordering.

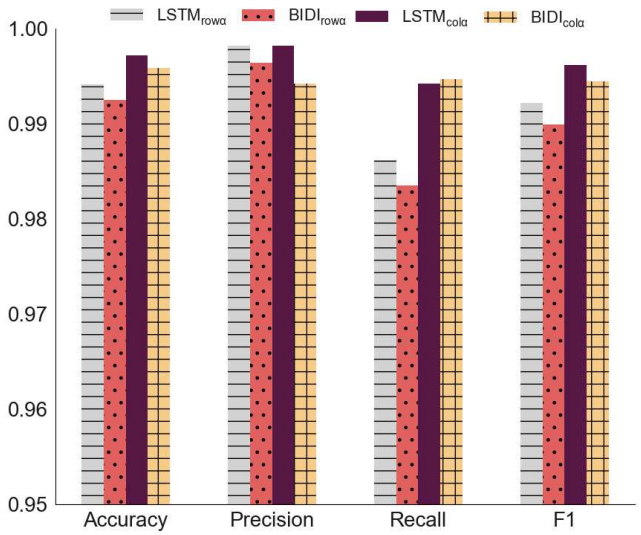

FIGURE 10. Results of LSTM and BIDI (colα and rowα) models.

colβ, colγ, and colδ) based on features random orderings. Additionally, we trained three LSTM (i.e. rowα, rowβ, rowγ) and BIDI models (i.e. rowα, rowβ, rowγ) based on unique processes random orderings.

Figure 10 shows the results of colα and rowα LSTM and BIDI models only (exact results numbers are shown in Table 4). LSTM$_{col\alpha}$ and BIDI$_{col\alpha}$ models achieved high results with close to 100% in all metrics. On the other hand, LSTM$_{row\alpha}$ and BIDI$_{row\alpha}$ have shown a decrease of $\approx$ 2% in terms of precision and $\approx$ 1% for f1 score. The evaluation metrics of the remaining trained models have reflected similar results as discussed in Table 6 and Table 8 in the Appendix. Figure 11 shows training and validation mean cross entropy loss during colα and rowα LSTM and BIDI models training phase. By examining the loss, it is clear that, unlike colα LSTM and BIDI models, rowα LSTM and BIDI models shows more fluctuation and difficulties in learning smoothly. We discuss and analyze these results in Section VII.

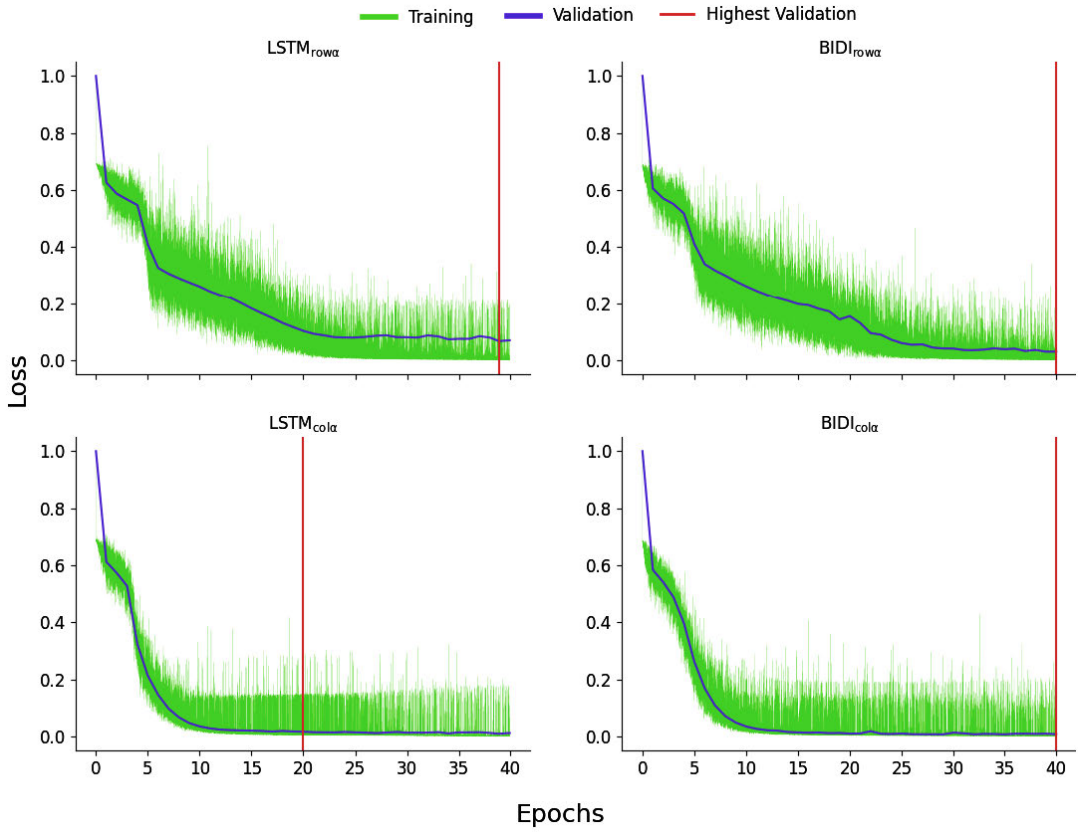

FIGURE 11. Mean cross entropy loss for LSTM and BIDI (colα and rowα) models.

TABLE 4. Results of LSTM and BIDI (colα and rowα) models.

Model	Accuracy	Precision	Recall	F1
LSTM$_{row\alpha}$	99.41%	99.81%	98.61%	99.21%
BIDI$_{row\alpha}$	99.19%	99.81%	98.04%	98.92%
LSTM$_{col\alpha}$	99.71%	99.82%	99.42%	99.62%
BIDI$_{col\alpha}$	99.58%	99.42%	99.46%	99.44%

TABLE 5. Training time for Optimal LSTM and BIDI models.

Model	Validation Accuracy	Epoch Reached	Time Elapsed(s)	Loss
LSTM$_{\alpha}$	99.62%	11	847	3.6%
BIDI$_{\alpha}$	99.85%	26	4160	3.8%
LSTM$_{\beta}$	99.65%	38	2888	0.9%
BIDI$_{\beta}$	99.94%	33	5247	1.4%
LSTM$_{row\alpha}$	97.77%	40	3045	2.9%
BIDI$_{row\alpha}$	99.24%	40	6219	3%
LSTM$_{col\alpha}$	99.78%	39	2993	1.8%
BIDI$_{col\alpha}$	99.94%	20	3126	2.2%

VI. COST ANALYSIS

In this section, we provide cost analysis with respect to the LSTM and BIDI models' training time. A problem with training RNN models is in the choice of the number of training epochs to use. Training the model for too many epochs can lead to overfitting (even with dropout layers), where as, training for too few epochs may lead the model to underfitting. In our case, we determined that 40 epochs are sufficient for our models to properly converge. During these 40 epochs, the set of weights that achieved the highest accuracy on the validation data set is recorded and chosen to be the most optimal model in each of the case. Although, some models can converge and learn faster than others.

Table 5 shows the epochs where the RNN models achieved the highest validation accuracy along with the time taken in seconds and the corresponding loss. In general, the LSTM models converged relatively faster than the BIDI models. The LSTM$_{\alpha}$, LSTM$_{\beta}$, LSTM$_{row\alpha}$, and LSTM$_{col\alpha}$ models were the fastest to converge respectively in 847, 2888, 3035, and 2993 seconds, while the BIDI$_{\alpha}$, BIDI$_{\beta}$, BIDI$_{row\alpha}$, and BIDI$_{col\alpha}$ models took more time to converge respectively in 4160, 5247, 6219, and 3126 seconds. This is due to the fact that BIDI models add another backwards layer to the model, and therefore increasing the total time to train. Input representation is another factor that affects the training time. The results in Table 5 indicate that some random *col* and *row* orderings[9] prolong the time taken for the model to converge.

[9]Detailed results for training time for different input representations are shown in Table 7 and Table 9 under Appendix.

Particularly, this is more evident for the *row* orderings where both $\text{LSTM}_{\text{row}\alpha}$ and $\text{BIDI}_{\text{row}\alpha}$ models achieved the highest validation accuracy in epoch 40. This can also be seen clearly in Figure 11, where $\text{LSTM}_{\text{row}\alpha}$ and $\text{BIDI}_{\text{row}\alpha}$ models training and validation curve is more fluctuant due to separating related unique processes.

As reflected by the results, all our models were able to detect the presence of malware within one input sample.

VII. DISCUSSION AND LIMITATIONS

In this section, we discuss some rationale about our results as well as limitations and potential future work improvements.

The results in Section IV-D illustrated that both LSTM and BIDI models achieve almost equally high performance. However, it was clear from the results in Section VI that the LSTM models achieved such performance in a shorter amount of training time. Even though the BIDI_{β} model achieved its highest validation accuracy after 33 epochs and the LSTM_{β} model achieved its highest validation accuracy after 38, the time taken for the BIDI model is almost double the time taken for the LSTM model. As such, there is no added value in using a BIDI models over LSTM models since both were able to achieve similar scores with respect to the evaluation metrics.

The results derived from the experiments in Section V-A showed that input representation in terms of the order of unique processes and features is a major concern in terms of detection performance and training cost. The *col* orderings results indicated that the order of features doesn't impact the performance of our models, where as, the *row* orderings results indicated that the order of unique processes in a given input sample impacts the results within a range of 1% to 2% in f1 score and recall. These results align with [41] where the authors concluded that the order of features is not as important as the order of processes in the samples for their CNN models. The order of processes highly impacted their obtained results with 8% enhancement after using an approach to provide a proper order. This advocates that RNN models are more robust to input sensitivity than CNN models. The aim of the random orderings experiments is to highlight the issue of input representation. A further analysis on how to systematically devise a proper input representation to enhance the results is left to future work.

One limitation in our work lies in the size of our experiments. We conducted 113 experiments each using a different malware executable, but more samples would allow us to obtain a deeper understanding of how our detection models perform against differing malware types. Another limitation in this work is the assumption that a VM is infected by a single malware. In practice, a VM can be infected by multiple malware simultaneously. An analysis of whether our detection system will work as expected during the presence of multiple malware working at once is needed. Further, our work focuses on detecting malware in a single VM. However, in cloud auto scaling architectures, a malware that infects a single VM can propagate to similarly configured VMs fairly quickly. As such, malware propagation as well as multiple malware infections are left to future work. Another limitation to our work is that it is possible for malware to slip in between the averages within a group of processes with the same name. This is a common drawback associated with any methodology that generates meta-stats (e.g., average, standard deviation, etc.). This drawback is confined to the unique process aspect of our approach since this is where we are averaging the measurements of processes in order to reduce the number of features.

TABLE 6. Random column ordering results.

Model	Accuracy	Precision	Recall	F1
$\text{LSTM}_{\text{col}\alpha}$	99.71%	99.82%	99.42%	99.62%
$\text{BIDI}_{\text{col}\alpha}$	99.58%	99.42%	99.46%	99.44%
$\text{LSTM}_{\text{col}\beta}$	99.73%	99.95%	99.33%	99.64%
$\text{BIDI}_{\text{col}\beta}$	99.59%	99.50%	99.42%	99.46%
$\text{LSTM}_{\text{col}\gamma}$	99.44%	99.19%	99.33%	99.26%
$\text{BIDI}_{\text{col}\gamma}$	99.51%	99.28%	99.42%	99.35%
$\text{LSTM}_{\text{col}\delta}$	99.28%	98.46%	99.64%	99.04%
$\text{BIDI}_{\text{col}\delta}$	99.04%	99.32%	98.13%	98.72%

VIII. CONCLUSION

In this paper, we introduce an approach of using LSTM and BIDI models for online malware detection based on processes system features. Results showed that both LSTM and BIDI models achieved outstanding performance (over 99%) on the testing dataset; however, the LSTM models required less time than the BIDI models to achieve such performance. Additionally, we analyzed the impact of input representations on our models by conducting random ordering experiments with respect to unique processes and features (i.e., *col* and *row* experiments). On one hand, the results showed that the order of the features doesn't impact the models performance, whereas, it impacts the training time of the models. On the other hand, the order of unique processes impacts the performance as well as the training time of the models.

In the future, we plan to increase the scale of our experiments by using thousands of malware samples including more malware families. Additionally, we plan to study the impacts of malware propagation to similarly configured VMs in a cloud environment on the robustness of our detection models. We also plan to study the impacts of multiple malware infections to the same VM.

APPENDIX A
RANDOM COLUMN ORDERINGS

Table 6, Figure 12, and Figure 13 show the remaining results generated by the LSTM and BIDI ($\text{col}\alpha$, $\text{col}\beta$, $\text{col}\gamma$, and $\text{col}\delta$) models that were trained each with a random feature ordering. In addition, Table 7 shows the training time and the number of epochs needed in order for these models to achieve their highest validation accuracy.

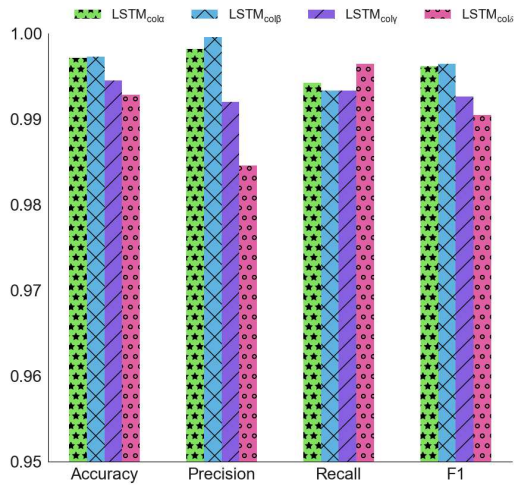

FIGURE 12. Results from LSTM models with random column ordering.

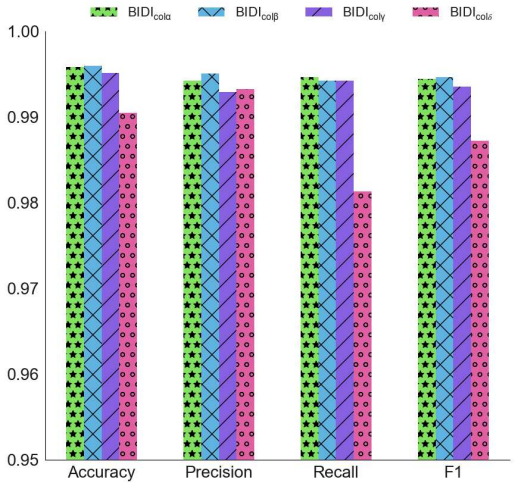

FIGURE 13. Results from BIDI models with random column ordering.

TABLE 7. Training time for random column ordering.

Model	Validation Accuracy	Epoch Reached	Time Elapsed(s)	Loss
LSTM$_{col\alpha}$	99.78%	39	2993	1.8%
BIDI$_{col\alpha}$	99.94%	20	3126	2.2%
LSTM$_{col\beta}$	99.77%	12	922	2%
BIDI$_{col\beta}$	99.94%	20	3121	2%
LSTM$_{col\gamma}$	99.66%	19	1456	2%
BIDI$_{col\gamma}$	99.91%	19	2954	2.4%
LSTM$_{col\delta}$	99.92%	26	1986	3%
BIDI$_{col\delta}$	99.89%	23	3595	3.1%

APPENDIX B
RANDOM ROW ORDERINGS

Table 8, Figure 14, and Figure 15 show the remaining results generated by the LSTM and BIDI (rowα, rowβ, and rowγ) models that were trained each with a random feature ordering.

TABLE 8. Random row ordering results.

Model	Accuracy	Precision	Recall	F1
LSTM$_{row\alpha}$	99.41%	99.81%	98.61%	99.21%
BIDI$_{row\alpha}$	99.19%	99.81%	98.04%	98.92%
LSTM$_{row\beta}$	99.56%	99.73%	99.10%	99.41%
BIDI$_{row\beta}$	99.24%	99.15%	98.84%	98.99%
LSTM$_{row\gamma}$	99.13%	99.45%	98.21%	98.83%
BIDI$_{row\gamma}$	99.24%	99.63%	98.35%	98.99%

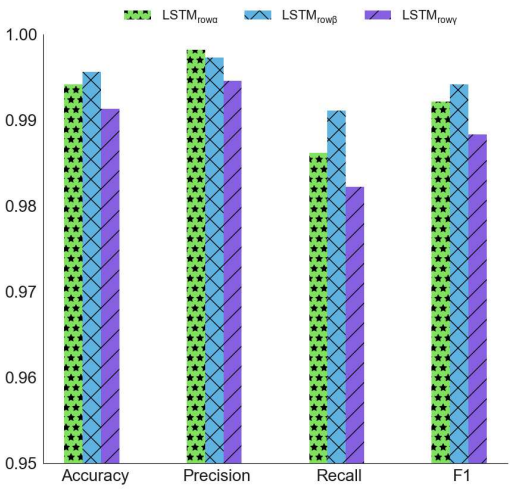

FIGURE 14. Results from LSTM models with random row ordering.

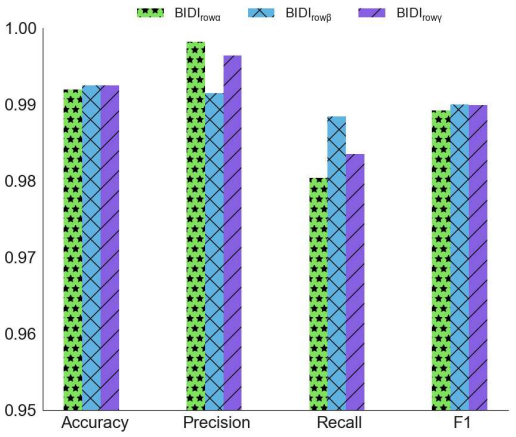

FIGURE 15. Results from BIDI models with random row ordering.

TABLE 9. Training time for random row ordering.

Model	Validation Accuracy	Epoch Reached	Time Elapsed(s)	Loss
LSTM$_{row\alpha}$	97.77%	40	3045	2.9%
BIDI$_{row\alpha}$	99.24%	40	6219	3%
LSTM$_{row\beta}$	99.48%	39	2986	2.4%
BIDI$_{row\beta}$	99.04%	34	5296	3.3%
LSTM$_{row\gamma}$	99.06%	36	2778	3.6%
BIDI$_{row\gamma}$	99.02%	40	6219	3%

In addition, Table 9 shows the training time and the number of epochs needed in order for these models to achieve their highest validation accuracy.

REFERENCES

[1] P. Mell and T. Grance, "The NIST definition of cloud computing," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. Special Publication 800-145, 2011.

[2] M. R. Watson, N.-U.-H. Shirazi, A. K. Marnerides, A. Mauthe, and D. Hutchison, "Malware detection in cloud computing infrastructures," *IEEE Trans. Dependable Secure Comput.*, vol. 13, no. 2, pp. 192–205, Mar. 2016.

[3] K. Dahbur, B. Mohammad, and A. B. Tarakji, "A survey of risks, threats and vulnerabilities in cloud computing," in *Proc. Int. Conf. Intell. Semantic Web-Services Appl.*, 2011, pp. 1–6.

[4] A. Gholami and E. Laure, "Security and privacy of sensitive data in cloud computing: A survey of recent developments," 2016, *arXiv:1601.01498*. [Online]. Available: http://arxiv.org/abs/1601.01498

[5] Z. Xiao and Y. Xiao, "Security and privacy in cloud computing," *IEEE Commun. Surveys Tuts.*, vol. 15, no. 2, pp. 843–859, May 2013.

[6] M. Rhode, P. Burnap, and K. Jones, "Early-stage malware prediction using recurrent neural networks," *Comput. Secur.*, vol. 77, pp. 578–594, Aug. 2018.

[7] A. Alotaibi, "Identifying malicious software using deep residual long-short term memory," *IEEE Access*, vol. 7, pp. 163128–163137, 2019.

[8] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi, "Malware detection with deep neural network using process behavior," in *Proc. IEEE 40th Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, Jun. 2016, pp. 577–582.

[9] M. Abdelsalam, R. Krishnan, and R. Sandhu, "Online malware detection in cloud auto-scaling systems using shallow convolutional neural networks," in *Proc. IFIP Annu. Conf. Data Appl. Secur. Privacy*. Cham, Switzerland: Springer, 2019, pp. 381–397.

[10] A. Shalaginov, S. Banin, A. Dehghantanha, and K. Franke, "Machine learning aided static malware analysis: A survey and tutorial," in *Cyber Threat Intelligence*. Cham, Switzerland: Springer, 2018, pp. 7–45.

[11] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 559–570, Jun. 2013.

[12] A. McDole, M. Gupta, M. Abdelsalam, S. Mittal, and M. Alazab, "Deep learning techniques for behavioral malware analysis in cloud IaaS," in *Malware Analysis Using Artificial Intelligence and Deep Learning*. Cham, Switzerland: Springer, 2021, pp. 269–285.

[13] M. Abdelsalam, R. Krishnan, Y. Huang, and R. Sandhu, "Malware detection in cloud infrastructures using convolutional neural networks," in *Proc. IEEE 11th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2018, pp. 162–169.

[14] I. Firdausi, C. Lim, A. Erwin, and A. S. Nugroho, "Analysis of machine learning techniques used in behavior-based malware detection," in *Proc. 2nd Int. Conf. Adv. Comput., Control, Telecommun. Technol.*, Dec. 2010, pp. 201–203.

[15] R. S. Pirscoveanu, S. S. Hansen, T. M. T. Larsen, M. Stevanovic, J. M. Pedersen, and A. Czech, "Analysis of malware behavior: Type classification using machine learning," in *Proc. Int. Conf. Cyber Situational Awareness, Data Anal. Assessment (CyberSA)*, Jun. 2015, pp. 1–7.

[16] P. Luckett, J. T. McDonald, and J. Dawson, "Neural network analysis of system call timing for rootkit detection," in *Proc. Cybersecur. Symp. (CYBERSEC)*, Apr. 2016, pp. 1–6.

[17] Y. Fan, Y. Ye, and L. Chen, "Malicious sequential pattern mining for automatic malware detection," *Expert Syst. Appl.*, vol. 52, pp. 16–25, Jun. 2016.

[18] S. Joshi, H. Upadhyay, L. Lagos, N. S. Akkipeddi, and V. Guerra, "Machine learning approach for malware detection using random forest classifier on process list data structure," in *Proc. 2nd Int. Conf. Inf. Syst. Data Mining (ICISDM)*, 2018, pp. 98–102.

[19] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, "Malware classification with recurrent networks," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Apr. 2015, pp. 1916–1920.

[20] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *Proc. Australas. Joint Conf. Artif. Intell.* Cham, Switzerland: Springer, 2016, pp. 137–149.

[21] M. A. Halim, A. Abdullah, and K. A. Z. Ariffin, "Recurrent neural network for malware detection," *Int. J. Advance Soft Compu. Appl*, vol. 11, no. 1, pp. 43–63, 2019.

[22] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah, "Android malware detection based on system call sequences and LSTM," *Multimedia Tools Appl.*, vol. 78, no. 4, pp. 3979–3999, Feb. 2019.

[23] W. Xie, S. Xu, S. Zou, and J. Xi, "A system-call behavior language system for malware detection using a sensitivity-based LSTM model," in *Proc. 3rd Int. Conf. Comput. Sci. Softw. Eng.*, May 2020, pp. 112–118.

[24] P. Mishra, K. Khurana, S. Gupta, and M. K. Sharma, "VMAnalyzer: Malware semantic analysis using integrated CNN and bi-directional LSTM for detecting VM-level attacks in cloud," in *Proc. 12th Int. Conf. Contemp. Comput. (IC)*, Aug. 2019, pp. 1–6.

[25] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 559–570, Jun. 2013.

[26] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2015, pp. 651–661.

[27] Z. Xu, S. Ray, P. Subramanyan, and S. Malik, "Malware detection using machine learning based analysis of virtual memory access patterns," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 169–174.

[28] F. Azmandian, M. Moffie, M. Alshawabkeh, J. Dy, J. Aslam, and D. Kaeli, "Virtual machine monitor-based lightweight intrusion detection," *ACM SIGOPS Oper. Syst. Rev.*, vol. 45, no. 2, pp. 38–53, Jul. 2011.

[29] Q. Guan, Z. Zhang, and S. Fu, "Ensemble of Bayesian predictors and decision trees for proactive failure management in cloud computing systems," *J. Commun.*, vol. 7, no. 1, pp. 52–61, Jan. 2012.

[30] M. Abdelsalam, R. Krishnan, and R. Sandhu, "Clustering-based IaaS cloud monitoring," in *Proc. IEEE 10th Int. Conf. Cloud Comput. (CLOUD)*, Jun. 2017, pp. 672–679.

[31] J. A. Dawson, J. T. McDonald, L. Hively, T. R. Andel, M. Yampolskiy, and C. Hubbard, "Phase space detection of virtual machine cyber events through hypervisor-level system call analysis," in *Proc. 1st Int. Conf. Data Intell. Secur. (ICDIS)*, Apr. 2018, pp. 159–167.

[32] A. McDole, M. Abdelsalam, M. Gupta, and S. Mittal, "Analyzing CNN based behavioural malware detection techniques on cloud IaaS," in *Proc. CLOUD*, 2020, pp. 64–79.

[33] F. Xiao, Z. Lin, Y. Sun, and Y. Ma, "Malware detection based on deep learning of behavior graphs," *Math. Problems Eng.*, vol. 2019, pp. 1–10, Feb. 2019.

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[35] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 4700–4708.

[36] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," 2014, *arXiv:1406.1078*. [Online]. Available: http://arxiv.org/abs/1406.1078

[37] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, May 2013, pp. 6645–6649.

[38] F. Ilhan, O. Karaahmetoglu, I. Balaban, and S. S. Kozat, "Markovian RNN: An adaptive time series prediction network with HMM-based switching for nonstationary environments," 2020, *arXiv:2006.10119*. [Online]. Available: http://arxiv.org/abs/2006.10119

[39] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[40] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Trans. Signal Process.*, vol. 45, no. 11, pp. 2673–2681, Nov. 1997.

[41] R. Klepetko and R. Krishnan, "Analyzing CNN model performance sensitivity to the ordering of non-natural data," in *Proc. 4th Int. Conf. Comput., Commun. Secur. (ICCCS)*, Oct. 2019, pp. 1–8.

JEFFREY C. KIMMELL is currently pursuing the B.S. degree in computer science. He will continue to pursue the master's degree in computer science. He is also a senior with Tennessee Technological University. His research interests include deep learning and AI based malware analysis in cloud.

ANDREW D. MCDOLE received the B.S. degree from Tennessee Tech University. He is currently a Graduate Student of Computer Science with Tennessee Tech University. He is also a member of CyberCorps and has worked extensively in deep learning based and AI malware analysis in cloud.

MAHMOUD ABDELSALAM received the B.Sc. degree from the Arab Academy for Science and Technology and Maritime Transportation (AASTMT), in 2013, and the M.Sc. and Ph.D. degrees from The University of Texas at San Antonio (UTSA), in 2017 and 2018, respectively. He is currently an Assistant Professor with the Department of Computer Science, Manhattan College. Prior to joining the Manhattan College, he worked as a Postdoctoral Research Fellow with the Institute for Cyber Security (ICS), UTSA. His research interests include computer systems security, anomaly and malware detection, cloud computing security and monitoring, cyber physical systems security, and applied machine learning.

MAANAK GUPTA (Member, IEEE) received the B.Tech. degree in computer science and engineering, India, the M.S. degree in information systems from Northeastern University, Boston, MA, USA, and the M.S. and Ph.D. degrees in computer science from The University of Texas at San Antonio (UTSA). He worked as a Postdoctoral Fellow with the Institute for Cyber Security (ICS), UTSA. He is currently an Assistant Professor of Computer Science with Tennessee Technological University, Cookeville, TN, USA. His research interests include security and privacy in cyber space focused in studying foundational aspects of access control and their application in technologies, including cyber physical systems, cloud computing, the IoT, and big data. He has worked in developing novel security mechanisms, models and architectures for next generation smart cars, smart cities, intelligent transportation systems, and smart farming. He is also interested in machine learning based malware analysis and AI assisted cyber security solutions. His research has been funded by the U.S. National Science Foundation (NSF), NASA, the U.S. Department of Defense (DoD), and private industry.

RAVI SANDHU (Fellow, IEEE) received the B.Tech. degree from IIT Bombay, the M.Tech. degree from IIT Delhi, and the M.S. and Ph.D. degrees from Rutgers University. He is currently a Professor of Computer Science, the Executive Director of the Institute for Cyber Security, and a Lead PI of the NSF Center for Security and Privacy Enhanced Cloud Computing, The University of Texas at San Antonio, where he holds the Lutcher Brown Endowed Chair of Cyber Security. Previously, he has served on the faculty for George Mason University, from 1989 to 2007, and Ohio State University, from 1982 to 1989. A prolific and highly cited author, his research has been funded by NSF, NSA, NIST, DARPA, AFOSR, ONR, AFRL, ARO, and private industry. He is also a Fellow of ACM and AAAS. He received numerous awards from IEEE, ACM, NSA, NIST, and IFIP. He was the Chairman of ACM SIGSAC, and founded the ACM Conference on Computer and Communications Security, the ACM Symposium on Access Control Models and Technologies, and the ACM Conference on Data and Application Security and Privacy.