Malware Detection in Cloud Infrastructures using Convolutional Neural Networks

Mahmoud Abdelsalam*, Ram Krishnan†, Yufei Huang‡ and Ravi Sandhu§
*†§Institute for Cyber Security and Center for Security and Privacy Enhanced Cloud Computing,
*§Department of Computer Science, †‡Department of Electrical and Computer Engineering
University of Texas at San Antonio, San Antonio, Texas, USA
Email: *mahmoud.abdelsalam@utsa.edu, †ram.krishnan@utsa.edu, ‡yufei.huang@utsa.edu, §ravi.sandhu@utsa.edu

***Abstract*—A major challenge in Infrastructure as a Service (IaaS) clouds is its exposure to malware. Malware can spread rapidly within a datacenter and can cause major disruption to a cloud service provider and its clients. This paper introduces and discusses an effective malware detection approach in cloud infrastructure using Convolutional Neural Network (CNN), a deep learning approach. We initially employ a standard 2d CNN by training on metadata available for each of the processes in a virtual machine (VM) obtained by means of the hypervisor. We enhance the CNN classifier accuracy by using a novel 3d CNN (where an input is a collection of samples over a time interval), which greatly helps reduce mislabelled samples during data collection and training. Our experiments are performed on data collected by running various malware (mostly Trojans and Rootkits) on VMs. The malware used in our experiments are randomly selected. This reduces the selection bias of known-to-be highly active malware for easy detection. We demonstrate that our 2d CNN model reaches an accuracy of $\simeq 79\%$, and our 3d CNN model significantly improves the accuracy to $\simeq 90\%$.**

***Keywords*-Security; Malware Detection; Cloud IaaS; Deep Learning; Convolutional Neural Networks**

I. Introduction

Cloud infrastructure has become increasingly prone to novel attacks and malware [1]–[6]. One of the most prevalent threats to cloud is malware. Cloud malware injection [3] is a threat where an attacker injects a malware to manipulate the victim's Virtual Machine (VM). Due to the nature of the cloud and automatic provisioning, a large number of the VMs are similarly configured. One example is when a web server scales-out due to increase in demand and scales-in when the demand goes down. This means that the attack that compromised one of the VMs is highly likely to compromise many of the other VMs. The attacker can inject a botware to use it for creating a botnet due to a large number of VMs available in scaling scenarios. As a result, the need for malware detection in VMs is critical.

In our earlier work [7], we showed that malware detection can be effectively performed by inspecting the performance and resource utilization metrics of VMs as a black-box. Although the approach works well with highly active malware (e.g. ransomware), it is not as effective for detecting malware that maintains a low-profile of resource utilization. In this paper, we develop a novel and effective technique to detect such low-profile malware that utilizes minimal system resources, by inspecting raw, fine-grained meta-data of each process in a VM.

Two major approaches have been explored for malware detection in the current literature: static analysis, where malware code is analyzed without running it, and dynamic analysis, where a malware is executed and its behavior observed in order to detect it. The pros and cons of these approaches for malware detection are well understood.

In this paper, we introduce and discuss a malware detection approach using Deep Learning (DL). We demonstrate the applicability of using a 2d Convolutional Neural Network (CNN) for malware detection through the utilization of raw, process behavior (performance metrics) data. Our approach falls under dynamic analysis. However, unlike most prior works that utilize machine learning (ML) in dynamic analysis to classify malware files, we use it for online malware detection. Note that the approach introduced in this paper is general and is not confined to the use of CNN. The choice of using CNN is due to its simplicity and training speed as opposed to other DL architectures such as Recurrent Neural Networks (RNNs). Applying and comparing different ML approaches is left to future work.

One of the biggest challenges in employing ML for malware detection is the **mislabeling problem**. This is because, during the training phase, there is no guarantee that a malware exhibited malicious behavior. While some malware start performing malicious activities immediately after infecting a machine, a reasonably sophisticated malware starts off as a process and idles until some condition is met (e.g., a command from its remote owner), which can occur at any time. In particular, such a condition may never occur during the training phase for the malware to activate. However, this issue is rarely addressed in existing literature except for the work in [8], which recognizes this issue. The authors stated that this problem can pollute the training and testing data; however, since there is no way around it, they had to make the assumption that it is alright to label all the data as malicious after a malware execution takes place. In other words, the assumption is that malware will always show malicious activity at all times.

We follow their assumption in this work but not to the fullest. Consider a more common scenario when a malware periodically (e.g., every 1 minute) performs malicious activ-

ities such as stealing and sending some information to its Command and Control servers (C&Cs). Now the malware is surely conducting a malicious behavior but only periodically. As a result, if a malware is run for 15 minutes and we collect a data sample every 10 seconds (total of 90 samples), all the collected data samples will be labeled as malicious whereas in fact only 15 of them are malicious. This will cause a **mislabeling problem** during the training phase.

To mitigate this problem, we refine the above assumption by assuming that a malware will show malicious activity within a time window. The underlying rationale is that while there is no way to know for sure that a malware ever exhibited malicious behavior during the training phase, it is more practical to consider a sliding window of time during which malicious behavior is exhibited instead of assuming that all data samples collected after malware injection indicate malicious activity. This increases the probability of correctly labeling our samples. Toward this end, we develop a 3d CNN classifier which takes a 3d input matrix containing multiple samples over a time window. In summary, the contributions of this paper are two-fold:

- We develop an effective approach for detecting malware by learning behavior from fine-grained and raw process meta-data that are available directly from the hypervisor. The approach we develop is resistant to the aforementioned mislabeling problem.
- We demonstrate the effectiveness of this approach by first developing a standard 2d CNN model that does not incorporate the time window, and then comparing it with a newly developed 3d CNN model that significantly improves detection accuracy mainly due to the employment of a time window as the third dimension, thereby mitigating the mislabeling problem.

To the best of our knowledge, our work is the first to apply 2d and 3d CNN on raw performance metrics of processes, which can be easily obtained through the hypervisor layer. This is critical if a cloud service provider were to offer such a malware detection service. Since the approach we propose does not require an agent to run within VMs, we avoid any major privacy and security concern for cloud tenants.

The remainder of the paper is organized as follows. Section II discusses related work on malware detection methods outlined as static and dynamic analysis. Section III outlines the methodology including the architecture of the CNN models used. Section IV describes the experiments setup and results. Section V gives a discussion about some of the important limitations and possible mitigations. Section VI summarizes and concludes this paper.

II. Related Work

This section provides an abbreviated introduction to the major malware detection techniques using ML. The majority of malware detection techniques falls under one of the two approaches: static analysis or dynamic analysis.

A. Static Analysis using Machine Learning

During static analysis, no execution of executables/binary files takes place. It is the process of analyzing executables by examining their code without actually executing them. There are two approaches used for static analysis. First, an executable file can be disassembled or reverse engineered using disassemblers to get the actual code. Then detection of malware takes place on the actual code. Most sophisticated malware can evade this method by embedding syntactic code errors that will confuse disassemblers but that will still function during actual execution. Second, analysis can be done directly on a binary file format. For example, one of the simplest forms of static analysis, is extracting parts of the binary file as features (n-grams). Then ML techniques are used to find malicious patterns. In [9], the approach is to remove n-grams that are known to be benign. For example, a worm that distributes itself via emails contains code to send an email which is benign in many applications, so removing these segments from the file, while comparing what is left to known malicious segments is a valid approach. The paper used different ML techniques including Artificial Neural Networks (ANN) and Decision Trees (DT). The works in [10]–[12] are similar but use different ML algorithms.

The authors in [13] use several DL models including LSTM and GRU based language models as well as a CNN model. This is a static analysis approach which works directly on the malware files without executing it. Similarly, [14], [15] use DL for malware static analysis.

Malware developers evade detection using static analysis approaches by introducing polymorphism, where a malware changes and evolves while preserving code semantics. Dynamic analysis approaches can help overcoming some of the static analysis drawbacks since they rely on monitoring the behavior as opposed to static inspection.

B. Dynamic Analysis using Machine Learning

In dynamic analysis, the executable is executed, typically, in an isolated environment (e.g., sandbox or VM) and information is gathered during execution (e.g., system calls, memory accesses or network communications). Dynamic analysis is used for malware files classification as well as for online malware detection (e.g., similar to intrusion detection systems). Many works exist in this area. In [16], the authors use ML techniques in order to monitor virtual memory for malicious access patterns caused by the malware. The features are represented in histograms of memory access. The authors train one model for each application which can be quite expensive.

The work in [17] uses multi-task learning using Deep Neural Networks (DNN) for malware detection and malware family classification of binary files. In multi-task learning, a set of network layers is shared between learning tasks.

The work in [18] applies deep learning for malware detection using process API calls log information. First, a Recurrent Neural Network (RNN) is used to extract features and then CNN is given these features as input. The downside

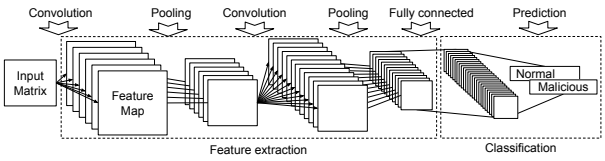


Fig. 1: CNN overview

of this work is using a sandbox to monitor processes. In most cases, malware will detect the presence of a sandbox and hide its true behavior. Also, this deals with single data sample without considering that malware can be benign at certain times and malicious at others.

The work in [19] uses dynamic analysis for malware files classification. Malware run in a lightweight VM and hundreds of thousands of features are extracted to be used in a DL technique. This approach is for files classification which is not naturally suitable for online malware detection.

Most dynamic analysis approaches for online detection deal with single samples and do not consider the mislabeling problem nor malicious patterns across windows of time.

In this paper, we are motivated by:

- The feasibility of applying CNN to VMs malware detection using fine-grained process performance metrics.
- Tackling the mislabeling problem by using 3d CNNs.

III. Methodology

This section provides an overview of the methodology used for malware detection in VMs using CNN.

A. Convolutional Neural Network

CNN is a type of DL that has been applied to images analysis and classification. One advantage of CNN is that it requires little pre-processing as compared to similar image classification algorithms since it works on raw data. It acts as a feature extractor which is very convenient since feature selection in most cases requires human experts.

Figure 1 shows the architectural overview of a CNN. Much like deep neural networks, CNN consists of input and output layers and multiple hidden layers. A *Convolutional layer* applies a convolution operation on the input matrix and passes the output to the next layer. A convolution operates on two inputs: feature map (input matrix) and convolution kernel (works as a filter) and outputs another image. The kernel is used to filter out certain information from the feature map and discard other information. In other words, a convolution operation uses multiple kernels where each kernel is responsible to extract and focus on a piece of information (e.g., one kernel might filter edge information). Usually, a convolutional layer is followed by a *Pooling layer* which takes the output of the convolutional layer as input. Pooling is an operation in which it down samples the feature maps received from the convolutional layer. It works by taking a certain area of the input and reduces it to a single value. For example, max pooling uses the maximum value from certain area, while average pooling uses the average value. Convolutional and pooling layers are followed by fully connected layers, which connect every neuron in one layer to every neuron in the next layer.

B. Process Performance Metrics

In this work, we use performance metrics as a way of defining a process behavior. Table I shows metrics that are selected to be collected for the VMs. Selected metrics are for the purpose of showing the effectiveness of our approach; in practice, many more metrics are available. For the sake of practicality, we assume no prior knowledge of any additional information other than the metrics we collect in Table I.

C. CNN Input

We represent each sample as an image (2d matrix) which will be the input to the CNN. Consider a sample X_t at a particular time t, that records n features (performance metrics) per process for m processes in a VM, such that:

$$\mathbf{X}_t = \begin{bmatrix} & f_1 & f_2 & \dots & f_n \\ p_1 & \vdots & \vdots & \dots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_m & \vdots & \vdots & \dots & \vdots \end{bmatrix}$$

Note that a CNN requires the same process to remain in the same row in each sample. For example, a process with PID 1 that resides in the first row of the matrix must remain in the first row across all upcoming samples. The CNN in computer vision takes fixed-size images as inputs, so the number of features (n) and processes (m) must be predetermined. The number of features is easily determined since we have a fixed number of collected features represented in Table I (28 in our case). On the other hand, determining the number of processes is not as easy since the processes are dynamic in nature. In highly active systems (e.g., web or app server), many processes get created and killed to handle client requests based on the workload.

A process is defined by a process identification number (PID) which is assigned by the OS. In a Linux based OS (used in our experiments), PID numbers will increase to a maximum system-dependent limit and then wrap around (recycle). The kernel will not reuse a PID before this wrap-around occurs.[1] The limit (maximum number of PIDs) is defined in /proc/sys/kernel/pid_max which is usually 32k. This number presents a problem because a matrix of $32k \times 28$ is a huge input matrix. Also having too many variables in the input requires a large number of input in any neural network. Limiting the max number of processes to a lower value and depending on the concept of wrap-around will not solve the problem because of many reasons. First, the reason the max number of processes is set to a very large number (i.e. 32k) is that it can confuse the kernel if the value is too small and wraps around too often, not to mention that it is hard to determine the appropriate number before hand.

[1]Linux Manual. http://man7.org/linux/man-pages/man5/proc.5.html

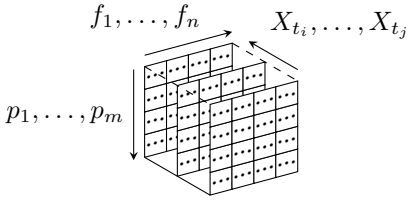
TABLE I: Virtual machines performance metrics

Metric Category	Description
Status	Process status
CPU information	CPU usage percent, CPU times in user space, CPU times in system/kernel space, CPU times of children processes in user space, CPU times of children processes in system space.
Context switches	Number of context switches voluntary, Number of context switches involuntary
IO counters	Number of read requests, Number of write requests, Number of read bytes, Number of written bytes, Number of read chars, Number of written chars
Memory information	Amount of memory swapped out to disk, Proportional set size (PSS), Resident set size (RSS), Unique set size (USS), Virtual memory size (VMS), Number of dirty pages, Amount of physical memory, text resident set (TRS), Memory used by shared libraries, memory that with other processes
Threads	Number of used threads
File descriptors	Number of opened file descriptors
Network information	Number of received bytes, Number of sent bytes

Second, there is no guarantee that, for instance, a process with a PID 1000 at time t_1 is going to be the same process at time t_{100}. Considering the wrap-around concept, this process might have been killed and a new different process could be assigned the same PID later on. This can cause inaccurate results by the CNN since an important requirement is that the same processes remain in the same rows at all times.

To solve these problems, instead of defining a process by it's PID, we define a process, referred to as **unique process**, by a 3-tuple: process name, command line used to run process, and the hash of the process binary file (if applicable). In cases where the same application (e.g., apache web server) forks multiple child processes (with the same name, cmd, and originated binary), we aggregate these processes by taking the average of their performance metrics. This also helps in smoothing the fluctuations of processes that have similar functions. In all of our experiments none of the VMs had more than 100 unique processes; however, for practicality, we set the maximum number of unique processes to 120 to accommodate for newly created unique processes. Any unavailable unique process (due to termination) at a particular time is padded with zero-values. In the rest of the paper, the term process and unique process are used interchangeably, where they refer to unique process.

The 3d CNN model input includes multiple samples over a time window. The input matrix $\mathbf{X}_{t_{ij}} =$



, where $\mathbf{X}_{t_{ij}}$ is the 3d input matrix containing samples from time t_i to t_j. As stated in section I, we use a 3d CNN model to enhance the results by capturing patterns over a small time window which in turn helps in mitigating the mislabeling problem.

IV. Experiment Setup and Results

In this section, first, we present the CNN model used in this work as well as the data preprocessing step. Second, we review our experimental setup. Then, we provide the results to illustrate that a 2d CNN can be effective in detecting low-profile malware using per-process performance metrics. Lastly, we show how using a 3d CNN can improve the results by attempting to solve the mislabeling problem.

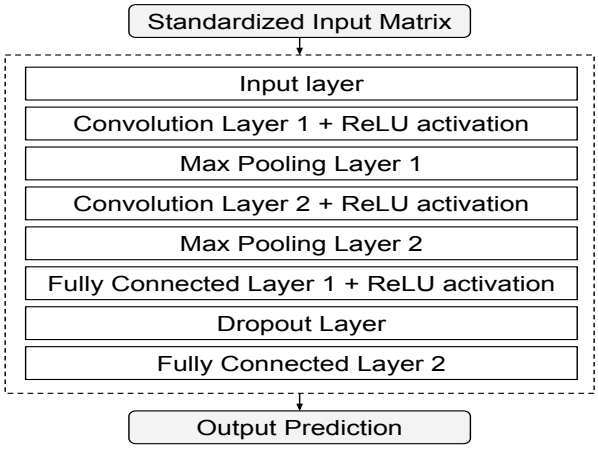


Fig. 2: Proposed CNN Model

A. Preprocessing

It is essential to CNN to have scaled data input for faster convergence and better accuracy results. A standard approach is to rescale the data to have a mean of 0 and standard deviation of 1. It is done in a per feature fashion. Given a set of features $F = \{f_1, f_i, ..., f_n\}$ and a set of samples $X = \{x_1, x_j, ..., x_t\}$, it is defined as $x^{(f_i)}_{j_{standardized}} = (x^{(f_i)}_j - \mu^{(f_i)})/\sigma^{(f_i)}$, where $x^{(f_i)}_j$ is a vector of values corresponding to feature f_i in the jth input sample, and $\mu^{(f_i)}$, $\sigma^{(f_i)}$ are respectively the mean and the standard deviation of values corresponding to feature f_i across all samples in set X. The same two sets of $\mu^{(f_i)}$ and $\sigma^{(f_i)}$ (obtained from the training dataset) are used for standardizing the validation and testing datasets.

B. CNN Model Architecture

Figure 2 shows the CNN model used in this work. It consists of 8 layers. First, the input layer which is basically received as the input matrix. Second, a convolutional layer which receives a $d \times 120 \times 28$ standardized matrix, representing samples in a particular time window, where

d is the depth of the input matrix and 120×28 is the length of the 2d matrices representing the number processes and features, respectively. Then, it performs a convolutional operation with 32 kernels of size $d \times 5 \times 5$ with zero-padded ending. The results of this layer are 32 feature maps of size $d \times 120 \times 28$. Third, a max pooling layer of size $2 \times 2 \times 2$ which down size each dimension by a magnitude of 2, resulting in a 32 feature maps of size $d/2 \times 60 \times 14$. The fourth and fifth layer are replicates of layer two and three so the output of the max pool layer 2 is 64 feature maps of size $d/4 \times 30 \times 7$. The last 3 layers are a fully connected layer with size of 1024, a dropout layer described below, and, last, another fully connected layer with size of 2 denoting the classification probability of a malicious or benign VM sample. Note that the model doesn't classify malicious or benign processes but rather the VM as a whole which means there is no way to know which process is malicious.

To reduce over fitting, we use a dropout [20] layer after the first fully connected layer, since it is shown in previous work [21] that dropout regularization works well with fully connected layers.

Rectified linear unit (ReLU), a simple and fast activation function, is simply defined as $f(x) = max(0, x)$. It turned out that ReLU (which is used in our work) works better in practice than the other activation functions as well as it's several times faster in training as stated in [22].

The model is trained using back-propagation for Adam Optimizer [23], a stochastic gradient descent that automatically adapt the learning rate. The optimizer works on minimizing the loss function. We use the mean cross entropy as a loss function. The model is also trained using mini-batches which is not reflected in the layers described above.

The described CNN model is used for both 2d CNN and 3d CNN except the former has one less dimension (i.e. the depth d of the input matrix is 1). The CNN structure used in this work is considered to be shallow as opposed to models such as GoogleNet and LeNet due to the limit of the experiments we could perform in our lab which, in turn, led to lack of large data sets. Experimenting in a larger scale and comparing different CNN models is left to future work.

C. Parameters Tuning

Parameters tuning is a very challenging problem in ML in general. It helps choosing the set of parameters that yields the best classification accuracy. A common approach, used for most of the parameters in our work, is grid search, where we define (based on our knowledge) bounds for each parameter and try all the combinations that yields the best classification accuracy during the validation phase. Other approaches can be more practical such as random search [24]. In our case, the set of important parameters are as follows. **Dropout**. Dropout is a regularization technique that turns neurons on/off in each layer to force them to go through different path. This operation improves generalization of the network and prevents over-fitting. We set this parameter to 0.5 [25]. **Learning rate**. This determines how fast we move toward the optimal weights in our network. If this parameter is very large, it will skip optimal values. On the other hand, if it is too small, it will take too much time to converge to the optimal values, and it may get stuck in local minima. Typically, a stochastic gradient descend uses decay learning rate to slow down the learning rate as it moves forward. AdamOptimizer (used in our study), adapts the learning rate automatically, however the adaptation maximum ceiling is defined by our learning rate parameter. If we set it very large, it will give the optimizer more room to adapt which can be problematic in some cases. The values we found reasonable during our experiments lies between $1e-3$ and $1e-5$. **Mini-batch size**. As CNN is using mini-batches to learn, we define the bounds of our mini-batches sizes between 10 and 30. Going lower or higher proved to decrease the accuracy.

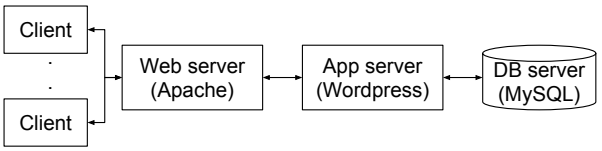


Fig. 3: 3-tier web architecture

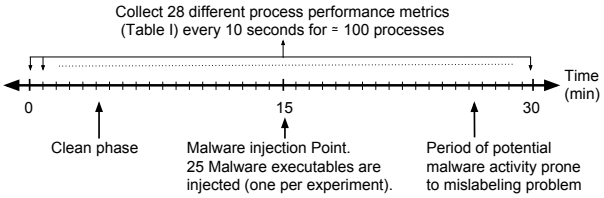


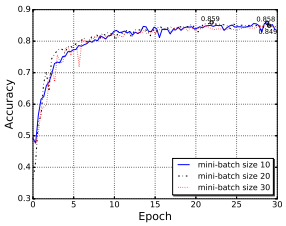
Fig. 4: Data collection overview

D. Experimental Setup

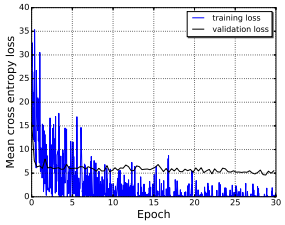
Our experiments were conducted on Openstack[2] (a major open-source cloud orchestration software). To simulate a real world scenario, we used a 3-tier web architecture (one of the most common cloud architectures according to Amazon[3]). Note that our work is not confined to the 3-tier web architecture use case used in the experiments since our approach relies on learning the behavior of processes in VMs. This means that learning approach of processes behavior would remain the same regardless the architecture in place. Figure 3 shows the setup used to conduct our experiments on Openstack. A 3-tier web architecture, typically, consists of 3 separate tiers: web, application and database server. In our case, we used Apache as a web server, Wordpress[4] (a major open-source content management system) that utilizes PHP as an application server and MySQL as a database server.

According to [26], Internet traffic is of self-similar nature. Thus, we built a multi-process traffic generator (set to the NS2[5] default parameters values), based on ON/OFF Pareto

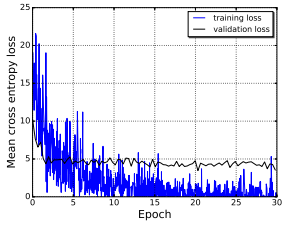
[2]Openstack website. https://www.openstack.org/
[3]Amazon architecture references. https://aws.amazon.com/architecture/
[4]Wordpress website. https://wordpress.org/
[5]NS2 tool manual. http://www.isi.edu/nsnam/ns/doc/node509.html



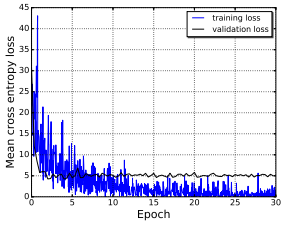
(a) Accuracy of 2d CNN. mini-batch sizes of 10, 20 and 30



(b) Mean cross entropy loss of 2d CNN. mini-batch size = 10



(c) Mean cross entropy loss of 2d CNN. mini-batch size = 20



(d) Mean cross entropy loss of 2d CNN. mini-batch size = 30

Fig. 5: 2d CNN trained with different mini-batch sizes. Optimized with learning rate of 1e-5 for AdamOptimizer

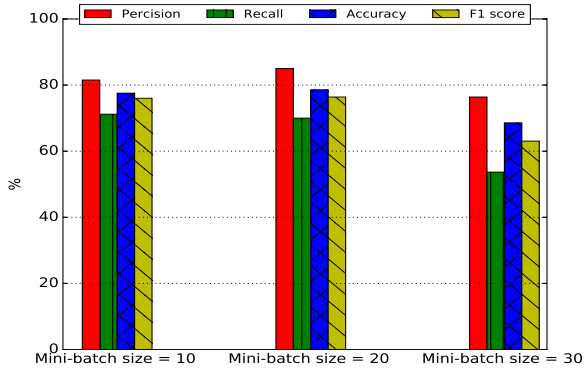


Fig. 6: 2d CNN classifiers results

distribution, to generate traffic for our experiments.

Figure 4 shows an overview of the data collection process. Each of our experiments was 30 minutes long. The aforementioned 3-tier architecture was created from known-to-be clean images. The first 15 minutes is the normal phase, where no malicious activity takes place, and is followed by 15 minutes of malicious phase, where a single malware is injected and executed in the application server. Few normal processes were injected during the normal phase to check the effectiveness of our approach in handling false positives. The malware was injected in the application server VM because most vulnerabilities, typically, lies in the application side.

The image used for spawning VMs is Ubuntu 16.04 which was modified to include a data collection agent. Data was collected at 10-second intervals in a JSON object. We refer to each of the collected objects at a particular time as a sample. For simplicity, we included an agent inside VMs to collect data; however, data collection could also be done through Virtual Machine Introspection (VMI) since similar metrics [27], [28] could be collected from the hypervisor.

The 25 malware binaries[6] used were randomly obtained from VirusTotal[7]. They mainly belong to 3 classes: Rootkits, Trojans and Backdoors and have unique SHA-256 hashes.

Most malware check for connection to their C&Cs, otherwise, they remain idle. Many researches (on malware dynamic analysis) use sandboxes or VMs in a controlled environment which can cause hindrance to the malware. To accommodate for this problem, all of our VMs are connected to the Internet outside of firewalls to prevent any intervention. To avoid data pollution, experiments were totally independent and all VMs used for one run were completely destroyed before the next run because malware can infect other VMs and possibly pollute subsequent runs.

We collected samples at 10 seconds intervals for 30 minutes duration, so we have a total of $\simeq$ 180 samples per experiment and $\simeq$ 4500 samples in total.

E. Evaluation

We use four evaluation[8] metrics. Precision is the number of correct malware predictions. Recall is the number of correct malware predictions over the number of true malicious samples. Accuracy is the measure of correct classification. F score is the harmonic mean of precision and recall. True Positive (TP) refers to malicious activity that occurred and was correctly predicted. False Positive (FP) refers to malicious activity that did not occur but was wrongly predicted. True Negative (TN) refers to malicious activity that did not occur and was correctly predicted. False Negative (FN) refers to malicious activity that occurred but was wrongly predicted.

F. 2d CNN Results

The data collected are divided into 3 sets: training, validation and testing sets with the percentages of 60%, 20%, and 20% respectively. The split is done on the number of experiments. For example, the 25 experiments (each using a different malware) is split to 15, 5 and 5 respectively. This means that the validation and testing phases are exposed to unknown malware. Training data is used to train the CNN models. Then, the validation set is used as a way to tune the parameters of the CNN. Once we get the highest validation accuracy for a model with specific set of parameters, we use the testing set to test the chosen model (optimized classifier). The classifiers were trained for 30 epochs as it turned out, in our case, that there was no extra gain of accuracy or decrease in mean cross entropy loss afterwards.

[6]https://github.com/mahmoudaslan/researchrepo/blob/master/malwarehashes
[7]VirusTotal website. https://www.virustotal.com

[8] $Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$, $Precision = \frac{TP}{TP+FP}$, $Recall = \frac{TP}{TP+FN}$, $Fscore = 2 \times \frac{Precision \times Recall}{Precision+Recall}$

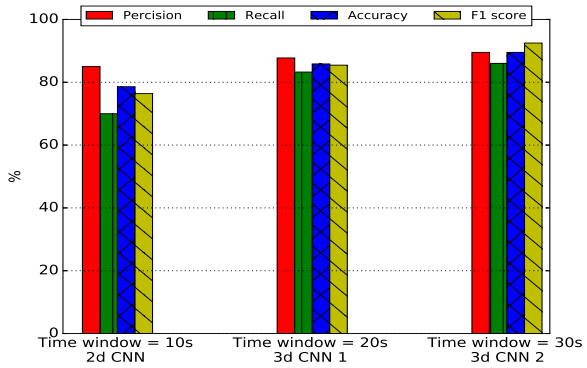


Fig. 7: Optimized 2d and 3d CNN classifiers results. 3d CNN classifiers are best optimized with learning rate of 1e-4 as well as with 20 and 30 mini-batch sizes, respectively.

We only show results for classifiers using learning rate of $1e-5$ because they showed the highest accuracy and lowest mean cross entropy loss. Figure 5 shows three trained classifiers based on different mini-batch sizes of 10, 20, and 30. Figures 5a shows the accuracy the three 2d classifiers, and similarly, Figures 5b, 5c and 5d show the mean cross entropy for mini-batch size of 10, 20 and 30, respectively.

In general, the results show that using mini-batch size of 20 yields the highest accuracy of 85.9% during validation.

Figure 6 shows the results of the 4 evaluation metrics. The CNN classifier with mini-batch of 20 shows the highest results when it is evaluated on the testing data set, while the classifier with mini-batch size of 30 shows the lowest (larger mini-batch sizes can lose generalization [29]); however, there is a drop in the overall performance of the classifiers on the testing data set where the highest accuracy is $\simeq 79\%$.

G. 3d CNN Results

The 3d CNN classifiers take time-windowed input. Samples inside this time window represent the depth of the input matrix. In fact, the 2d CNN is a special case of the 3d CNN where the depth is 1. Our experiments is done on 2 time-windows: 20 and 30 seconds. Since data is collected in 10 seconds intervals, a 10 seconds time window means 1 data sample and, similarly, 20 and 30 seconds time windows means 2 and 3 data samples, respectively.

Figure 7 shows a comparison of the performance metrics of the 2d and the newly tested 3d classifiers. These results are based on the testing data set. We refer to the classifiers as shown in Figure 7: 2d CNN, 3d CNN 1 (20 seconds time window) and 3d CNN 2 (30 seconds time window). The results showed significant improvement of using 3d CNN 1 and 3d CNN 2. The accuracy of 3d CNN 1 and 3d CNN 2 classifiers jumped to $\simeq 86\%$ and $\simeq 90\%$, respectively, as opposed to the 2d CNN classifier accuracy of $\simeq 79\%$.

V. Discussion

In this section, we discuss some relevant issues in our approach and some possible improvements for future work.

Accuracy drop between validation and test. The 2d CNN classifiers showed a drop of accuracy from $\simeq 86\%$ (validation dataset) to $\simeq 79\%$ (testing dataset). Similarly, a drop of accuracy also happened during 3d CNN classifiers evaluation (from $\simeq 97\%$ to $\simeq 90\%$ and $\simeq 89\%$ to $\simeq 86\%$). Although it might seem normal considering the validation set is biased since it is used for parameters tuning, one reason is that the malware included in the testing data set (after manual examination) is shown to have more different behavior than ones included in the training and validation set. Note also that malware which apparently has the same purpose can have different behavior which can confuse classifiers that uses malware classes information. For example, one Trojan we analyzed opens a back-door and remains idle, while another opens a back-door, steals and sends system information over the Internet. In our experiments, we randomly selected our malware from few classes (trojans, rootkits, etc..) to completely unbias our experiments.

Mislabeling problem. Using 3d CNN, we improved the mislabeling problem stated in Section I. Figure 8 shows a behavior of a malware for just 1 metric. The spike in the figure shows the time when the malware first booted up. Then, the malware keeps idle for specific time not performing any malicious activity. Labeling all samples corresponding to the benign area shown in the figure will pollute the data because the classifier learns that these actions are malicious while in fact they are not. On the other hand, when the malware steals and sends data over the Internet, samples should be labeled as malicious. Differentiating between those two actions is not possible unless it is seen by human experts. In most cases, researches take the risk of this kind of pollution because there is no way around it. A partial solution is to take both the shown areas as one sample and state that during this time window a malicious activity has happened. This is essentially what our 3d CNN classifiers are trying to do by decreasing the number of mislabeled samples as well as capturing patterns over a small time window. In theory, the larger the window the better; however, a very large time window would need a large amount of data , as well as it would act as a window of opportunity for the malware to maliciously act before detection and possible mitigation.

3d CNN's need for data. We experimented on two time windows (20 and 30 seconds) due to the limited amount of data. Increasing the time window (meaning increasing the depth of the input matrix), needs to stack multiple data samples together. Trying to experiment with 40 seconds time window and above caused dramatic decrease in accuracy because the CNNs did not have enough data to converge and learn properly. Using 3d CNN showed significant improvement with a very short time window, so having large enough data can further improve the results.

Processes and metrics ordering. One advantage of CNN is that it takes into account the spatial structure of the data; however, in our case, it seems that the input lacks spatial structure across columns and rows of the matrix. For example, if we substituted feature f_1 column with feature f_2

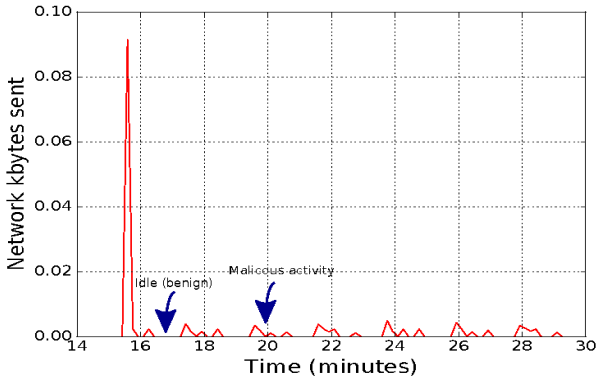


Fig. 8: Malware behavior of the *network sent kBs* metric.

column, it is still going to represent the input. On the other hand, in the case of a normal 2d image, this substitution will distort the image. The same situation is true with the rows of our input matrices when, for example, substituting process p_1 row with process p_2 row. Note that correlations might exist between the features (e.g. when CPU percent goes up, memory usage goes up as well); however, we did not use this information in our work. We believe that obtaining correlation information about the features to be used in ordering our input matrices might help with getting better results. It is true for processes as well, although it is not as easy because of the processes' dynamic nature and the possibility of newly created processes during testing time.

VI. Conclusion and Future Work

In this work, we introduced a malware detection method for VMs using 2d CNN model by utilizing performance metrics. Results showed a reasonable accuracy of $\simeq 79\%$ on the testing dataset. We noted the problem of mislabeling and we improved the performance by introducing 3d CNN model which uses samples over a time-window. It adds a 3rd dimension (depth) to the 2d input matrix representing the samples inside the defined time window. Results showed a significant improvement of accuracy of $\simeq 90\%$ for 3d CNN 2 classifier which is practically acceptable.

In the future, we plan to dedicate a pre-training step to evaluate the effectiveness of ordering the processes and features in the input matrix. We also plan to increase the scale of our experiments by using more malware binaries which will allow evaluating different time-window sizes for the 3d CNN models as well as using deeper CNN models.

Acknowledgment

This work is partially supported by NSF CREST Grant HRD-1736209, CNS-1423481, CNS-1538418, DoD ARL Grant W911NF-15-1-0518.

References

[1] B. Grobauer, T. Walloschek, and E. Stocker, "Understanding cloud computing vulnerabilities," *IEEE Security & Privacy*, vol. 9, 2011.
[2] M. Jensen, J. Schwenk, N. Gruschka, and L. L. Iacono, "On technical security issues in cloud computing," in *IEEE CLOUD*, 2009.
[3] N. Gruschka and M. Jensen, "Attack surfaces: A taxonomy for attacks on cloud services," in *IEEE CLOUD*, 2010, pp. 276–279.
[4] Z. Xiao and Y. Xiao, "Security and privacy in cloud computing," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 2, 2013.
[5] K. Dahbur, B. Mohammad, and A. B. Tarakji, "A survey of risks, threats and vulnerabilities in cloud computing," in *ISWSA*, 2011.
[6] A. Gholami and E. Laure, "Security and privacy of sensitive data in cloud computing: a survey of recent developments," *arXiv preprint arXiv:1601.01498*, 2016.
[7] M. Abdelsalam, R. Krishnan, and R. Sandhu, "Clustering-based IaaS cloud monitoring," in *10th IEEE CLOUD*. IEEE, 2017.
[8] J. Demme and et al., "On the feasibility of online malware detection with performance counters," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013.
[9] G. Tahan, L. Rokach, and Y. Shahar, "Mal-ID: Automatic malware detection using common segment analysis and meta-features," *Journal of Machine Learning Research*, vol. 13, no. Apr, 2012.
[10] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *Journal of Machine Learning Research*, vol. 7, no. Dec, 2006.
[11] T. Abou-Assaleh and et al., "N-gram-based detection of new malicious code," in *COMPSAC*, vol. 2. IEEE, 2004.
[12] A. Shabtai and et al., "Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey," *information security technical report*, vol. 14, no. 1, 2009.
[13] B. Athiwaratkun and J. W. Stokes, "Malware classification with LSTM and GRU language models and a character-level cnn," in *ICASSP*. IEEE, 2017.
[14] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *10th MALWARE*. IEEE, 2015.
[15] S. Seok and H. Kim, "Visualized malware classification based-on convolutional neural network," *Journal of the Korea Institute of Information Security and Cryptology*, vol. 26, no. 1, 2016.
[16] Z. Xu, S. Ray, P. Subramanyan, and S. Malik, "Malware detection using machine learning based analysis of virtual memory access patterns," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017.
[17] W. Huang and J. W. Stokes, "MtNet: a multi-task neural network for dynamic malware classification," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016.
[18] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi, "Malware detection with deep neural network using process behavior," in *COMPSAC*, vol. 2. IEEE, 2016.
[19] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *ICASSP*. IEEE, 2013.
[20] G. E. Hinton and et al., "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, 2012.
[21] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *Proceedings of the 30th international conference on machine learning (ICML-13)*, 2013.
[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
[23] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[24] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, 2012.
[25] P. Baldi and P. J. Sadowski, "Understanding dropout," in *Advances in Neural Information Processing Systems*, 2013.
[26] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, "On the self-similar nature of ethernet traffic (extended version)," *IEEE/ACM Transactions on networking*, vol. 2, no. 1, 1994.
[27] F. Azmandian, M. Moffie, M. Alshawabkeh, J. Dy, J. Aslam, and D. Kaeli, "Virtual machine monitor-based lightweight intrusion detection," *ACM SIGOPS Operating Systems Review*, vol. 45, 2011.
[28] M. R. Watson and et al., "Malware detection in cloud computing infrastructures," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 2, 2016.
[29] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *arXiv preprint arXiv:1609.04836*, 2016.