Using Autoencoders to Identify Kernel Level Rootkit Exploits to System Call Functions

Iram Lee
Dept. of Electrical and Computer Engineering
University of Texas at San Antonio
San Antonio, TX., USA
iram.lee@gmail.com

***Abstract*—A kernel level rootkit is a type of malware that exploits a computer system by manipulating important kernel data structures and functions, allowing an attacker to potentially gain full control. Given the elevated capabilities of a rootkit, it is a challenging task to detect and remove once a system is compromised. Inspired by the human auto-immune system, a deep learning model is proposed to provide a computer system with the intelligence to automatically recognize and aid in the recovery from this type of exploits. This is achieved by leveraging existing technologies in a novel approach. Using autoencoders and a binary file visualization technique, the proposed model is able to look at a malicious or corrupted binary, and then select its appropriate benign counterpart to indicate a suitable recovery. This method may also be used for other types of malware besides rootkits, due to the flexibility offered by the visualization technique.**

Keywords— autoencoders, rootkit, deep learning, malware, security

I. Introduction

In recent years, the use of deep learning techniques to fight against cyber-attacks has become more prominent. These methods are being used for both host-based and network-based defense systems. A wide range of applications include malware classification, malware detection, network intrusion detection, etc. As attackers develop more sophisticated malware, more intelligent defense systems are required, and deep learning provides an alternative.

An example of such advanced malware is a kernel-level rootkit. A kernel-level rootkit will try to manipulate important kernel data structures (such as the system call table) and functions by employing several methods. It is able to gain "root" access and hide processes, files, directories, and other malicious activities from system administrators. Some examples of these methods are direct kernel object manipulation (DKOM), inline function hooking, and code bytes patching.

For example, inline function hooking will place a pointer to some other malicious code inside a normal system function. The benign system function starts executing, jumps to the malicious code, then returns and finishes executing. This corrupted function will look similar to its normal counterpart, with the exception of the additional malicious code executed within itself. Fig. 1 below shows a diagram of inline function hooking.

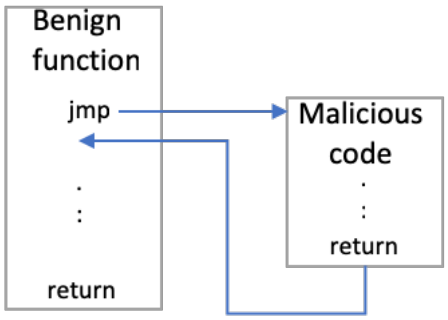

Fig. 1. Example of inline function hooking. An unconditional jump to a memory location containing malicious code is placed inside the benign function. After the malicious code is executed, it returns to the benign function and resumes normal execution.

In this paper, a deep learning (DL) model is proposed for the identification of the benign system function, given its malicious counterpart. This is attained without prior knowledge of the malicious function. A stacked Convolutional Denoising Autoencoder (DAE) and fully-connected classifier are trained with executable binaries represented as gray-scale images. This model is able to look at a malicious or corrupted binary, and then select its appropriate benign counterpart to indicate a suitable recovery.

A. Binary File Visualization as Gray-scale Images

In [1] Nataraj et al. presented a method to visualize malware binaries as gray-scale images. The binary file is read as a vector of 8-bit unsigned integers, which is then converted to a 2D array. This array can be visualized as a gray-scale image in the range [0, 255], where 0 is black and 255 is white. In our case, we represent the benign and malicious binaries as gray-scale images. These are used to train the convolutional DAE. Some of the advantages of using gray-scale images are that the binary files do not need to be executed (i.e. static analysis is performed). Also, the problem becomes an image recognition task, in which DL methods are well known to have strong capabilities.

B. Denoising Autoencoders for Image Reconstruction

A denoising autoencoder or DAE is a type of feedforward neural network that receives corrupted data and is trained to predict the original uncorrupted version of that data as its

output. The model consists of two parts: an encoder and a decoder. In this work, we think of the exploited system call function as a corrupted version of its benign counterpart.

The DAE learns by minimizing a loss function using common techniques such as gradient descent and backpropagation. Training of the DAE is performed using pairs of corrupted and uncorrupted inputs. As shown in Fig. 2 below, the encoder takes the corrupted sample $x^{\sim}$ and maps it into an internal representation $h = f(x^{\sim})$, which is used along with the uncorrupted sample x to learn a reconstruction distribution $g(h)$, the output of the decoder, by minimizing a loss function L.

One of the biggest challenges during this project was finding an appropriate dataset for model training. Due to this limitation, the dataset utilized was artificially created using data augmentation techniques commonly used in computer vision. These techniques allow generating additional samples by performing transformations on the images. Such transformations include image rotations, flips, adding random noise, etc. The executable binaries of 29 Linux system call functions were used to create the dataset by employing data augmentation.

To the best of our knowledge, this work is the first to use convolutional denoising autoencoders and binary gray-scale images for the identification of such malware exploits. By defining an exploited system call function as a corrupted version of its benign counterpart, the model will be able to identify the appropriate benign function when looking at the corrupted version. This capability can be advantageous for a run-time automatic defense system. As mentioned before, the problem is translated into an image recognition task, which makes this method flexible to be used for recovery from other types of malware as well.

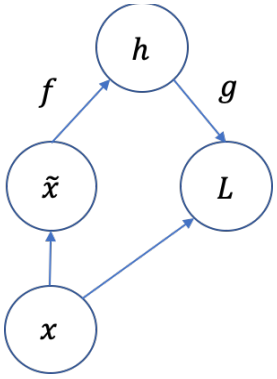

Fig. 2. Computational graph of the cost function for a denoising autoencoder, which is trained to reconstruct a clean data point x from its corrupted version $x^{\sim}$. This is accomplished by minimizing a loss function L. Source: Deep Learning by Ian Goodfellow, Yoshua Bengio and Aaron Courville.

II. Related Work

This section describes previous works that use deep learning techniques combined with gray-scale image visualizations of binary files.

Originally, L. Nataraj et al. presented the technique for visualizing malware binaries as gray-scale images and subsequently training a kNN classifier for malware classification [1]. One of the advantages noted in this work is the resilience of this technique to code obfuscation by malware authors. They also noted that the images of malware variations that belong to the same malware family show visual similarities. At the same time, they are different from malware samples belonging to other malware families. These differences and similarities make it possible to classify them as belonging to a particular malware family.

In [2] Yan et al. Created a system that uses gray-scale images from malware binaries to train a convolutional neural network (CNN) for malware classification. They combine the results from the CNN with those of a long short term memory (LSTM) model analyzing malware opcode sequences. The results of their experiments showed a detection accuracy of 99.88%. These results show the viability of using gray-scale binary images for tasks combining deep learning and malware binaries.

In their work [3] Choi et al. present a CNN model for malware detection using gray-scale images from both malware and benign files for training. Their results showed an accuracy of 96%.

Finally, in [4] David and Netanyahu use denoising autoencoders for generating malware signatures. Malware signatures are commonly employed by anti-virus software to detect infected files. Using these artificially generated signatures, a support vector machine (SVM) is trained for malware classification with an accuracy of 98.6%.

The above-mentioned works are concerned primarily with malware classification. However, they show that both malware images and denoising autoencoders have been successfully implemented in similar tasks.

III. Proposed Methodology

In this approach, the deep learning model is able to look at a malicious or corrupted binary, and then select its appropriate benign counterpart. To accomplish this, a stacked convolutional denoising autoencoder (DAE) and a densely connected classifier were used. The two models were trained separately. First, the convolutional DAE was trained using corrupted and uncorrupted image pairs. When training was complete, the convolutional DAE learned the parameters to make a reconstruction of an uncorrupted image when given a corrupted one.

The classifier was then trained using the reconstructed images generated by the autoencoder. A total of 29 classes, representing system call functions, were used for the classification task. During inference, the combined model will receive a corrupted image (i.e. the binary of a corrupted function), create a reconstruction and then classify and select the appropriate uncorrupted binary.

An overview of the model's architecture is shown in Fig. 3 below.

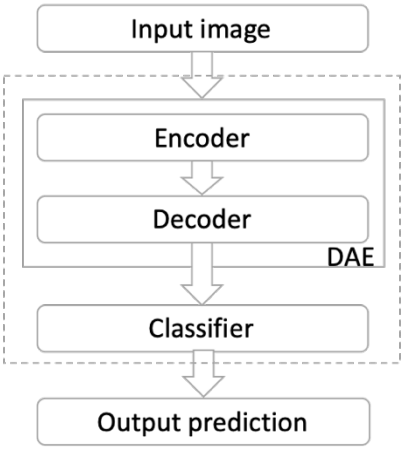

Fig. 3. Model's architecture: A classifier is attached to the output of the DAE, which consists of an encoder and a decoder.

The denoising autoencoder's underlying model is based on convolutional neural networks (CNNs), which are known to perform better with images. The encoder was implemented as a stack of convolutional and max-pooling layers, while the decoder is a stack of convolutional and up-sampling layers.

Convolution operation: the convolution operation takes a 3D input feature map and extracts patches of the map (usually 3x3 windows). The 3D feature map represents an image as a matrix with dimensions (height x width x channel), in the case of gray-scale images the channel dimension is 1 (levels of gray). Then a transformation to the patches is done by multiplying by a convolution kernel (a learned weight matrix) or filters. The patches are reassembled resulting in an output feature map. The convolutional layer is followed by a max-pooling layer.

Max-pooling operation: the pooling operation down-samples the feature map by a factor of 2. In the case of max-pooling, the operation extracts windows from the input feature maps and outputs the max value of each channel.

The net effect of the stacked convolutional and max-pooling layers is a down-sampling of the input image in which the most relevant features are extracted.

In the case of the decoder, the convolutional layers are followed by up-sampling layers. The up-sampling layer simply repeats the rows and columns of the input data by size 2 respectively. In other words, it can be thought of as the opposite of the pooling operation (without any learned parameters). This will output a reconstructed image with the same dimensions as the input image (height x width x channel).

Convolutional layers in the model use the rectified linear unit (ReLU) activation function, simply defined as $f(x) = max(0,x)$. Only the last convolutional output layer uses a sigmoid activation function, defined as $f(x) = 1 / (1 + e^{-x})$ and which maps positive values of x into the interval (0, 1).

All convolutional layers include padding, which allows us to capture all the edges of the image during the convolution operation. This is an important aspect, since we are interested in capturing all the details in the image, which in our case is the representation of a binary file. The model was trained using gradient descent (RMSprop optimizer) and the mean squared error loss function.

Fig. 4 below shows an overview of the underlying architecture of the convolutional DAE implemented in the Keras framework. The Input layer at the beginning is used to instantiate a Keras tensor, which is used by subsequent layers. It takes a gray-scale image with dimensions 32x32x1 and outputs a tensor with the same dimensions.

Finally, a fully-connected classifier was attached to the end of the convolutional DAE base. The classifier is made up of an Input and a Flatten layers followed by two fully-connected layers with a Dropout layer in between (see Fig. 5 below).

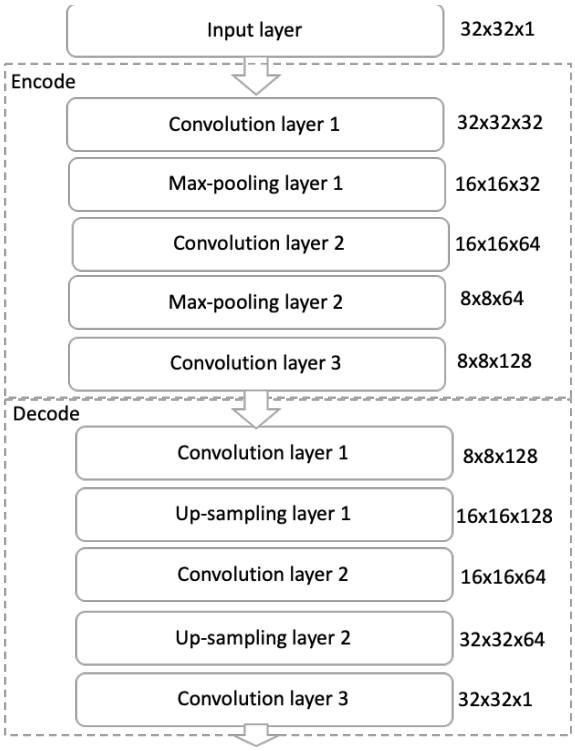

Fig. 4. Overview of the Convolutional Denoising Autoencoder. The dimensions on the right side are the shape of the output feature maps. The final output is a 32x32x1 reconstructed image.

Dropout is a common regularization technique to avoid overfitting. In this case, a dropout rate of 0.5 was used. This means that 50% of the output features will be randomly zeroed out. The activation function for the 1st dense layer is ReLU and the output layer uses a softmax activation function. Softmax will map each of the 29 output categories to a probability value (between 0 and 1). The classifier was trained with RMSprop and the categorical cross entropy loss function since we are dealing with a multilabel classification problem.

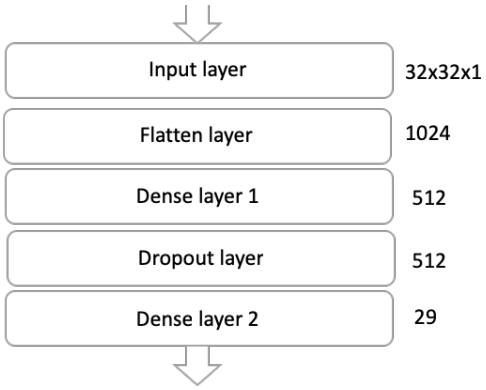

Fig. 5. Overview of the classifier model. The Flatten layer takes the 3D image array and flattens it into a 1D tensor. Dimensions on the right are the shape of the output tensors. The final layer is a 1D tensor with 29 features, which represent the selected category *(i.e. one of our 29 system functions).*

IV. Implementation and Experimental Results

A. Dataset

Due to the limited availability of data samples, the dataset was created using image data augmentation. First, a set of 29 Linux system function binaries were selected, these are listed in table I below. The binaries were collected from a system running Ubuntu server 16.04LTS.

TABLE I. List of system functions selected for classification.

No.	System call function name		No.	System call function name
1	bash		15	kill
2	chgrp		16	kmod
3	chmod		17	ls
4	chown		18	mkdir
5	cp		19	mknod
6	cpio		20	nc
7	dash		21	netstat
8	dir		22	openvt
9	dmesg		23	ps
10	fuser		24	rm
11	ifconfig		25	rmdir
12	init		26	rmmod
13	insmod		27	ss
14	ip		28	su
			29	xtables-multi

This selection was somewhat arbitrary, but depending on the application, certain system calls will be more important than others from a security perspective. After selecting the binaries, 380 copies per binary were created giving a total of 11020 binaries. A corrupted copy of each binary was then created. The final dataset is comprised of 11020 clean samples and 11020 corresponding corrupted samples. Also, label vectors were created using one-hot encoding for the 29 categories (i.e. system function names). Labels were only used for the classification task.

The dataset was further divided for each clean and corrupted as follows:

TABLE II. Dataset Partitions.

Training sets	8000
Validation sets	2000
Test sets	1020
Total	11020

B. Data Preprocessing

The binaries were first transformed into gray-scale images, with the malicious code treated as the corrupted pair of the normal uncorrupted one. The images in Fig. 6 below show an example of real binaries corresponding to the benign read() system call and the version implemented by the rootkit known as Rial.

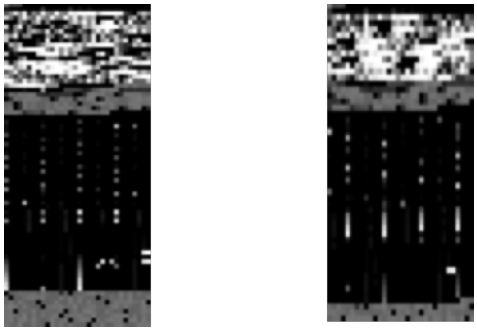

Fig. 6. Gray-scale images for the read system call (left) vs malicious read system call from the Rial rootkit (right).

For the dataset, two different methods were used to create the corrupted samples. The first approach was to introduce random Gaussian noise to the images. Some of the sample images (reshaped to 32x32 pixels) are shown in Fig. 7 below.

The second approach was to use random transformations, including image rotations, flips, etc. Sample images are shown in Fig. 8.

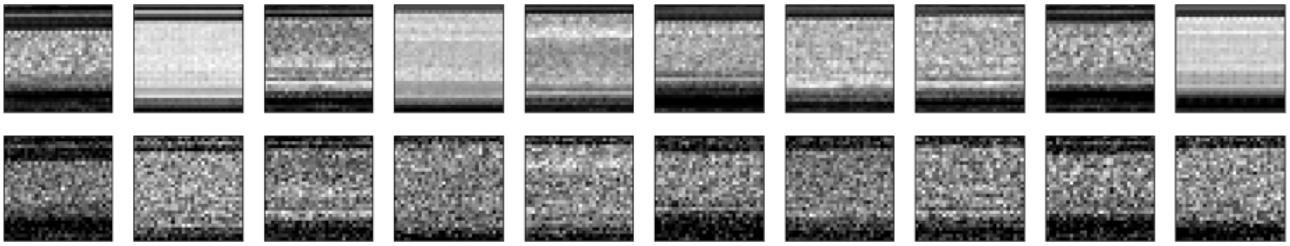

Fig. 7. Sample 32x32 pixels images generated from binaries. Top: clean images, bottom: noisy images.

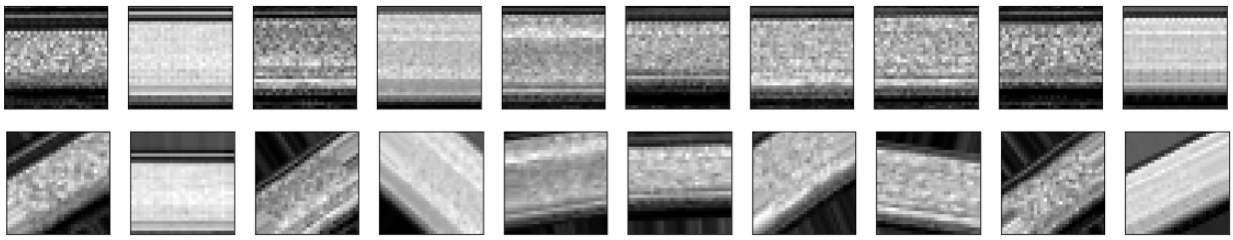

Fig. 8. Sample 32x32 pixels images with random transformations. Top: clean images, bottom: transformed images.

These two approaches were evaluated separately by using each dataset to train the model separately.

C. *Experimental Results*

The deep learning models described above were implemented in Keras and trained in a Google Cloud Platform (GCP) VM instance with 8 vCPUs. The first training and testing round was performed using the images augmented with Gaussian noise, with the following results.

The Convolutional DAE showed a validation loss of 0.0005 after 50 epochs, with the validation curve closely following the training curve. This indicates that the model was not overfitting:

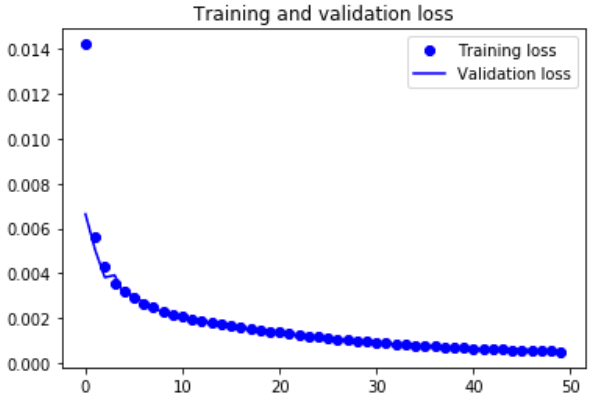

Fig. 9. Convolutional DAE training and validation loss after 50 epochs using the noisy image dataset.

Fig. 10 below shows a comparison of the clean, noisy and reconstructed images.

Next, the classifier was trained with the reconstructed images and then tested with the test dataset. Validation accuracy of the classifier reached 0.8970 after 50 training epochs. This metric indicates that in 89.7% of the cases a corrupted binary image was correctly classified or correlated to its benign counterpart.

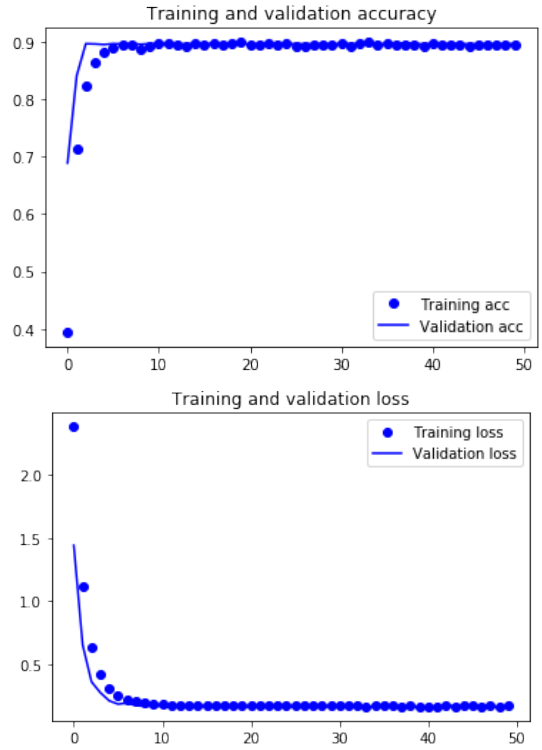

Fig. 11. Accuracy (top) and loss (bottom) of the classifier using the noisy image dataset.

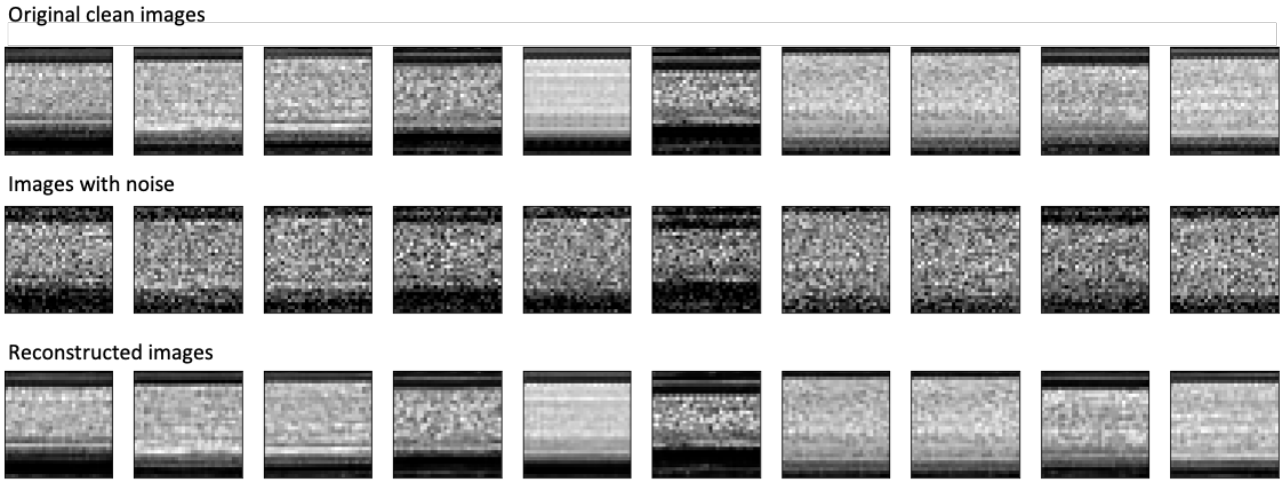

Fig. 10. The bottom images are the reconstruction output from the Convolutional DAE.

In this case, the validation curve also follows closely the training curve, indicating no overfitting. However, the accuracy appears to flatten after the first 10 epochs.

The next round of training and testing was performed using the dataset with the randomly transformed images.

Validation loss for the convolutional DAE reached 0.0020 after 50 training epochs. The validation curve closely follows the training curve, indicating no overfitting.

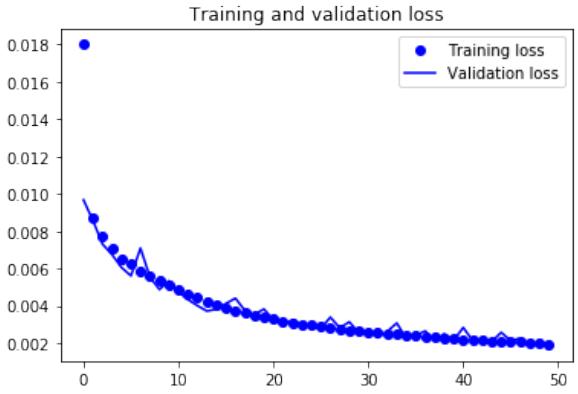

Fig. 12. Convolutional DAE training and validation loss after 50 epochs using the transformed image dataset.

Fig. 13 below shows the image comparisons for this dataset.

Finally, the classifier was trained using the reconstructed images. The validation accuracy reached 0.8745 after 50 epochs, showing the same behavior as in the previous results. Again, this metric indicates that in 87.45% of the cases a corrupted binary image was correctly classified or correlated to its benign counterpart. The curves indicate no overfitting, but the accuracy appears to flatten after the first 10 epochs.

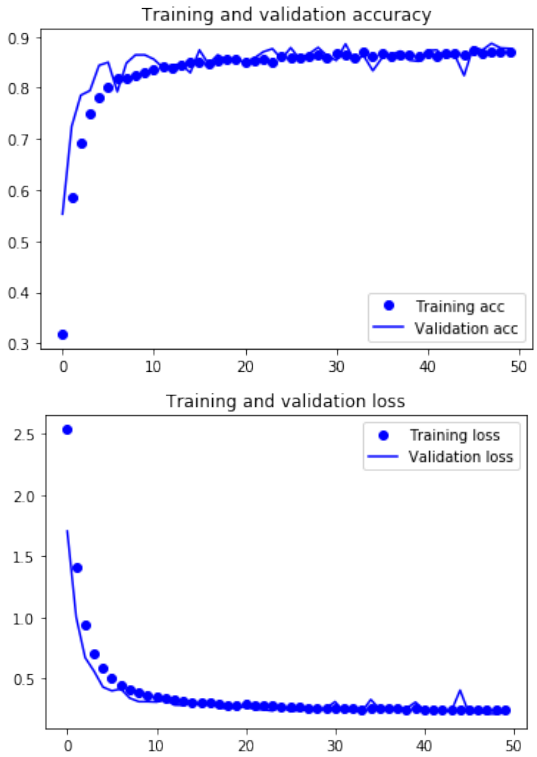

Fig. 14. Accuracy and loss of the classifier using the transformed image dataset.

In general, when using the noisy image dataset, the classifier showed a slightly better accuracy of 89.7%. Also, the convolutional DAE seems to do a better reconstruction of the noisy images.

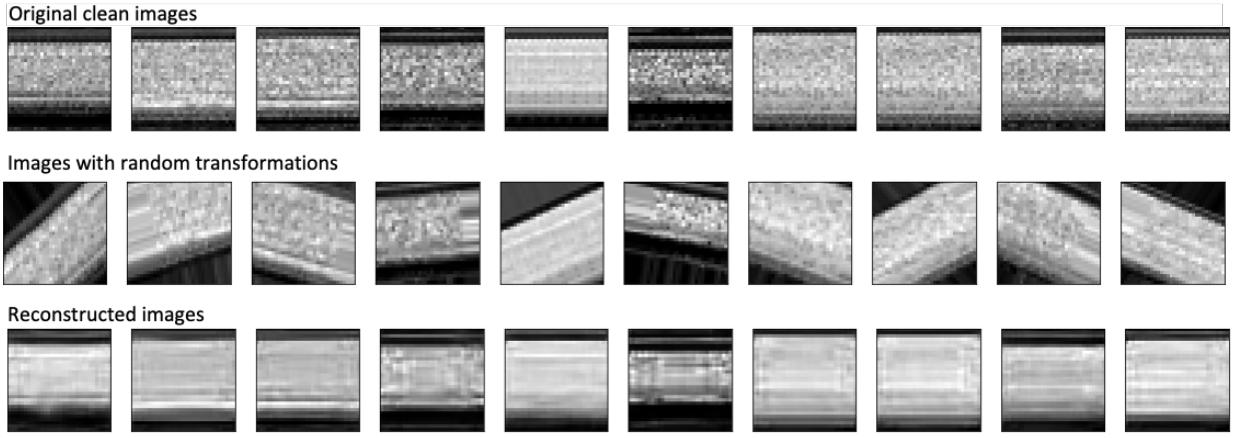

Fig. 13. The bottom images are the reconstructed output from the convolutional DAE.

V. Limitations and Future Work

As noted before, the biggest limitation of this project was the lack of an appropriate dataset. The data augmentation techniques used to construct the dataset may only provide a limited representation of "real world" data.

A proposed approach for collecting actual samples of corrupted system call functions is to implement a similar program as suggested by Levine et al. in [5]. They created a program that is able to copy the system call code directly from kernel memory and write the executable object code to a file. This is, however, a time-consuming process and therefore will be considered as part of future work.

The experiment results showed that the classifier accuracy in both instances reached a limit just below 90% after only a few epochs and showed no overfitting. As part of future work, the accuracy of the classifier will be improved by trying additional model optimizations, such as hyperparameter tuning and increasing model's capacity.

VI. Conclusions

In this paper, a deep learning model to identify kernel-level rootkit exploits to system call functions was introduced. Using a stacked convolutional denoising autoencoder and a densely connected classifier, the model is able to recognize and select the appropriate benign system call function by looking at a corrupted version of that function. The final accuracy of the model was close to 90% with a validation accuracy of 89.7%.

Lastly, the model presented in this work can be trained to be used with other types of malware. The visualization of binary files as gray-scale images offers this flexibility. This work presents a novel approach that combines existing techniques and applies them to the area of cyber-security.

References

[1] L. Nataraj, S. Karthikeyan, G. Jacob, B.S. Manjunath, "Malware images: visualization and automatic classification," International Symposium on Visualization for Cyber Security (VizSec), Jul. 2011.

[2] J. Yan, Y. Qi, Q. Rao, "Detecting malware with an ensemble method based on deep neural network," Hindawi. Security and Communication Networks. Volume 2018, Article ID 7247095, 16 pages https://doi.org/10.1155/2018/7247095.

[3] S. Choi, S. Jang, Y. Kim and J. Kim, "Malware detection using malware image and deep learning," 2017 International Conference on Information and Communication Technology Convergence (ICTC), Jeju, 2017, pp. 1193-1195

[4] O. E. David, N. S. Netanyahu, "DeepSign: deep learning for automatic malware signature generation and classification," International Joint Conference on Neural Networks (IJCNN), Jul. 2015

[5] J. G. Levine, J. B. Grizzard, P. W. Hutto, H. L. Owen, "A methodology to characterize kernel level rootkit exploits that overwrite the system call table," IEEE SoutheastCon, 2004. Proceedings.

[6] J. B. Grizzard, H. L. Owen, "On a ?-kernel based system architecture enabling recovery from rootkits," First IEEE International Workshop on Critical Infrastructure Protection (IWCIP'05), Nov. 2005.

[7] A. Bunten, "UNIX and linux based rootkits techniques and countermeasures," Unpublished.

[8] C. Salls, Y. Shoshitaishvili, N. Stephens, C. Kruegel, G. Vigna, "Piston: uncooperative remote runtime patching," Proceedings of the 33rd Annual Computer Security Applications Conference, Dec. 2017.

[9] I. Firdausi, C. Lim, A. Erwin, A. S. Nugroho, "Analysis of machine learning techniques used in behavior-based malware detection," Second International Conference on Advances in Computing, Control, and Telecommunication Technologies, Dec. 2010.

[10] A. Salem, S. Banescu, "Metadata recovery from obfuscated programs using machine learning," Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, Dec. 2016.

[11] K. Han, J. H. Lim, E. G. Im, "Malware analysis method using visualization of binary files," Proceedings of the 2013 Research in Adaptive and Convergent Systems, Oct. 2013.

[12] K. Kosmidis, C. Kalloniatis, "Machine learning and images for malware detection and classification," Proceedings of ACM Pan-Hellenic Conference on Informatics, Sep. 2017.

[13] L. Nataraj, V. Yegneswaran, P. Porras, J. Zhang, "A comparative assessment of malware classification using binary texture analysis and dynamic analysis," Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence, Oct. 2011.

[14] X. Xie, W. Wang, "Rootkit detection on virtual machines through deep information extraction at hypervisor-level," IEEE Conference on Communications and Network Security (CNS), Oct. 2013.

[15] M. Sewak, S. K. Sahay, H. Rathore, "An investigation of a deep learning based malware detection System," Proceedings of the 13th International Conference on Availability, Reliability and Security, Aug. 2018.

[16] J.Kargaard, T. Drange, A. Kor, H.Twafik, E. Butterfield, "Defending IT systems against intelligent malware," 9th IEEE International Conference on Dependable Systems, Services and Technologies, May 2018.

[17] J. Kong, Designing BSD rootkits : an introduction to kernel hacking. San Francisco, CA: No Scratch Press Inc., 2007.

[18] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning. MIT Press, 2017.

[19] F. Chollet, Deep Learning with Python. Manning Publications, 2017.