Concealed Malware Detection using Adversary-Resistant Deep Neural Networks

Michael Fuhrer | mfuhrer@vt.edu
ECE 5590 – System and Software Security – Class Project

ABSTRACT

This paper intends to address the significant security vulnerability introduced by novel adversarial, malware-obfuscation techniques, like MalGAN, which generate specially constructed noise to conceal malware from a machine-learning based malware detection engine. To accomplish this, I introduce non-differential layers into a deep neural network to attempt to harden it against gradient analysis during adversary training. Although this approach, ultimately failed, I provide insight into my process and where my implementation likely failed.

CCS CONCEPTS

• Intrusion detection systems • Neural networks • Machine learning • Malware and its mitigation •

1. Background and Problem Statement

A promising alternative to classical malware detection approaches is machine-learning detection, wherein some machine learning model is trained to detect malware. These detectors act as black boxes, where an unknown set of operations are performed on a software's features to determine whether the software likely has malicious intent. Although these models generally have a higher computational overhead than classical methods, they are particularly effective at correctly labelling new, priorly-unseen malware [1].

The approach we will focus in on in this report are neural networks, whose primary feature involves a dense interconnection of trainable linear weights and biases to transform the set of input features into some desired output, oftentimes a classifier. Because of their density, neural networks are particularly adept at modelling and understanding hidden patterns [2], allowing them to find tell-tale signs of malware more readily.

However, machine learning approaches have one severely detrimental vulnerability that could allow an attacker with the right tools to completely bypass its detection.

The objective of this paper is to use previously proposed hardening techniques to create a neural network that is resistant against the following attack.

2. Threat Model

The threat model in consideration is of an attacker seeking to obfuscate a piece of malware in such a way that a neural-network-based malware detector fails to correctly identify the malware. The attacker is assumed to have possession of a local black box copy of the detector. Although they cannot see the inner working of the detector, they can use it to test whether a piece of software is labeled as malicious or not.

The obfuscation technique considered in the threat model is dubbed as MalGAN [3]; it involves training an *adversarial* neural network to pad the malware with specialized noise, designed specifically to 'trick' the detection neural network into thinking the malware benign. It is important to note that, for an adversary to train properly, they must at a minimum have access to a black-box version of the detector neural network to train against.

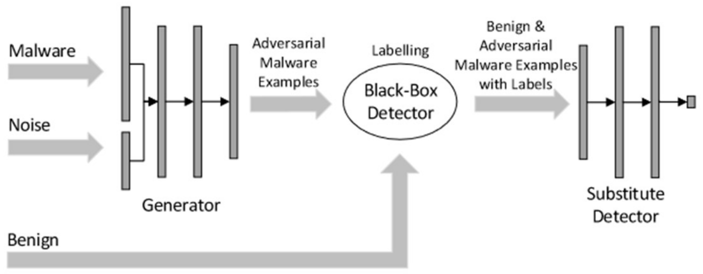
Malware
Noise
Generator
Adversarial Malware Examples
Labelling
Black-Box Detector
Benign & Adversarial Malware Examples with Labels
Substitute Detector
Benign

Figure 1. MalGAN Architecture, [3]

MalGAN, originally proposed by Hu and Tan, uses a 'generator' neural network to engineer noise to bypass the detector. The generator's inputs are a fixed-length vector representing the feature digest of an executable file and a noise vector. The generator trains by attempting to maximize the loss, i.e., maximize the inaccuracy, of the black box detector.

Because this loss must be found in relation to gradients within the training loop, it is not sufficient to only use a black box detector, as the training process cannot monitor how the black-box detector operates upon its inputs. Instead, MalGAN uses an additional 'substitute' detector, which attempts to best imitate the black-box detector. Using the gradients from the substitute detector, the generator can approximate the relationship between the inaccuracy of the black-box detector and the noise it generates.

In Hu and Tan's study, they found that their obfuscation approach was incredibly successful, changing the average true positive rate of their detectors from 93.4% to less than 1%.

3. Approach and Significance

Because MalGAN is so effective at bypassing machine-learning-based detection models, this project's objective is to survey and attempt to implement a method of hardening a neural network against an adversarial generator.

One promising approach was proposed by Wang et al. in 2017 [4]. The design principle they propose towards hardening a neural network against an adversary involves introducing a non-differential, non-invertible layer into the neural network. The key aspect of a non-differential layer is that it is difficult to calculate the gradient across the layer; this difficulty directly effects the efficacy of MalGAN. Because the MalGAN architecture must reliably model the gradient of the black-box detector using the substitute detector, making the gradient more difficult to find hinders the performance of the generator.

The neural network layer Wang et al. had the most success within their approach was a Locally Linear Embedding (LLE) transformation. This layer reduces the dimensionality of the input features while preserving the locality of data entries. In other words, it reduces the number of features available, but does so in a way that keeps similar entries close together spatially. This transformation is non-parametric, as the transformation it applies to an input is dependent upon the inputs and can therefore not be expressed in a closed form. This makes calculating the gradient across the layer difficult for adversarial generators.

Wang et al. used various testbenches to show the efficacy of their approach. Most relevantly, they used a testbench analogous to MalGAN, where they tested a neural network detector's resistance to generated obfuscated malware. Their observed true positive rate against a trained adversary went from 26.19% with their control neural network to 95.02% using their adversary-resistant model. It became my goal to see whether I could replicate their process in my project.

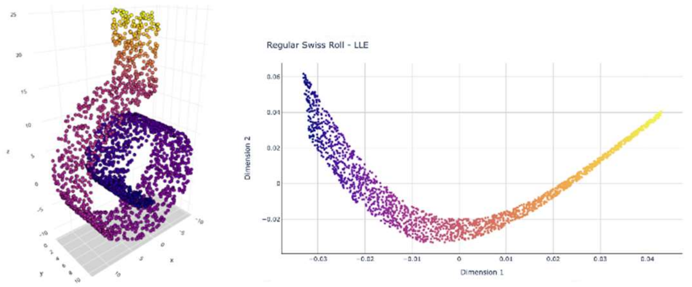

Figure 2. (Left) 3-dimensional data set before applying LLE transformation. (Right) 2-dimensional reduction of dataset after applying LLE transformation. [5]

4. Implementation

4.1 Dataset

The dataset I chose to use was SOREL-20M [6]. It was important to find a dataset that was sufficiently large and could be downloaded safely by security amateurs; SOREL-20M met both criteria. It contains ~20 million entries of both malware and benign software. Malware executables within their dataset have been 'disarmed' without affecting their digital signature, allowing them to be inspected while making accidental execution nigh impossible. This dataset also contains a file of preprocessed feature vectors, meaning it was not necessary to download any malware to train the neural networks.

Due to hardware limitations my training and testing sets contained only 50,000 and 25,000 non-overlapping samples respectively.

4.2 Feature Extraction

The feature extraction method used by SOREL-20M and, by extension, my project was borrowed from the Elastic Malware Benchmark for Empowering Researchers (EMBER) [7]. This benchmark uses a library, LIEF, to extract a wide array of features from a portable executable (PE) file. EMBER categorizes the data it extracts from an executable into nine main categories described in the appendix.

One notable consideration of this feature set is what data within the PE could be modified *without* removing any original code. As such, we can only change certain elements of the feature vector. Features like histograms, timestamps, and similar non-hashed characteristics can be fully modified. The assumption is that the generator could append noise of any length or alter parts of the PE that do not impact the execution of that file. Features which represent the amount or size of the file are marked as add only; the generator cannot remove code from the PE, only add onto it. Lastly, because certain features are hashed, our generator cannot meaningfully alter these features, as there is no way to translate the modified hashes into the characteristic that would generate such a hash. Therefore, of the original 2381-length feature vector, only 638 of those features may be modified by the adversarial generator.

4.2 Control Neural Network

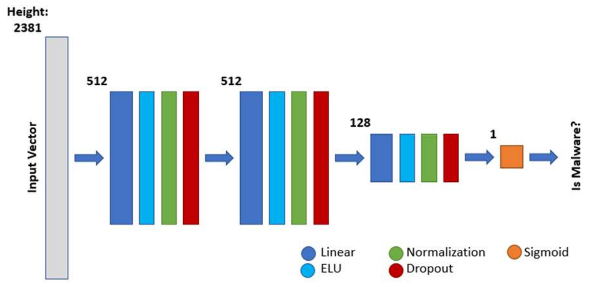

Figure 4. Architecture of control neural network detector.

With a way for the adversary to meaningfully alter a PE file's contents, I could begin training and testing the control, non-resistant neural network to detect malware. All neural networks were constructed and trained using TensorFlow. The architecture of this control neural network follows recommendations by Rudd et al. [8] for building malware classifiers. It contains three primary linear layers, each separated by an exponential linear unit, normalizer, and dropout layers.

I first trained the control neural network with non-obscured malware and benign file samples from SOREL-20M. Remarkably, my implementation's performance matched and sometimes rivaled that of other researchers, achieving an accuracy of 96.09% on the testing set. My configuration, however had a bias towards labelling software as benign; this can be seen when comparing the false positive rate of 2.49% to the false negative rate of 9.41%.

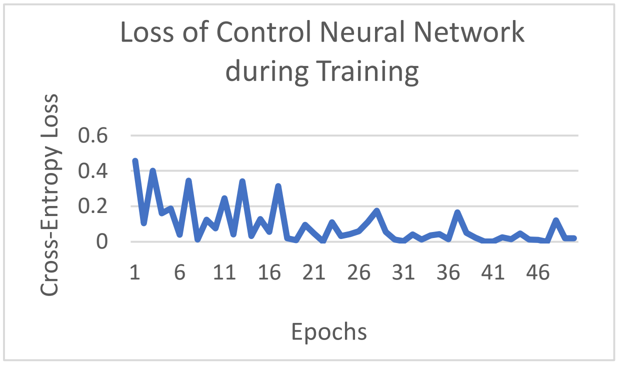

Figure 5. Cross-entropy losses of control neural network during training.

	Hu & Tang's (Multi-layer perceptron)	Rudd et al.'s	**Ours**
Accuracy	94.89%	99.67%	96.09%

Figure 6. Comparison of control neural network's accuracy against prior studies.

With the control model trained to accurately detect malware, I implemented Hu and Tan's approach to MalGAN. The generator is a much simpler feed forward neural network, which takes in a feature vector concatenated with a 100-feature long noise vector to generate a 2381-element-long vector which is then used to edit the original feature vector according to the specifications in Section 4.2. I reused the control neural network's architecture for the substitute detector.

The results from training the adversarial generator agree with Hu and Tan's findings. The generator was successfully able to bypass the control neural network's detection. The control's false negative rate went from 9.41% on non-obfuscated examples to 91.68% on adversary-obfuscated examples.

4.3 Adversary Resistant Neural Network

The adversary-resistant neural network is a modified version of the control neural network. Before inputs are passed onto the first linear layer, they are first modified with a Locally Linear Embedding transformation. During testing, I tried training the model with output dimensions of 256, 512, and 1024.

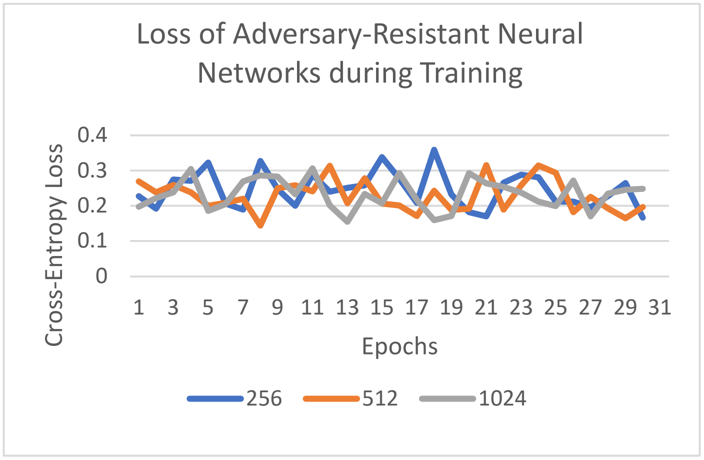

Figure 6. Cross-entropy losses of adversary-resistant neural networks with LLE reductions to 256, 512, and 1024 features.

Unlike the control, these neural networks' losses did not approach 0 over time. This suggests that reducing the feature dimensions of the input vector may remove or obscure certain features that are key for detecting malware. This is further supported by the performance of these networks during testing.

	LLE Output Dimensions		
	256	512	1024
Accuracy	94.89%	99.67%	96.09%
FNR	34.36%	28.57%	23.95%
FPR	1.65%	1.94%	2.14%

Figure 7. Performance of adversary-resistant neural networks against training set.

As the output dimensions of the LLE transformation decreases, the false negative rate of the model increases.

When trained against these models, the adversarial generators yielded the following false negative rates.

		LLE Output Dimensions		
	Control	256	512	1024
FNR	91.68%	99.39%	98.80%	99.05%

Figure 8. False negative rates of adversary-resistant neural networks compared to control against adversary-obfuscated malware.

5. Conclusion

Unfortunately, these results show overwhelmingly that my iteration of an adversary-resistant neural network failed in its goal. Not only was its baseline accuracy less than the control, but every iteration of the adversary-resistant model detected less obfuscated malware than the control.

One explanation for the discrepancy between my results and that of Wang et al.'s is the feature set used. The feature set used by Wang et al. was a much simpler; it contained binary values representing the presence of a "file system access or sequence of access events". The EMBER feature set is much more analog, with each feature containing a wider range of possible values. This likely makes LLE mapping a less reliable tool in reducing feature dimensionality, and thus key features in detecting malware are lost.

However, I believe it is still worthwhile to investigate different methods of introducing non-differentiability into a neural network, as finding one that is compatible with such a nuanced feature set like EMBER's could yield improved accuracy against obfuscated and non-obfuscated malware.

Although my primary goal was not satisfied, I am still pleased with the outcome, especially with the performance of my control neural network; I have had little prior experience with neural networks, so this was fun and challenging way to build a practical tool.

APPENDIX

A.1: Breakdown of features within an EMBER feature vector.

Byte Histogram (256 Features): Histogram of bytes within PE file. Fully modifiable – generator can append any number of bytes to impact histogram.

Byte Entropy Histogram (256 Features): Histogram of joint probability of byte value and local entropy. Fully modifiable.

String Extractor (104 Features): Extracts information about strings within PE. Limited modification: some features, like number of strings, are add only. Others, like average string length, are fully modifiable.

General File Info (10 Features): Contains information about file size, number of imports/exports, etc. Add only.

Header File Info (62 Features): Machine architecture, OS, linker, and other metadata about source. Limited modification. Size of header is add only; time stamp and version numbers are fully modifiable, hash features are non-modifiable.

Section Info (255 Features): *Hashed* section names, sizes, and entropy. Non-modifiable due to hashes.

Imports Info (1280 Features): *Hashed* table of imported libraries and functions. Non-modifiable due to hashes.

Exports Info (128 Features): *Hashed* table of exported functions. Non-modifiable due to hashes.

Data Directories (30 Features): Size and virtual address of first 15 data directories. Non-modifiable.

A.2: GitHub Repository Link.

github.com/MichaelFuhrer/ECE5900ClassProject

Please note that this repository does not contain any of the SOREL-20M dataset, nor any instances of malware. One would have to install the SOREL-20M dataset separately to properly operate this repository.

REFERENCES

[1] W. Wong and M. Stamp, “Hunting for metamorphic engines,” *Journal in Computer Virology*, vol. 2, no. 3, pp. 211–229, 2006.

[2] T. Danka, “Why are neural networks so powerful?,” Mathematics of machine learning. [Online]. Available: https://www.tivadardanka.com/blog/universal-approximation-theorem. [Accessed: 07-May-2022].

[3] W. Hu and Y. Tan, “Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN.” *arXiv preprint arXiv:1702.05983*, 2017.

[4] Q. Wang et al., “Learning Adversary-Resistant Deep Neural Networks.” *arXiv preprint arXiv:1612.01401*, 2017.

[5] S. Dobilas, “LLE: Locally linear embedding - a nifty way to reduce dimensionality in Python,” *Medium*, 05-Feb-2022. [Online]. Available: https://towardsdatascience.com/lle-locally-linear-embedding-a-nifty-way-to-reduce-dimensionality-in-python-ab5c38336107. [Accessed: 05-May-2022].

[6] R. Harang and E. Rudd, "SOREL-20M: A large scale benchmark dataset for malicious PE detection." *arXiv preprint arXiv:2012.07634*, 2020.

[7] H. Anderson and P. Roth, “EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models.” *arXiv preprint arXiv:1804.04637*, 2018.

[8] E. Rudd et al., “ALOHA: Auxiliary Loss Optimization for Hypothesis Augmentation.” *arXiv preprint arXiv:1903.05700v1*, Mar 2019.