Adversarial Malware Generation Using GANs

Final Report

Zayd Hammoudeh

1 Introduction

Malware detection is an instance of competitive game theory. Security companies develop techniques to detect existing and emerging threats. In response, malware creators design new programs that circumvent detection. Security researchers continue to investigate using machine learning to improve malware detection [1, 2, 3]. Previous research has framed malware detection as binary classification over Boolean feature vectors. These feature vectors can represent DLLs or APIs used by the malware [4] or n-gram byte models [5].

A (machine) learner is considered *robust* if its classification performance does not meaningfully degrade when noise is added to its input. Achieving learner robustness is difficult in most environments and is particularly challenging for malware detection due to an *adversary* that is specifically incentivized to develop novel techniques that circumvent state-of-the-art detectors. These attacks are particularly challenging to detect since they may be unlike anything observed to date, i.e., no training data exists with which to train the model. This usually results in reactive, rather than proactive, detection.

In their seminal work, Szegedy et al. [6] introduced the concept of *adversarial examples*, which are deliberate perturbations to an example that cause the object to be misclassified. These deceptive perturbations are specifically chosen to preserve the essence of the original object. While adversarial examples are exceptionally powerful at exploiting a learner's inherent *brittleness*, they are generally too difficult to manually construct.

Existing approaches for generating adversarial examples can be subdivided into two primary categories. First, gradient-based methods perform local search and select possible perturbations based on gradient *ascent* over some loss function. Such approaches work well for continuous valued-features, e.g., pixel values in an image. However, malware classifiers generally rely on binary feature vectors where gradient-based walks may not be possible. An alternate approach uses generative adversarial networks (GANs) which were first proposed by Goodfellow et al. [7]. This technique relies on two neural networks working in concert to learn how to fool a classifier.

The primary contribution of this work is a GAN-based adversarial malware generator. The remainder of this paper is structured as follows. Section 2 reviews previous work for generating adversarial malware. Section 3 provides an overview of our architecture. We provide architectural implementation details in Section 4. Section 5 describes the experimental setup, including the dataset used, as well as our results. Section 6 outlines future work and provides closing remarks.

2 Previous Work

The binary feature space used by malware detectors makes construction of adversarial examples particularly challenging. The requirement that the modified malware preserves the original functionality imposes an additional constraint.

In [8], al-Dujaili et al. propose an algorithm to train adversarially robust deep neural networks for malware classification. They outline four techniques for generating adversarial examples all of which rely on gradient-based methods. As described previously, the binary feature space severely hinders gradient traversal and may explicitly prevent gradient ascent.

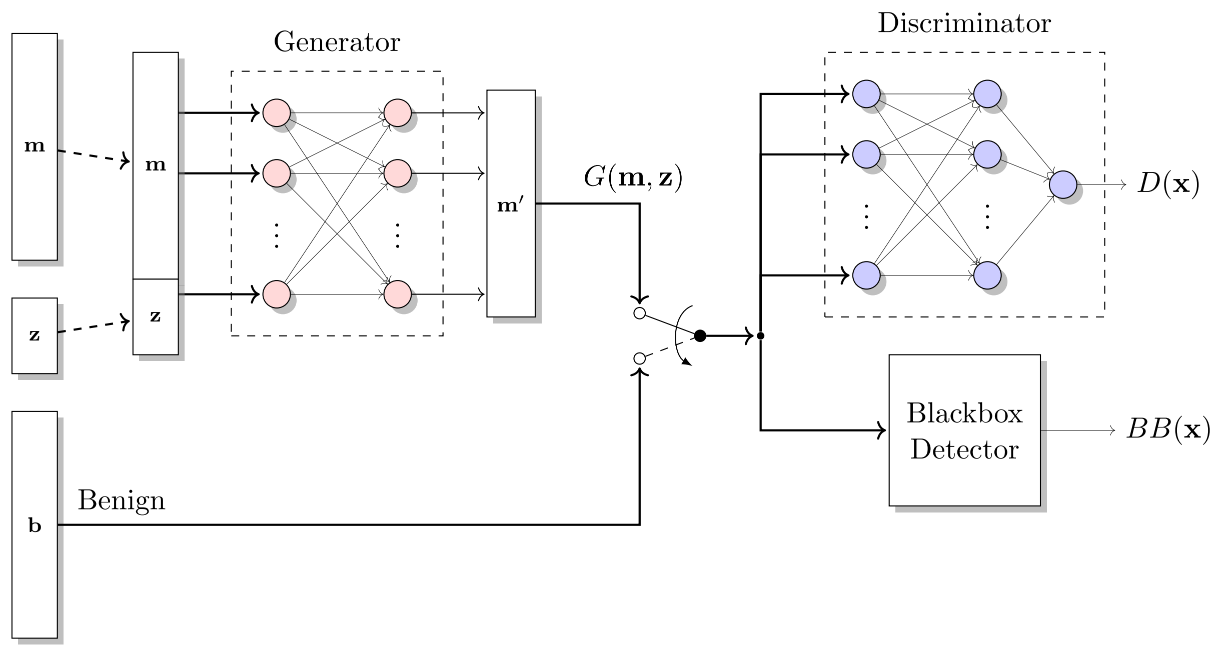

Figure 1: Adversarial malware generator's training architecture

This project implements the key ideas of Hu & Tan in [9]. Their architecture constructed adversarial malware using GANs. However, neither their dataset nor implementation has ever been released. Therefore, this work serves as a verification of their ideas. In addition, Hu & Tan provide no statistics regarding the properties of their adversarial examples.

3 Architecture of the Adversarial Malware Generator

Figure 1 shows the adversarial malware generator's training architecture. There are two primary components, which are described in the following two subsections. Section 3.3 outlines how the malware generator is trained.

3.1 Blackbox Detector

The blackbox detector, BB, is a discrete entity that is distinct from the malware generator. The detector's input is an M-dimensional binary vector $\mathbf{x}$, and its output is the predicted class of $\mathbf{x}$, i.e., malware or benign. Since the detector is a blackbox, no information about BB's underlying machine learning model(s) is assumed or used by the generator. No retraining of the detector occurs at any point.

3.2 Generative Network

The basic idea of a generative adversarial network (GAN) is quite intuitive. There are two complementary subcomponents (i.e., neural networks) — a generator and a discriminator. As their names suggest, the generator constructs new (adversarial) examples while the discriminator tries to distinguish whether a given example is malicious or benign. The following subsections describe these two blocks in greater detail.

Algorithm 1 Adversarial malware generator training

```
1:  while not converged do
2:      $\mathbf{M} \leftarrow$ Batch of malware examples
3:      $\mathbf{z} \leftarrow$ Random latent vector
4:      $\mathbf{M}' \leftarrow G(\mathbf{M}, \mathbf{z})$
5:      $\hat{\mathbf{y}}_\mathbf{M} \leftarrow D(\mathbf{M}')$
6:      Update generator with $L_G(\hat{\mathbf{y}}_\mathbf{M})$
7:      Update discriminator with $L_D(D(\mathbf{M}'), \hat{\mathbf{y}}_\mathbf{M})$
8:
9:      $\mathbf{B} \leftarrow$ Batch of benign examples
10:     Update discriminator with $L_D(D(\mathbf{B}), BB(\mathbf{B}))$
```

3.2.1 Generator

Generator, G, takes a malware program's binary feature vector $\mathbf{m}$ and perturbs it to create a new, adversarial vector, $\mathbf{m}'$, that ideally is able to deceive the blackbox detector. Randomness and diversity is injected into the perturbation process through the Z-dimensional latent vector $\mathbf{z}$ drawn i.i.d. from the uniform distribution $\mathcal{U}(0, 1)$.

The output of G's feedforward network is a vector $\mathbf{o} \in \{(0, 1)\}^M$. Recall though that each feature in $\mathbf{m}$ represents whether the program uses a particular DLL or system call. Therefore, fractional feature values are meaningless. The network addresses this by binarizing $\mathbf{o}$ into a new Boolean vector $\mathbf{g}_\theta$ via a simple threshold function. However, this binarization process is non-differentiable which would block backpropagation (i.e., neural network learning). Therefore, during training, unbinarized vector $\mathbf{o}$ is retained and used for backpropagation.

One of the constraints of our architecture is that the generated malware, $\mathbf{m}'$, must preserve all functionality in the original $\mathbf{m}$. As such, all bits that were originally one in $\mathbf{m}$ must still be one in $\mathbf{m}'$; if that were not the case, some DLL or system needed by $\mathbf{m}$ would be disabled. Hence, the generator only allows bit flips from zero to one. This is achieved by taking the element-wise maximum of the original input $\mathbf{m}$ and the binarized vector $\mathbf{g}_\theta$ as shown in Eq. (1). This behavior is identical to a bitwise-OR across each dimension, but since bitwise-OR is not a differentiable operation, it cannot be used in a backpropagation-based neural network.

$$\mathbf{m}' = \max\{\mathbf{m}, \mathbf{g}_\theta\}. \tag{1}$$

3.2.2 Discriminator

Since the malware detector is a blackbox, we can only observe a binary output, i.e., malware or benign. This binary indicator lacks sufficient information for the generator to efficiently learn how to construct adversarial examples. Instead, the role of the discriminator is to learn an approximation of the blackbox detector's decision function. The discriminator in turn provides a differentiable function the generator can use to construct a gradient for learning.

3.3 Training the Network

The architecture's GAN is trained using the procedure shown in Algorithm 1. For each iteration of the **while** loop, the generator and discriminator are first trained together on a *batch* of malware examples $\mathbf{m} \in \mathbf{M}$. Next just the discriminator is trained on a set of benign examples, $\mathbf{b} \in \mathbf{B}$.

The generator and discriminator loss functions are shown in Eq. (2) and Eq. (3) respectively.

$$L_G = \mathbb{E}\Big\{ \ln D\Big(G(\mathbf{m},\mathbf{z})\Big)\Big\} \tag{2}$$

$$L_D = -\mathbb{E}_{BB(\mathbf{x})=\text{Malware}}\Big\{ \ln D(\mathbf{x})\Big\} - \mathbb{E}_{BB(\mathbf{x})=\text{Benign}}\Big\{ \ln\big(1 - D(\mathbf{x})\big)\Big\} \tag{3}$$

These equations closely follow the standard GAN loss functions described by Goodfellow et al. in [7]. It is important to note that for an example $\mathbf{x}$, the discriminator loss function does not consider example $\mathbf{x}$'s actual class label. This is because the network is *not* trying to learn how to generate malware that looks like actual benign samples. Instead, the network's sole objective is to generate samples that fool the blackbox detector.

4 Implementation Overview

This section provides a brief overview of our architecture's implementation details. Our source code is located in [10].

The generator and discriminator are written from scratch using the PyTorch framework [11]. For maximum end-user flexibility and to guarantee efficient data collection, our implementation natively supports both CPU and CUDA execution. As closely as possible, we followed the GAN best practices compiled in [12] by notable researchers and practitioners, including the lead author of PyTorch.

Comparison with Hu & Tan Hu & Tan [9] only provide a very high-level description of their implementation and do not any provide information about where to find their source code. Despite an extensive search, we were unable to find any details on their implementation. We attempted to contact Hu & Tan in January but received no response. This eliminated any possibility of comparing two implementations.

4.1 Blackbox Detector

As explained in Section 3.1, the blackbox detector emulates a third-party malware detector and for each program yields a binary classification i.e., malicious or benign. We implemented the detector using Python's `Scikit-Learn` machine learning library [13]. The supported classifiers are: random forest, decision tree, multilayer perceptron (i.e., a feedforward neural network), and logistic regression. These are the same set of blackbox classifiers used by Hu & Tan with the exception that support vector machine (SVM) was excluded. As described in Section 5.1, our dataset is significantly higher dimension than Hu & Tan's, and `Scikit-Learn`'s SVM implementation is three to four orders of magnitude slower than the other four classification algorithms on such high dimensional data.

4.2 Generator

The generator is a feedforward neural network that modifies malware binary vectors to make them appear benign to the blackbox detector. Our generator has two hidden layers, each with 256 neurons. The hidden layers use the *Leaky ReLU* (Rectified Linear Unit) activation function defined in Eq. (4), with $\alpha = 0.01$. While we experimented with other activation functions (e.g., ELU — Exponential Linear Unit), they generally led to poor results, most likely due to the well-known *vanishing gradient* problem.

$$g(x) = \begin{cases} x & x \geq 0 \\ \alpha x & \text{Otherwise} \end{cases} \tag{4}$$

The output layer has M neurons, i.e., one for each feature in $\mathbf{m}$. These neurons all use sigmoid activations to ensure their output is within the range $(0, 1)$.

4.3 Discriminator

The discriminator is also a feedforward neural network, with a structure similar to the generator. The primary difference is the discriminator's output layer is a single neuron. Again the output uses a sigmoid activation function, but for a different reason. The discriminator's output represents the *probability* the input vector is malware so by Kolmogrov's axioms must be bounded between 0 and 1 exclusive.

Although the sigmoid function's output can never equal exactly zero or one for finite inputs, we observed that floating point errors could cause the discriminator's loss to overflow, resulting in the well-known *exploding gradient* problem. We experimented with addressing the problem via gradient clipping, but this significantly degraded the discriminator's performance. We instead chose to bound the generator's output to the range $[\epsilon, 1 - \epsilon]$. After experimenting with multiple platforms including GPUs, we empirically observed that $\epsilon = 10^{-7}$ provided the best performance without risking overflow.

5 Experimental Results

This section provides an overview of our experimental setup and results.

5.1 Dataset

In [9], Hu & Tan used a dataset of 180,000 programs, approximately 70% of which are malware. Each program is represented as a 160-bit Boolean feature vector. Hu & Tan have not published their dataset, and we wrote to Hu & Tan on January 22, 2019 requesting a copy of the dataset for benchmarking purposes. Unfortunately, we received no reply. Therefore, we are unable to perform a comparative analysis with their work.

We therefore used Al-Dujaili et al.'s [8] dataset of 54,620 programs (~65% malicious) in our experiments. Each program in al-Dujaili's dataset is represented as a 22,761-bit Boolean feature vector.

5.2 Generator Effectiveness

Similar to [9], we tested our architecture's performance across four blackbox classification algorithms, namely: logistic regression, multilayer perceptron (one hidden layer of 100 neurons), decision tree, and random forests (100 trees). In each experiment, the SLEIPNIR dataset was split into 60% train, 20% validation, and 20% test.

Table 1 lists our architecture's performance for the four learning algorithms. Each reported value represents the empirical probability that a sample from the corresponding set was classified

Table 1: Blackbox detector malware prediction rate (in percent) for the benign, malware, and adversarial sets

Learner	Benign	Malware	
		Original	Adver. Gen.
Logistic Regression	10.9	94.4	0
Multilayer Perceptron	9.2	93.5	0
Decision Tree	11.2	94.2	0.01
Random Forest	9.7	94.6	2.6

as malware by the blackbox detector. For the unaltered (original) malware examples, the blackbox detector's malware detection rate was high at >93.5%.

Observe that on the benign set, the blackbox detector also has a relatively high false positive rate around 9–12%. The detector is therefore aggressive in classifying an example as malicious, meaning the dataset is quite challenging. In contrast, the malware detection rates for adversarially generated examples is between 0–3%. This means that the adversarial examples on average look significantly more benign than actual benign set. Therefore, the malware generator effectively defeats blackbox detection irrespective of the classification algorithm.

6 Conclusions & Future Work

Generative adversarial networks are an effective tool to defeat machine learning-based malware detection. Signature-based detection may be more robust to GAN-based attacks since signatures are inherently more complex to forge than a Boolean feature vector. However, they are not immune either. Improved training of detectors is required with a specific emphasis on adversarial robustness as described in [8].

Practicality dictates the limited scope of this work, and significant opportunity for future extension remains. First, we did not study the diversity of the adversarially generated malware either independent of or as a function of latent vector $\mathbf{z}$. As with any GAN output, the potential exists for *mode collapse*. If output diversity is low, only a very insignificant modification to the blackbox detector is required for renewed malware detection. In addition, an adversarially generated malware's feature vector may have several thousand additional bits set. The generator's loss function (see Eq. (2)) could be updated to include a (decaying) regularizer that reduces the Hamming distance between the original and adversarial feature vectors.

References

[1] N. Peiravian and X. Zhu, "Machine learning for Android malware detection using permission and API calls," in *Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, ICTAI '13, pp. 300–305, IEEE Computer Society, 2013.

[2] I. Firdausi, C. lim, A. Erwin, and A. S. Nugroho, "Analysis of machine learning techniques used in behavior-based malware detection," in *Proceedings of the 2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies*, ACT '10, (Washington, DC, USA), pp. 201–203, IEEE Computer Society, 2010.

[3] J. Sahs and L. Khan, “A machine learning approach to android malware detection,” in *Proceedings of the 2012 European Intelligence and Security Informatics Conference*, EISIC ’12, (Washington, DC, USA), pp. 141–147, IEEE Computer Society, 2012.

[4] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo, “Data mining methods for detection of new malicious executables,” in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP ’01, pp. 38–, 2001.

[5] J. Z. Kolter and M. A. Maloof, “Learning to detect and classify malicious executables in the wild,” *Journal of Machine Learning Research*, vol. 7, pp. 2721–2744, Dec. 2006.

[6] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, “Intriguing properties of neural networks,” in *International Conference on Learning Representations*, 2014.

[7] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS’14, (Cambridge, MA, USA), pp. 2672–2680, MIT Press, 2014.

[8] A. Al-Dujaili, A. Huang, E. Hemberg, and U.-M. O’Reilly, “Adversarial deep learning for robust detection of binary encoded malware,” *arXiv preprint arXiv:1801.02950*, 2018.

[9] W. Hu and Y. Tan, “Generating adversarial malware examples for black-box attacks based on GAN,” *arXiv Preprint*, vol. abs/1702.05983, 2017.

[10] “Malware GAN source code.” `https://github.com/ZaydH/MalwareGAN`.

[11] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in PyTorch,” in *NIPS-W*, 2017.

[12] S. Chintala, E. Denton, M. Arjovsky, and M. Mathieu, “How to train a GAN? tips and tricks to make GANs work.” `https://github.com/soumith/ganhacks`.

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.