# Malware Generation Using GANs

# Introduction

Malware detection may be regarded as a competitive game between security firms attempting to detect threats and malware producers attempting to avoid detection (Peiravian). Previous research on malware detection has approached it as a binary classification issue using Boolean feature vectors reflecting properties such as utilized DLLs/APIs (Singh and Lakhotia) or n-gram byte models (Kolter et al.). Malware detectors, on the other hand, suffer with robustness when adversaries paid to avoid detection devise innovative assaults unlike anything seen during training. This results in reactive detection rather than proactive detection (Bruna et al.). Szegedy et al. proposed adversarial examples, which are modest changes to inputs that induce misclassification while keeping the core of the input. It is difficult to generate them, yet they are required to leverage model brittleness (Bruna et al.). Szegedy et al. proposed adversarial examples, which are modest changes to inputs that induce misclassification while keeping the core of the input. It is difficult to generate them, yet they are required to leverage model brittleness (Bruna et al.). Existing adversarial example creation techniques include gradient ascent, which suffers with binary features, and generative adversarial networks (GANs) (Goodfellow et al.). A GAN-based adversarial malware generator is presented in this work. Section 3 provides an overview of the architecture. Architecture of the Adversarial Malware Generator

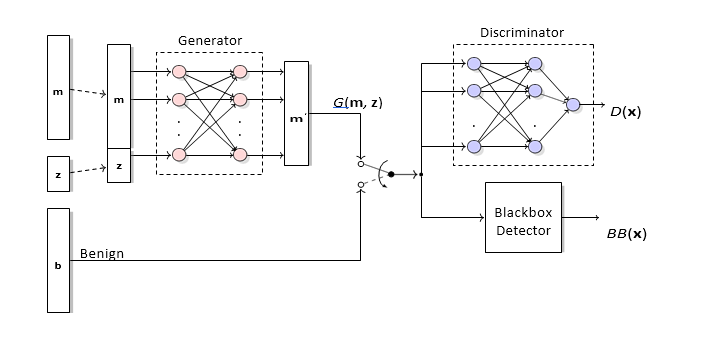


Figure 1: Adversarial malware generator’s training architecture

Figure [1](#_bookmark1) shows the adversarial malware generator’s training architecture. There are two primary components, which are described in the following two subsections.

## Blackbox Detector

The blackbox detector, BB, is a separate entity from the malware creator. The detector receives an M-dimensional binary vector x as input and returns the expected class of x, i.e., malware or benign. The generator assumes and uses no information about BB's underlying machine learning model(s) because the detector is a blackbox. At no time does the detector need to be retrained.

## Generative Network

The basic concept of a generative adversarial network (GAN) is simple. A generator and a discriminator are complementary subcomponents (i.e., neural networks). The generator, as the name implies, creates new (adversarial) instances, whereas the discriminator attempts to determine if a particular example is malicious or benign. The subsections that follow go into additional depth about these two blocks.

**Algorithm 1** Adversarial malware generator training

1: **while** not converged **do**

2: **M** ← Batch of malware examples

3: **z** ← Random latent vector

4: **M***j* ← *G*(**M***,* **z**)

5: **yˆM** ← *D*(**M***j*)

6: Update generator with *LG*(**yˆM**)

7: Update discriminator with *LD*(*D*(**M***j*)*,* **yˆM**)

8: **B** ← Batch of benign examples

9: Update discriminator with *LD*(*D*(**B**)*, BB*(**B**))

### Generator

The generator, G, in the proposed architecture takes a malware program’s binary feature vector, m, and perturbs it to create a new adversarial vector, mJ, that can deceive the detector. The perturbation process is diversified through a Z-dimensional latent vector, z, drawn independently and identically from the uniform distribution U(0, 1). The output of G’s feedforward network is a vector o ∈ {(0, 1)}M, which is binarized into a Boolean vector gθ using a threshold function. However, to allow backpropagation during training, the unbinarized vector o is retained.

A key constraint is that the generated malware, mJ, must preserve all functionality of the original m. Therefore, all bits that were originally one in m must remain one in mJ. This is achieved by taking the element-wise maximum of the original input m and the binarized vector gθ, which is equivalent to a bitwise-OR operation across each dimension. However, since bitwise-OR is not differentiable, it cannot be used in a backpropagation-based neural network. The equation for this operation is:

mJ = max(m, gθ) **(1)**

This ensures that no DLL or system call needed by m is disabled in mJ, allowing the generator to only flip bits from zero to one.

### Discriminator

Because the malware detector is a blackbox, we can only see a binary result, indicating whether it is malicious or benign. This binary indicator does not provide enough information for the generator to learn how to build hostile instances effectively. The discriminator's task is to learn an approximation of the blackbox detector's decision function. In turn, the discriminator produces a differentiable function that the generator may utilize to build a gradient for learning.

## Training the Network

The architecture’s GAN is trained using the procedure shown in Algorithm [1.](#_bookmark4) For each iteration of the **while** loop, the generator and discriminator are first trained together on a *batch* of malware examples **m** ∈ **M**. Next just the discriminator is trained on a set of benign examples, **b** ∈ **B**.

The generator and discriminator loss functions are shown in Eq. [(2)](#_bookmark7) and Eq. [(3)](#_bookmark8) respectively.

*LG* = E ln *D G*(**m***,* **z**) (2)

*LD* = −E*BB*(**x**)=Malware ln *D*(**x**) − E*BB*(**x**)=Benign ln 1 − *D*(**x**) (3)

, , , ,

These equations closely follow the standard GAN loss functions described by Goodfellow et al. in [.](#_bookmark21) It is important to note that for an example **x**, the discriminator loss function does not consider example **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.

# Implementation Overview

This section provides a brief overview of our architecture’s implementation details. Our source code is located in [.](#_bookmark24) The generator and discriminator are written from scratch using the PyTorch framework [.](#_bookmark25) 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 by notable researchers and practitioners, including the lead author of PyTorch.

# Reference

* Bruna, Joan, et al. “Intriguing Properties of Neural Networks.” Openreview.net, 24 Dec. 2013, openreview.net/forum?id=kklr\_MTHMRQjG. Accessed 12 June 2023.
* Goodfellow, Ian J, et al. “Generative Adversarial Networks.” *ArXiv.org*, 2014, arxiv.org/abs/1406.2661.
* Kolter, J. Z., et al. “Learning to Detect and Classify Malicious Executables in the Wild.” *Journal of Machine Learning Research*, 2006, www.semanticscholar.org/paper/Learning-to-Detect-and-Classify-Malicious-in-the-Kolter-Maloof/73fb1f854dd152af88835298a54b9bfbb9f711ce. Accessed 26 Nov. 2023.
* Peiravian, Naser. “Machine Learning for Android Malware Detection Using Permission and API Calls | IEEE Conference Publication | IEEE Xplore.” *Ieeexplore.ieee.org*, ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6735264. Accessed 26 Nov. 2023.
* Singh, Prabhat K., and Arun Lakhotia. “Analysis and Detection of Computer Viruses and Worms.” *ACM SIGPLAN Notices*, vol. 37, no. 2, Feb. 2002, pp. 29–35, https://doi.org/10.1145/568600.568608. Accessed 8 Aug. 2021.