Fake malware generation using HMM and GAN

Harshit Trehan[1] and Fabio Di Troia[1 [0000−0003−2355−7146]]

San Jose State University, San Jose CA 95192, USA

Abstract. In the past decade, the number of malware attacks have grown considerably and, more importantly, evolved. Many researchers have successfully integrated *state-of-the-art* machine learning techniques to combat this ever present and rising threat to information security. However, the lack of enough data to appropriately train these machine learning models is one big challenge that is still present. Generative modelling has proven to be very efficient at generating image-like synthesized data that can match the actual data distribution. In this paper, we aim to generate malware samples as opcode sequences and attempt to differentiate them from the real ones with the goal to build fake malware data that can be used to effectively train the machine learning models. We use and compare different Generative Adversarial Networks (GAN) algorithms and Hidden Markov Models (HMM) to generate such fake samples obtaining promising results.

Keywords: malware · fake malware generation · GAN · HMM · word embedding · machine learning.

1 Introduction

Malicious software, or malware in short, is a program that is specifically designed to harm computer systems by affecting devices, stealing or tampering with data, and even harming people. According to data collected by SonicWall, there were a total of 9.9 billion malware attacks worldwide in 2019 alone [30]. Thus, protection of computer systems from malware is an integral component of information security, and malware research plays an important role in securing computer systems.

To overcome these threats, machine learning techniques have been researched and applied in the malware detection domain. Their models are trained by extracting features such as opcode sequences, API calls, bytes vectors, and many other [1], [27], [33]. Although machine learning techniques have shown promising results, there are still some challenges to be taken in consideration, such as malware code obfuscation [23], the availability, or lack thereof, of large public datasets for the training phase [8], and adversarial machine learning [13] to deceive the machine learning models.

In this paper, we use mnemonic opcodes extracted from malware executable files belonging to five different malware families to generate realistic fake malware samples by implementing Generative Adversarial Networks(GAN) [9] and

Hidden Markov Model (HMM). We use multiple machine learning classification techniques, namely, Support Vector Machines, k-Nearest Neighbor, Random Forest, and Naïve Bayes Classifier to differentiate between fake and real samples and compare the two techniques (HMM and GAN) based on their performance. The main goal of this project is to develop practical use cases for fake malware opcode sequences and serve as a *proof-of-concept* for using generative modelling to synthesize mnemonic opcode sequences. An embedding step is also introduced to convert the sequence of opcodes before being used to train the classifiers. While the majority of research in this field leaned towards creating fake malware images, this work introduces the creation of fake opcode sequences comparing HMM and different GAN variants.

The remainder of this paper is organized as follows. In Section 2, we go over previous and related work. Also, we give a brief summary of the techniques and concepts that we used in this paper. In Section 3, we explain our workflow and give a description of our malware generation pipeline. In Section 4, we go over the actual implementation and our experimental setup. In Section 5, we provide the results of our experiments. Finally, in Section 6 we discuss the results and the future directions for our project.

2 Background

In this Section, we discuss the background of malware classification and the use of generative modelling. We highlight the gap in the literature with respect to generated/synthetic malware opcode samples. We also give a brief introduction to Hidden Markov Models (HHMs) and Generative Adversarial Networks (GANs). Further reading about the machine learning techniques used to evaluate our results can be found at [29], [7], [28], and [5].

2.1 Background and Related Work

A recent trend in malware research is creating images from malware executable files and using them to perform malware detection and classification. This gives the opportunity to use image-analysis techniques, and allows for the use of powerful deep neural networks which perform exceptionally well with images [35], [17].

In terms of generative modelling, many researchers used malware images to generate malware samples as that gives the advantage of boosting the dataset, and even performing data augmentation to real samples. For example, in [6] the authors adopted malware as images applying Variational Auto Encoder (VAE) and GANs to boost the malware dataset. They obtained a 2% and 6% increase in accuracy in case of, respectively, VAE and GAN. In another similar research [19], the authors used GAN and observed a 6% increase in accuracy using the benchmark ResNet-18 model trained on malware data.

Data augmentation or boosting using malware as images and generative modelling techniques is becoming increasingly popular. The drawback of this technique, though, is that converting malware files to images is computationally expensive. Moreover, training deep convolutional networks is also computationally expensive taking long time to train and test the models. Using GANs with images has similar overheads. An alternative solution is described in [12], where the authors propose a GAN based model, called "MalGAN", that is capable to bypass black-box malware detection systems whit almost 0% of detection rate. They used API features extracted from the malware samples as they are executed in a virtual environment. Despite of the impressive results, executing malware in a sandbox environment to extract the API features is again a not negligible overhead.

It is clear that there is a gap in the literature when it comes to generating malware samples using non-image features or representations of malware. Hence, we explore this gap by utilizing mnemonic opcodes extracted from malware files and generating mnemonic opcode sequences obtained by applying HMM and three different GAN architectures (see Section 2.3, 2.4, 2.5).

2.2 Hidden Markov Models

Hidden Markov Model (HMM) is a machine learning technique which is widely and effectively used for statistical analysis of time-series or sequential data. They have been successfully used in speech analysis and recognition [24], malware classification [2], and genes sequence analysis [18]. A Markov model is defined as a statistical model which has states and where the transition probabilities from one state to another are known. On the other hand, in an HMM the underlying states are not known to the observer. HMM, in fact, relies on the probability distribution of observing a set of observation symbols for each state [31].

We can use HMM to solve three Problems:

1. **Problem 1:** Given an observation sequence, $\mathcal{O}$, and a model λ, we can find $P(\mathcal{O}|\lambda)$. This means that we can compute a score for the sequence $\mathcal{O}$ w.r.t. λ [31].
2. **Problem 2:** Given a model λ and an observation sequence $\mathcal{O}$, we can determine the hidden states of the Hidden Markov Model. That is, we can uncover the Markov process underneath [31].
3. **Problem 3:** Given an observation sequence $\mathcal{O}$ and dimensions N and M, we can find the model λ of the given dimensions that best represents $\mathcal{O}$. This basically means that we are training the model to match the observation sequence [31].

The solution to these problems is implemented through the Baum-Welch algorithm [32]. In this paper, we solved all three of these problems, and more details are given in Section 4.2.

2.3 Generative Adversarial Networks

A Generative Adversarial Network(GAN) [9] model consists of two neural networks, the discriminator and the generator, which participate in a zero-sum game to achieve Nash equilibrium. The objective of the two networks is different from each other but the overall goal of the algorithm is to generate data samples that conform to a probability distribution p_g which is similar to the true data probability distribution p_{true}. The generator tries to fool the discriminator by forcing it to classify the generated samples as real, while the discriminator tries to correctly classify such samples. More information about the GAN working and architecture can be found in [9] and [3].

GAN Training When actually training the model, the loss function used is Binary Crossentropy [21] which calculates the difference in the probability distribution of true samples, labelled 1, and false samples labelled 0. The weights of both models are updated independently of each other using two loss functions on the models parameterized by their weights.

The GAN training algorithm is described in Algorithm 1. Figure 1 shows the training steps graphically.

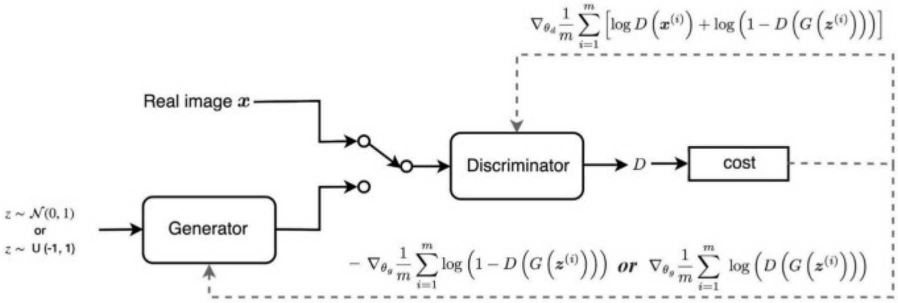
$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right]$
Real image $\boldsymbol{x}$
Discriminator
D
cost
$z \sim \mathcal{N}(0,1)$
or
$z \sim$ U (-1, 1)
Generator
$-\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$ ***or*** $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$

Fig. 1. GAN training pipeline [14].

GAN Limitations Although GANs excel in learning complex data distributions, there exist major challenges in training GANs, such as mode collapse, vanishing gradient, internal covariate shift, failure mode, and more. To overcome these problems, several novel variants and architectures of GANs have been researched and implemented. The work in [16] and [20] provide a comprehensive analysis of the challenges in GAN training and the advantages and disadvantages of various GAN architectures.

2.4 Wasserstein GAN

Wasserstein GAN (WGAN) [4] was first proposed in 2017 by M. Arjovsky et al. as an improvement over the vanilla GAN. They first published a paper [3] highlighting the important theoretical implications of GAN training as proposed

Algorithm 1 GAN Training [9].

```
Require:
  $G$: generator, $D$: discriminator, $m$: mini-batch size, $x_i$: real samples belonging to $p_t$,
  $z_i$: noise samples from known distribution $p_z$
  for number of epochs do
      Sample a minibatch of size $m$ from real data: $\{x_1, x_2, ..., x_m\}$ belonging to $p_t$
      Sample a minibatch of noise of size $m : \{z_1, z_2, ..., z_m\}$ belonging to $p_z$
      Generate fake samples using $z_i$, $i \in \{1, .., m\}$: $\{G(z_1), ..., G(z_m)\}$
      Feed $x_i$ and $G(z_i)$ to $D$ and updates it's weights using stochastic gradient ascent
          $$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x_i) + \log(1 - D(G(z_i)))]$$
      Sample a minibatch of noise of size $m : \{z_1, z_2, ..., z_m\}$ belonging to $p_z$
      Update Generator's weight by stochastic gradient descent on $G(z_i : \theta_g)$
          $$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} [\log(1 - D(G(z_i)))]$$
  end for
```

by Ian J. Goodfellow et al. [9], and outlined the mathematical reasoning and proofs for some of the issues surrounding GAN training.

WGAN working The main idea of the WGAN is that instead of optimizing the JS Divergence between two probability distributions, the use of a different distance metric as the loss function is proposed, that is, the Wasserstein distance or Earth-Mover distance. The Wasserstein distance is referred to as the Earth-Mover distance because it can be thought of as the minimum amount of energy cost required to transform the shape of a pile of dirt representing a probability distribution into the shape of another. The dirt is "transported" from one pile to another and the cost is calculated as the mass moved times the distance. More details about this approach can be found in [4].

The WGAN training algorithm is described in Algorithm 2.

WGAN limitations The main drawback of the WGAN algorithm is the way K-Lipschitz continuity is enforced [4]. Clipping the weights into a compact space $[-c, c]$ is not a very good way to enforce this constraint. It can lead to the model failing to learn more complex distributions and even saturating before reaching optimality. In fact, if the clipping parameter is large, it takes too much time for the weights to reach their limit and, thus, jeopardizing the training. On the other hand, if the clipping is small, we need to take in consideration the vanishing gradients problem.

Algorithm 2 WGAN Training [4]. Default parameter values are: $\alpha = 0.00005$, $n_{critic} = 5$ and $c = 0.01$

```
Require:
  $\alpha$: learning rate, $m$: mini-batch size, $c$: clipping size, $n_{critic}$: number of critic iterations per generator iteration, $C_w$: critic parameterized by weights $w$, $G_\theta$: generator parameterized by weights $\theta$.
  while $\theta$ has not converged do
      for t = 0, 1,.., $n_{critic}$ do
          Sample a minibatch of size $m$ from real data: $\{x_1, x_2, ..., x_m\}$ belonging to $p_t$
          Sample a minibatch of noise of size $m : \{z_1, z_2, ..., z_m\}$ belonging to $p_z$
          $C_w \leftarrow \nabla_x[\frac{1}{m}\sum_{i=1}^{m} f_w(x_i) - \frac{1}{m}\sum_{i=1}^{m} f_w(G_\theta(z_i))]$
          $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, C_w)$
          $w \leftarrow clip(w, -c, c)$
      end for
      Sample a minibatch of noise of size $m : \{z_1, z_2, ..., z_m\}$ belonging to $p_z$
      $G_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(G_\theta(z_i))$
      $\theta \leftarrow \theta - \alpha \cdot RMSProp(\theta, G_\theta)$
  end while
```

2.5 WGAN with Gradient Penalty

Wasserstein GAN with Gradient Penalty (WGAN-GP) was first introduced in 2017 by Ishaan Gulrajani et al [10]. The main objective of this architecture is to overcome the drawback of WGAN which is the way Lipschitz continuity is enforced.

To solve this, the authors in [10] propose an improved WGAN training method. They present Corollary 1 in [10] which claims that the optimal critic in WGAN has gradient norm equal to the value 1 and it is 1-Lipschitz continuous. Using this fact, a "penalty" is imposed on the critic if its gradient's norm deviates from the value 1. The training algorithm used in WGAN-GP is very similar to WGAN's algorithm minus the weight clipping part and the addition of the gradient penalty [10].

The WGAN-GP training algorithm is described in Algorithm 3.

3 Methodology

In this Section, we detail our fake malware generation pipeline, feature extraction for fake sample evaluation, and the machine learning pipeline for our experiments.

3.1 Fake Malware using HMM

The methodology adopted for generating fake malware samples using HMM is explained here:

1. Create observation sequence $\mathcal{O}$ of length $T = 30,000$ for each family.

Algorithm 3 WGAN with Gradient Penalty Training [10]. Default parameter values are: $\lambda = 10$, $n_{critic} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$ and $\beta_2 = 0.9$

```
Require:
  $m$: mini-batch size, $\lambda$: penalty coefficient, $n_{critic}$: number of critic iterations per
  generator iteration, $\alpha, \beta_1, \beta_2$: Adam parameters, $C_w$: critic parameterized by weights
  $w$, $G_\theta$: generator parameterized by weights $\theta$.
  while $\theta$ has not converged do
      for t = 0, 1,.., $n_{critic}$ do
          for i=1, 2,..., m do
              Sample $x \in p_t$, $z \in p_z$ and random number $\epsilon \in U[0, 1]$
              $\tilde{x} \leftarrow G_\theta(z)$
              Interpolated sample: $\hat{x} = \epsilon x + (1 - \epsilon)\tilde{x}$
              $L_i = f_w(\tilde{x}) - f_w(x) + \lambda(\|\nabla_{\hat{x}} f_w(\hat{x})\|_2 - 1)^2$
          end for
          $C_w \leftarrow \nabla_w \frac{1}{m} \sum_{i=1}^{m} L_i$
          $w \leftarrow \text{Adam}(w, C_w, \alpha, \beta_1, \beta_2)$
      end for
      Sample a minibatch of noise of size $m : \{z_1, z_2, ..., z_m\}$ belonging to $p_z$
      $G_\theta \leftarrow -\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} f_w(G_\theta(z_i))$
      $\theta \leftarrow \text{Adam}(\theta, G_\theta, \alpha, \beta_1, \beta_2)$
  end while
```

2. Train 21 HMM models for each malware family with $T = 30,000, N = 2$ and $M \in \{20, 21, ..., 40\}$, where M is taken as top $M - 1$ most frequent opcodes and every opcode not present in top $M - 1$ was marked as "other" or M. Section 4 explains why we chose these values for M.
3. Score these 21 HMM models for each family by testing them against samples from the other four families and benign dataset.
4. Select the best value of M, say M', from these models for each family and train 10 HMM models be setting $N = M = M'$.
5. Score the 10 models for each family.
6. Select the two highest scoring models from Step 4 and use their γ matrix to find out the most likely state sequence of the HMM model. The most likely state sequence represents the fake samples.
7. Score and evaluate these fake samples as explained in Section 3.4.

3.2 Fake Malware using GAN

We use three different GAN architectures to generate fake samples, that is, GAN, Wasserstein GAN (WGAN), and Wasserstein GAN with Gradient Penalty (WGAN-GP). The methodology adopted for generating fake malware samples using GANs is explained here:

1. Train GAN models for each family, and save generator models at an interval of 200 epochs for GAN and 500 for both WGAN and WGAN-GP.
2. Generate fake samples in batches of 32 using the saved generative models.

3. Evaluate them against real data samples by simply testing the integer vectors (Section 3.3) representing real samples and fake samples.
4. Repeat Step 4 five times and then average the results.
5. Select the best scoring model as the final generative model for each family, giving a total of five generator models per architecture.
6. The models selected in Step 6 are used to generate fake samples for each family and the samples are evaluated as explained in Section 3.4.
7. Repeat Steps 2-6 for WGAN and WGAN-GP architectures.

3.3 Feature Extraction

In this Section we explain our feature extraction process and the types of features used for evaluation. We extract three different features from the real and fake samples to train our machine learning models.

- **Normal integer vector conversion of opcodes:** We simply map the mnemonic opcodes to integers.
- **Word2Vec:** We treat the real samples as our corpus and create Word2Vec embedding of length 100 for each opcode. We use this embedding to create a vector for each data sample by simply summing up the embedding vector of each opcode in a given sample. Then, we normalize it by the length of the sample.
- ***n*-grams:** We create bigrams (n=2) from the real dataset and find the top 20 bigrams based on the frequency. Then, a vector of length 20 is created for each data sample which contains the frequency count of these 20 bigrams. We treat these vectors as our bigram features.

3.4 Evaluation

We evaluated all the HMM models by creating the Receiver Operating Characteristic (ROC) curve for each model and calculating the Area Under the Curve (AUC). For GAN, instead, we used a different approach because the most common application for GANs is in the image domain. Most researches use benchmark datasets such as MNIST, CIFAR10, and ImageNet to evaluate the performance of their GANs. A batch of generated images is saved every few hundred epochs, for instance 200, and then are visually inspected. There are two common metrics used to evaluate the quality of generated images, that is, Inception Score [26] and Fréchet Inception Distance (FID) [11]. However, we generate opcode sequences which can not be inspected visually. Hence, to evaluate our GAN models, we saved the generative model at every 200 epochs for GAN and 500 epochs for WGAN and WGAN-GP. From all the saved generative models we generated fake samples and classified them against real samples using Random Forest classifier. The model, identified by the epoch number, that gave the lowest classification results was chosen as the best generative model from that architecture, and then used for evaluation as explained in Section 4.

Accuracy, Precision and Recall To score and evaluate the quality of the fake samples (HMM and GANs), we trained four machine learning models with each of the three features (Word2Vec, Bigram, integer vectors) and calculated the Accuracy, Precision, and Recall for each model. The process is explained here:

1. Randomly sample 100 real data data samples and take 100 fake samples.
2. Extract features from real and fake samples as mentioned in Section 3.3.
3. Fit four different models, namely SVM, Random Forest, Naive Bayes classifier, and k-Nearest Neighbor on the training data using 5-fold cross validation.
4. Calculate the accuracy, precision and recall for each split done by 5-fold cross validation and use the average as the final result.

4 Implementation

In this Section, we give a detailed explanation of our dataset and the configuration of our HMM models, the different GAN approaches, and the machine learning techniques implemented for evaluation of our fake samples.

4.1 Dataset

Our dataset consists of five malware families and a benign dataset. Each malware family has over 1000 samples and the benign dataset has over 700 samples, both containing mnemonic opcode sequences. To build sich dataset, we began with the Malicia dataset [22] which has over 50 malware families, and selected WinWebSec and Zbot families since these two has more than 1000 samples each. The rest of the three families were collected from VirusShare [25]. This dataset has over 120,000 malware executables and it is around 100 Gigabytes in size, from which we selected Renos, VBInject, and OnLineGames families.

We used `objdump` which is a command line program part of the GNU Binary Utilities library for Unix-like operating systems. This program is used to disassemble executables into Assembly code and, hence, to extract the mnemonic opcodes. Specifically, such code is processed via a Python script to remove all the unnecessary information such as registers, labels, and addresses, to obtain sequences containing only the opcodes found in the code. A summary of our dataset along with each malware family's type is given in Table 1.

4.2 HMM Implementation

The HMM algorithm was implemented following the algorithm given in [31]. We wrote the code in C++, with the addition of an external Python script to preprocess our data and create the observation sequence $\mathcal{O}$ of length $T = 30,000$. We concatenated the mnemonic opcodes from different samples of a family until we reached a length of $30,000$. This was done for all five families in our dataset.

Table 1. Dataset summary

Malware Family	Type	Samples
Benign	Benign samples	706
OnLineGames	Password Stealer	1513
Renos	Trojan Downloader	1568
VBInject	Worm	2694
WinWebSec	Rogue	4360
Zbot	Password Stealer	2136

The number of unique opcodes for each family was very high and setting M to such large values makes training of HMM models computationally infeasible. Thus, we experimented with selecting the top n most frequent opcodes from the observation sequence, where $n \in \{20, 21, ..., 40\}$. The value n is represented as the parameter M in HMM, and its optimal value for each family served as the dimensions of our HMM model in the next set of experiments ($N = M = M'$).

Afterwards, we solved Problem 2 of HMM to find the most likely state sequence which will act as our fake malware samples generated using HMM. For each family, our model dimensions were $N \times M$, where $N = M = M'$ and M' was the best value of M for each individual family.

We trained ten different HMM models, each with 5000 random restarts for each malware family. All ten of these models were scored the same way as explained above, using 500 true samples and 500 false samples. Out of these ten models, we selected the two best ones with the highest AUC value. The γ matrix from these two models was used to find the most likely hidden state sequence. Each model gives us a sequence of 30,000 length. Finally, we divided this sequence into 50 “fake” samples of length 600 each. This gives us a total of 100 fake samples per family.

4.3 GAN Implementation

We implemented all three GAN architectures in Python using TensorFlow and Keras with TensorFlow backend. For GAN, we used Adam optimizer with the following parameters:

$$Adam(lr = 0.0003,\ \beta_1 = 0.5,\ \beta_2 = 0.99)$$

These parameters gave the best results and, thus, they were chosen. The loss function used was Binary Crossentropy as it is equivalent to the loss function for GAN. The models were trained for 10000 epochs.

For GANs, the use of Batch Normalization [15] layer is recommended as the training is done using minibatches of data. The variance in the input data implicitly caused by minibatches slows down training and requires the use of very small learning rates, otherwise the gradients and weights of layers may change drastically from minibatch to minibatch. For the discriminator we have one input layer, two fully connected hidden layers, and an output layer with just one neuron. The activation function for the output layer is Sigmoid since we are using Binary Crossentropy loss, and Sigmoid gives a value between $[0, 1]$ which is interpreted as the score for a sample or the probability. The activation function for the hidden layers is LeakyReLU. LeakyReLU is recommended over ReLU because ReLU outputs 0 for all negative inputs which causes vanishing gradients problem. LeakyReLU has the hyperparameter α which is used to scale negative outputs. We used $\alpha = 0.2$ for our experiments. LeakyReLU activation function is:

$$f(x) = \begin{cases} \alpha x & x \leq 0 \\ x & x > 0 \end{cases} \tag{1}$$

The generator has one input layer, three fully connected hidden layers with a batch normalization layer after every hidden layer, and finally an output layer with 600 neurons, which is the length of the opcode sequence we want to generate. The activation function for hidden layers is, again, LeakyReLU, and for the output layer we used TanH. We scale all of our inputs between $[-1, 1]$, and TanH also gives an output between that range, which is what we expect from the generator. We experimented with different layers for both networks, including Convolutional 1D layers, and fully connected Dense layers had the best performance.

GAN Stabilizing Techniques We further utilized stabilizing techniques to improve GAN training. All the techniques are discussed in [26] which was published in 2016 by some of the co-authors of the original 2014 paper on GANs [9]. The techniques were Minibatch Discrimination, Label Smoothing, and Label Switching.

4.4 WGAN Implementation

For WGAN, we used RMSProp optimizer. RMSProp is recommended by the paper authors in [4] because the training was more stable for RMSProp as compared to Adam which is momentum based. The learning rate chosen is also a small value:

$$RMSProp(lr = 0.00001)$$

The architecture of our WGAN is the same for the critic and the generator, except the input and output layers. We trained each WGAN model for $100,000$ epochs using minibatches of data.

The actual models are compiled and trained separately for the critic and generator. For the generator, we have the same activation function for hidden layers (LeakyReLU) and output layer (TanH). For the critic, however, we used no activation function or used linear activation in the output layer. This approach allows the loss function to be computed easily when implementing the WGAN algorithm given in [4]. These layers and networks gave the best result, hence, we chose these as our final networks.

4.5 Wasserstein Distance

The loss function or the Wasserstein distance between real and fake samples can be written as follows:

$$\text{Critic loss} = \text{critic's avg. real samples score - critic's avg. fake samples score}$$
$$\text{Generator loss} = \text{- critic's avg. fake samples score}$$

This interpretation is correct because we want the critic network to learn the K-Lipschitz function that will calculate the Wasserstein distance. We are only concerned with the output of the function and not actually knowing the function. Assuming the network has learnt the correct function, we can interpret the Wasserstein distance as the loss given above.

Since neural networks use stochastic gradient descent they seek to minimize the loss values. For the generator, minimizing the loss value will mean that the critic will be encouraged to score the fake samples higher. For example, a score of 5 on fake samples will mean -5 loss for the generator and a score of 10 will mean -10 loss. For the critic, in order to minimize the loss, the score for real samples will be encouraged to be small. This will maximize the distance between the generated and fake samples and at the same time minimize both losses. This is implemented simply by using no activation function in the output layer for the critic and using -1 label for fake samples and +1 for real samples.

4.6 WGAN with Gradient Penalty Implementation

For WGAN with Gradient Penalty, we used Adam optimizer. Unlike WGAN, momentum based optimizers seem to work well for WGAN-GP. The parameters for the optimizer were:

$$Adam(lr = 0.0001,\ \beta_1 = 0.5,\ \beta_2 = 0.9)$$

We trained each WGAN-GP model using minibatches for $100,000$ epochs. We decided to use Convolutional 1D layers for the models because using fully connected Dense layers had worse performance as compared to Conv1D layers. In the critic network, we used three hidden Conv1D layers with 64, 128, and 256 filters and filter size 3. In the generator network, we also used three Conv1D layers with 64, 32, and 16 filters, and filter size 3. The activation functions for the hidden Conv1D layers is again LeakyReLU.

The output layer of the generator is a fully connected Dense layer with 600 neurons, and the activation function is again TanH. Similar to WGAN, the output layer of the critic network has no activation function because we still need to calculate the Wasserstein loss/distance. In the actual implementation, the calculation of the penalty is very straightforward. Using Keras, we get the gradients for the critic network with respect to the averaged/interpolated image. Then, we calculated the Euclidean norm, or L2-norm, of these gradients. Finally, the penalty was calculated as $(1 - L2\_norm)^2$. The authors in [10] advised against the use of Batch Normalization in the critic network. They suggested that, if required, Layer Normalization could be used. We experimented with Layer Normalization but the performance degraded, hence, we decided not to implement it. For the generator, we still used Batch Normalization layer.

We used $\lambda = 10$, that is, the penalty coefficient, and the parameter n_critic = 7, that is, the number of critic iterations per generator iteration. Additionally, after every 500 epochs, we trained the critic for 100 iterations and, then, updated the generator. This is a common practice to train the critic to an optimal value which is the aim for WGAN and WGAN-GP. This allows for exact Wasserstein distance calculation instead of an approximation and, therefore, the generator receives the correct gradient updates to converge properly.

5 Results and Discussion

In this Section, we discuss and present the results of our experiments.

5.1 HMM Results

The first set of experiments were conducted to determine the optimal value of M for each family. Then next set of experiments were conducted to train the best HMM models which were used to generate fake malware samples. The summary of the results and the best value of M chosen for each family is given in Table 2.

Table 2. Best value of M for each family

Malware Family	AUC Score	Best value of M
WinWebSec	0.911	22
Zbot	0.840	20
Renos	0.815	22
OnLineGames	0.867	22
VBInject	0.889	25

We fixed these values of M for the rest of our experiments. For HMM models to generate fake samples by solving Problem 2, we fixed the dimensions as $N = M$, where M is the best value for each family as listed in Table 2.

Our next experiments consisted of training ten different HMM models with dimensions as mentioned above and choose the two best models out of ten. We chose the two highest scoring models and calculated their most likely hidden state sequence using the γ matrix from the models. After breaking the two γ matrices of 30,000 length each into 100 samples of length 600 each, we tested these fake samples against real samples as explained in Sections 3.3, 3.4. Due to low accuracy, precision, and recall scores, the model was not able to differentiate between real and fake samples. Results from each of the four algorithms are given in the following Section.

HMM Classification Results We first performed hyperparameter tuning for the four machine learning algorithms and fixed the best parameters for the rest of the experiments.

1. **SVM:** Grid search on the values of C, kernel, and degree with ranges: $C \in \{1, 2, \ldots, 10\}$, $kernel \in \{rbf, poly, linear\}$, and $degree \in \{2, 3, 4, 5\}$. We found that polynomial kernels were overfitting the data, hence, the final parameters for SVM were $C = 5$ and kernel=rbf.
2. **Naïve Bayes:** No hyperparameter tuning required for Naïve Bayes classifier.
3. **Random Forest:** Grid search on the number of decision trees to use, and maximum depth of trees, with ranges: number of trees $\in \{10, 20, \ldots, 80\}$, max depth of trees $\in \{2, 3, \ldots, 10\}$. We found that using 50 decision trees with max depth of 5 performed best without overfitting the real malware samples.
4. ***k*-NN:** Grid search on the number of neighbors to consider (k) with range: $k \in \{4, 5, \ldots, 20\}$. The value $k = 8$ worked well, and the distance metric chosen was Euclidean.

We used 5-fold cross validation and the scores given are the average scores from 5-fold cross validation. By using Word2Vec features, SVM, Random Forest, and k-NN classifiers, we were able to differentiate between real and fake samples efficiently. Especially SVM with accuracy, precision, and recall equal to 1.00 for all the families, except Zbot with 0.97, 0.99, and 0.95, respectively. However, Naïve Bayes classifier had low recall rates for Zbot (0.73) and OnLineGames (0.76). We attribute this result to the ineffectiveness of the classifier rather than the quality of fake samples.

When Bigram features were applied, all four classifiers were able to differentiate between real and fake samples very effectively with accuracy, precision, and recall in between 0.97 and 1.00, with the only exception of OnLineGame with accuracy and precision rates equal to 0.96 and 0.93 when Naïve Bayes classifier and k-NN were used.

Finally, by using integer vectors, the metrics rates were less consistent, varying between 0.59 and 1.00, with particularly poor results when Naïve Bayes and k-NN classifiers were used. We attribute these low scores to integer vectors being a weaker feature representation for the data.

5.2 GAN Results

We experimented with the stabilizing techniques mentioned in Section 4.3. Although the training stabilized across all five families using these techniques, the results improved for Zbot, Renos, and VBInject but got worse for WinWebSec and OnLineGames. This is a common phenomenon when training GANs. The loss values for the discriminator and generator do not necessarily indicate or correspond to the model's performance or quality of the generated samples. Fake samples were generated using the best chosen models in batches of 32 since that was the batch size during training. Generating samples in same batch sizes as the training size, generally, gives better results.

We used the same hyperparameters as discussed in Section 5.1, and tested the fake samples using all three features mentioned above.

Using Word2Vec and Bigram features, the scores for all four families dipped a little as compared to the HMM results. SVM and Random Forest reached accuracy, precision, and recall above 0.90 for these two features, except for OnLineGames with 0.88 precision with Random Forest. Low precision rate means high false positive rate which was the most desirable result for us. Naïve Bayes had low overall scores for Word2Vec and Bigram features on account of it being a weaker classifier. Interestingly, k-NN obtained the lowest overall scores for these two features. This can be attributed to the way k-NN algorithm works and that the generated data distribution is slightly closer to the real data distribution as compared to HMM fake samples.

For integer vectors, we found that all four classifiers were not able to effectively differentiate between real and fake samples. As seen with the previous experiments, integer vectors are a weaker feature representation but the difference in results between HMM integer vector classification and GAN integer vector classification does suggest that the GAN models were able to perform better than HMM. For k-NN and Renos, the precision and recall are 0%, which means that the model was not able to distinguish between fake and real at all based on just the integer vectors.

5.3 WGAN Results

Unlike GAN, the loss values when training WGAN gave reliable information about the model's progress and convergence. Hence, for WGAN and WGAN-GP, we first discussed the loss curves and convergence and, then, gave the classification results for the four machine learning techniques.

Convergence and Loss Values The loss value for the critic and the generator converged very fast in the first few epochs and, then, stayed the same for the remaining epochs. We tried a lot of different hyperparameters, such as changing the value of "n_critic", different clipping value, and different learning rates. Even changing the networks entirely and using Convolutional 1D instead of fully connected Dense layers did not help. The value of loss did not change after the first few epochs. This shows that clipping the weights is a major drawback in WGAN (Section 2.4) as it saturates the model, and the weights do not update after a point. Any change in weight is nullified by the clipping step. Interestingly, all four families converged to the same loss value for the critic and generator.

The clipping step stops the training since the weights can not change beyond the clipping range and do not respond to the gradient updates that are back propagated through the network.

WGAN Classification Results The best generative model from WGANs was chosen independently for each family. We used the same hyperparameters as discussed in Section 5.1, and tested the fake samples using all three features in batches of 32.

Using Word2Vec and Bigram features, SVM and Random Forest were able to effectively differentiate between real and fake samples generated by WGAN, with accuracy, precision, and recall ratios between 0.96 and 1.00 for all the families. Interestingly, even Naïve Bayes and k-NN performed well, even though we found from the previous results that they were the two weaker classifiers. This means that the WGAN fake samples were of inferior quality compared to HMM and GAN.

Using integer vectors, the results for SVM and Random Forest were high (in between 0.85 and 1.00), but not as effective as the ones obtained with Word2Vec and Bigram features. Again, integer vectors proved to be a weak feature representation that makes classification hard. For k-NN and Naiïve Bayes with integer vectors, we obtained extremely low recall rates for some families, such as 0.25 for VBInject, 0.37 for OnLineGames, and 0.53 for Renos. However, these low recall rates were accompanied by high precision rates of almost 1.00 across all the families.

5.4 Wasserstein GAN with Gradient Penalty

As with WGAN, the critic's loss value helps monitor the model's performance for WGAN-GP. The WGAN-GP paper [10] mentions that the the critic's loss should start at a large number and then converge towards zero. The generator's loss is not very insightful and can fluctuate. Thus, first we discuss the loss curves and then give the classification results.

Convergence and Loss Curves The loss curves for all five families showed a similar shape, with the start value for the critic that started at around -28 and then slowly converged to around -4. This is the expected behavior and means that our model was training properly.

The critic loss curves for the other four families also showed similar shapes but with slightly different values of convergence. Training the models for more epochs, around $200,000 - 300,000$, would be ideal for full convergence.

The loss curve for the generator was not very informative about the model's performance and training, as the loss values kept oscillating.

WGAN-GP Classification Results The best generative model from WGAN-GPs was chosen independently for each family. We used the same hyperparameters as discussed in Section 5.1, and tested the fake samples using all three features in batches of 32.

Using Word2Vec and Bigram features, all four machine learning techniques were not able to give very good classification results. Compared to WGAN and GAN, the metrics rates were much lower (in between 0.77 and 1.00). This means that the quality of the fake samples generated by WGAN-GP generative models is better as compared to WGAN and GAN. The most surprising result is the dip in Random Forest's classification. Random Forest is one of the better classifiers out of the four classifiers that we used. For Zbot, Renos, and VBInject the overall accuracy for Random Forest was around 0.70. For WinWebSec and OnLineGames, the accuracy was also low at 0.82 and 0.81 for Word2Vec, respectively, and even lower for Bigram features at 0.81 and 0.74. This is a promising result since we saw that classifying real and fake samples using these two features was very effective, getting high accuracy and precision scores previously.

Using integer vector features the scores for SVM, Naïve Bayes, and k-NN classifiers were very low (in between 0.00 and 0.90). These three models were not able to distinguish between real and fake samples just based on the integer representation. This was confirmed by accuracy scores in range of 0.50 and 0.60, and even lower for Naïve Bayes at less than 0.50 for WinWebSec, Zbot, Renos, and VBInject families. Random Forest did a better job as compared to the other three techniques but the accuracy was still around 0.70 for WinWebSec, Zbot, Renos, and around 0.60 for OnLineGames and VBInject. This again showed that the quality of fake samples generated by WGAN-GP generative model was much better than the other GAN architectures and HMM.

Table 3. SVM-WGAN with Gradient Penalty scores for each feature.

Malware Family	Features								
	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	0.88	1.00	0.81	0.78	0.71	0.93	0.58	0.62	0.46
Zbot	0.84	1.00	0.70	0.89	1.00	0.79	0.69	0.71	0.64
Renos	0.88	1.00	0.81	0.94	0.91	0.98	0.41	0.38	0.42
OnLineGame	0.86	0.97	0.79	0.77	0.77	0.88	0.48	0.49	0.50
VBInject	0.81	0.89	0.76	0.84	0.77	.97	0.44	0.37	0.37

5.5 Comparison of the Results

The complete results for the WGAN-GP experiments as summarized in Table 3, 4, 5, and 6. The results of the other approaches can be found in [34]. In Figure 2, we compare the four different approaches by computing the average

Table 4. Naïve Bayes-WGAN with Gradient Penalty scores for each feature.

Malware Family	Features								
	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	0.81	0.83	0.86	0.95	1.00	0.91	0.42	0.48	0.45
Zbot	0.79	0.89	0.73	0.92	0.93	0.91	0.35	0.35	0.34
Renos	0.81	0.91	0.76	0.89	0.93	0.88	0.34	0.37	0.47
OnLineGame	0.76	0.81	0.73	0.89	0.86	0.94	0.62	0.63	0.68
VBInject	0.83	0.90	0.79	0.81	0.80	0.80	0.44	0.47	0.53

Table 5. Random Forest-WGAN with Gradient Penalty scores for each feature.

Malware Family	Features								
	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	0.82	0.85	0.85	0.81	0.86	0.79	0.72	0.73	0.72
Zbot	0.77	0.84	0.75	0.81	0.87	0.79	0.78	0.79	0.77
Renos	0.78	0.84	0.76	0.83	0.87	0.85	0.72	0.73	0.70
OnLineGame	0.81	0.87	0.79	0.74	0.78	0.81	0.63	0.63	0.73
VBInject	0.74	0.75	0.82	0.82	0.81	0.85	0.60	0.71	0.54

Table 6. *k*NN-WGAN with Gradient Penalty scores for each feature.

Malware Family	Features								
	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	0.68	0.84	0.56	0.79	0.96	0.65	0.50	0.00	0.00
Zbot	0.79	0.97	0.64	0.82	0.91	0.78	0.55	0.60	0.9
Renos	0.86	1.00	0.74	0.75	0.82	0.71	0.57	0.53	0.18
OnLineGame	0.81	0.89	0.78	0.82	0.87	0.81	0.70	0.90	0.48
VBInject	0.86	1.00	0.74	0.77	0.97	0.61	0.49	0.27	0.10

accuracy per malware family over the three different embeddings. We can see that the sequences generated by HMM and WGAN techniques are the ones more easily detected, that is, their generated fake malware is not confused with the real malware data. GAN obtains better results but they are not far from the previous ones. WGAN-GP, on the other hand, is the approach that clearly shows its potential in confusing the classifiers. In fact, the average accuracy obtained with the four classifiers is consistently poor. This shows the difficulty in detecting the fake WGAN-GP data from the real one.

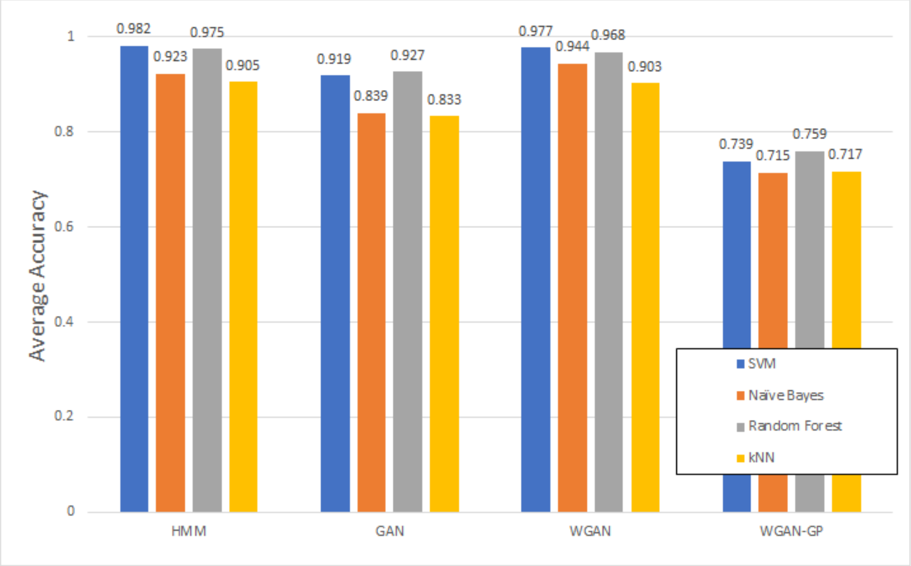

Fig. 2. Comparison of the results

.

6 Conclusion and Future Work

In this paper, we aimed at utilizing different generative modelling techniques to generate fake malware mnemonic opcode sequences. We utilized four different techniques, that is, Hidden Markov Models (HMMs), Generative Adversarial Networks (GANs), Wasserstein Generative Adversarial Networks (WGANs), and Wasserstein Generative Adversarial Networks with Gradient Penalty (WGAN-GP).

Previous work has shown that when malware are converted to images for detection, GANs can be utilized successfully to generate fake malware images. However, there is a gap in the literature when it comes to generating malware opcode sequences. Converting malware to images and training GANs on images has an exceeded overhead. Opcode sequences, instead, provide a simpler representation of the data.

We used three different feature extraction techniques to generate the malware opcode sequences, namely, Word2Vec, Bigram, and integer vectors. Classification results showed that Word2Vec and Bigram features gave a better representation of the malware data since for all four generative models the classification results were superior. Integer vectors, on the other hand, do not capture the true distribution of the real malware samples. This is confirmed by the comparatively lower classification scores when tested using integer vectors.

Fake samples generated by HMM were quite effectively distinguishable by SVM, Random Forest, and k-NN classifiers. Especially by using Word2Vec and Bigram features, these three classifiers obtained accuracy above 0.90 for all five of the tested families. Naïve Bayes classifier, instead, had much lower scores with any of the three feature extraction techniques. This implies that Naïve Bayes is a weaker classifier for malware classification using opcode sequences.

Using generative models from GAN, we saw a slight improvement in the results with the fake malware being confused in larger number with the legitimate ones. For WGAN, the results were instead not promising. In fact, the classifiers were able to identify the fake malware samples with scores close to the ones obtained in the HMM experiments. This was attributed to the weight clipping step in the WGAN algorithm, that inhibits the critic network's ability to properly learn the real data's representation. However, for WGAN-GP we got the best results. We saw that the classification outcome was now relatively poor, even when the more informative Word2Vec and Bigram features were applied. In fact, for all four classifiers, we obtained accuracy in between 0.70 and 0.82. For integer vectors the results were even more promising, as the accuracy score dipped to around 0.50 and 0.60.

From our experiments, we found that HMM, GAN, and WGAN generative models were not able to properly learn the true data distribution, especially with the Word2Vec and Bigram features. WGAN-GP, instead, clearly performed better. Hence, we concluded that using WGAN-GP algorithm is the best approach to successfully generate fake malware opcode sequences such that they appear closer to the real data distribution. This serves as a *proof of concept* that GAN algorithms, in particular WGAN-GP, can be successfully applied to generate malware opcode sequences, and not only in generating image data. The generative models can be used to boost malware datasets for some families that have very few data samples. Moreover, we confirmed that testing the quality of fake samples with Word2Vec and Bigram features is a better option as compared to integer vectors.

6.1 Future Work

There are a lot of different directions that this paper can be expanded in. For example, the dataset can be enlarged and the experiments can consider a larger number of malware families. Furthermore, instead of training individual GAN models for each family, a multi-class generative model can be considered. Another possible application is to use trained generative models to boost or augment the datasets for families that have a limited number of data samples. Other GAN variants could also be considered and compared, such as EBGAN and LSGAN. Finally, experiments with LSTM-GAN can be conducted since stateful networks can provide interesting results.

References

1. Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M., Giacinto, G.: Novel feature extraction, selection and fusion for effective malware family classification. In: Proceedings of the sixth ACM conference on data and application security and privacy. pp. 183–194 (2016)
2. Annachhatre, C., Austin, T., Stamp, M.: Hidden markov models for malware classification. Journal of Computer Virology and Hacking Techniques **11** (05 2014). https://doi.org/10.1007/s11416-014-0215-x
3. Arjovsky, M., Bottou, L.: Towards principled methods for training generative adversarial networks (2017)
4. Arjovsky, M., Chintala, S., Bottou, L.: Wasserstein gan (2017)
5. Biau, G., Scornet, E.: A random forest guided tour. TEST: An Official Journal of the Spanish Society of Statistics and Operations Research pp. 197–227 (2016). https://doi.org/10.1007/s11749-016-0481-7, https://doi.org/10.1007/s11749-016-0481-7
6. Burks, R., Islam, K.A., Lu, Y., Li, J.: Data augmentation with generative models for improved malware detection: A comparative study*. In: 2019 IEEE 10th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON). pp. 0660–0665 (2019). https://doi.org/10.1109/UEMCON47517.2019.8993085
7. Cortes, C., Vapnik, V.: Support-vector networks. Mach Learn 20 pp. 273–297 (1995). https://doi.org/10.1007/BF00994018, https://doi.org/10.1007/BF00994018
8. Gibert, D., Mateu, C., Planes, J.: The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. Journal of Network and Computer Applications **153**, 102526 (2020). https://doi.org/https://doi.org/10.1016/j.jnca.2019.102526, https://www.sciencedirect.com/science/article/pii/S1084804519303868
9. Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial networks (2014)
10. Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., Courville, A.: Improved training of wasserstein gans (2017)
11. Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., Hochreiter, S.: Gans trained by a two time-scale update rule converge to a local nash equilibrium. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. p. 6629–6640. NIPS'17, Curran Associates Inc. (2017)
12. Hu, W., Tan, Y.: Generating adversarial malware examples for black-box attacks based on gan (2017)
13. Huang, L., Joseph, A.D., Nelson, B., Rubinstein, B.I., Tygar, J.D.: Adversarial machine learning. In: Proceedings of the 4th ACM workshop on Security and artificial intelligence. pp. 43–58 (2011)
14. Hui, J.: Gan - what is generative adversarial networks gan? (Dec 2019), https://jonathan-hui.medium.com/gan-whats-generative-adversarial-networks-and-its-application-f39ed278ef09
15. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift (2015)
16. Jabbar, A., Li, X., Omar, B.: A survey on generative adversarial networks: Variants, applications, and training. ArXiv **abs/2006.05132** (2020)

17. Jain, M.: Image-based malware classification with convolutional neural networks and extreme learning machines. https://scholarworks.sjsu.edu/etd_projects/900/ (Dec 2019)
18. Krogh, A.: An introduction to hidden markov models for biological sequences. In: Salzberg, S., Searls, D., Kasif, S. (eds.) Computational methods in molecular biology. pp. 45–63. Elsevier, United Kingdom (1998)
19. Lu, Y., Li, J.: Generative adversarial network for improving deep learning based malware classification. In: 2019 Winter Simulation Conference (WSC). pp. 584–593 (2019). https://doi.org/10.1109/WSC40007.2019.9004932
20. M R, P., Jayagopal, P.: Generative adversarial networks: a survey on applications and challenges. International Journal of Multimedia Information Retrieval **10** (03 2021). https://doi.org/10.1007/s13735-020-00196-w
21. Mannor, S., Peleg, D., Rubinstein, R.: The cross entropy method for classification. In: Proceedings of the 22nd International Conference on Machine Learning. p. 561–568. ICML '05, Association for Computing Machinery (2005). https://doi.org/10.1145/1102351.1102422
22. Nappa, A., Rafique, M.Z., Caballero, J.: The malicia dataset: identification and analysis of drive-by download operations. International Journal of Information Security **14**(1), 15–33 (2015)
23. O'Kane, P., Sezer, S., McLaughlin, K.: Obfuscation: The hidden malware. IEEE Security Privacy **9**(5), 41–47 (2011). https://doi.org/10.1109/MSP.2011.98
24. Rabiner, L.: A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of the IEEE **77**(2), 257–286 (1989). https://doi.org/10.1109/5.18626
25. Roberts, J.M.: VirusShare.com - Because Sharing is Caring. http://www.virusshare.com (2011)
26. Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., Chen, X.: Improved techniques for training gans (2016)
27. Santos, I., Penya, Y.K., Devesa, J., Bringas, P.G.: N-grams-based file signatures for malware detection. ICEIS (2) **9**, 317–320 (2009)
28. Sawla, S.: Introduction to Naiïve Bayes for classification. https://medium.com/@srishtisawla/introduction-to-naive-bayes-for-classification-baefefb43a2d (2018)
29. Scikit-learn: K Neighbors Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html (Accessed on: (09/05/2021))
30. SonicWall: Sonicwall 2020 Cyber Threat Report. https://www.sonicwall.com/news/2020-sonicwall-cyber-threat-report (2020)
31. Stamp, M.: A revealing introduction to hidden markov models. Science pp. 1–20 (01 2004)
32. Stamp, M.: Introduction to Machine Learning with Applications in Information Security. Chapman &amp; Hall/CRC, 1st edn. (2017)
33. Sun, Z., Rao, Z., Chen, J., Xu, R., He, D., Yang, H., Liu, J.: An opcode sequences analysis method for unknown malware detection. p. 15–19. ICGDA 2019, Association for Computing Machinery (2019)
34. Trehan, H.: Fake malware opcodes generation using HMM and different GAN algorithms. Master's thesis, San Jose State University (2021), https://scholarworks.sjsu.edu/etd$_projects$/1001/
35. Yajamanam, S., Selvin, V.R.S., Di Troia, F., Stamp, M.: Deep learning versus gist descriptors for image-based malware classification. In: Icissp. pp. 553–561 (2018)