San Jose State University
SJSU ScholarWorks

Master's Projects Master's Theses and Graduate Research

Spring 5-25-2021

Fake malware opcodes generation using HMM and different GAN algorithms

Harshit Trehan

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Information Security Commons

Fake malware opcodes generation using HMM and different GAN algorithms

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Harshit Trehan

May 2021

© 2021

Harshit Trehan

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Fake malware opcodes generation using HMM and different GAN algorithms

by

Harshit Trehan

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2021

Dr. Fabio Di Troia	Department of Computer Science
Dr. Katerina Potika	Department of Computer Science
Dr. Nada Attar	Department of Computer Science

ABSTRACT

Fake malware opcodes generation using HMM and different GAN algorithms

by Harshit Trehan

Malware, or malicious software, is a program that is intended to harm systems. In the past decade, the number of malware attacks have grown and, more importantly, evolved. Many researchers have successfully integrated cutting edge Machine Learning techniques to combat this ever present and growing threat to cyber and information security. One big challenge faced by many researchers is the lack of enough data to train machine learning models and specifically deep neural networks properly. Generative modelling has proven to be very efficient at generating synthesized data that can match the actual data distribution.

In this project, we aim to generate malware samples as opcode sequences and attempt to differentiate between the fake and real samples. We use different Generative Adversarial Networks (GAN) algorithms and Hidden Markov Models (HMM) to generate fake samples.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Fabio Di Troia who has helped me immensely throughout my research project and guided me through the challenges of this research. He has patiently answered all my questions and encouraged me to keep trying hard and face the tough challenges.

I would also like to thank my committee members, Dr. Katerina Potika and Dr. Nada Attar who have given me their valuable time and feedback.

Finally, I would like to thank all my professors and family members who have believed in me and provided me the support I needed every step of the way. I am eternally indebted to them for always expecting the best from me and pushing me to give my best.

TABLE OF CONTENTS

CHAPTER

1 Introduction 1

2 Background 4

2.1 Background and Related Work 4

2.2 Hidden Markov Models 5

2.2.1 HMM Introduction and Working 6

2.3 Generative Adversarial Networks 8

2.3.1 GAN Working and Architecture 8

2.3.2 GAN Training 10

2.3.3 GAN Limitations 10

2.4 Wasserstein GAN 11

2.4.1 WGAN working 11

2.4.2 WGAN Training 13

2.4.3 WGAN limitations 14

2.5 WGAN with Gradient Penalty 15

2.6 k-Nearest Neighbor 17

2.7 Support Vector Machines 17

2.8 Naïve Bayes Classifier 19

2.9 Random Forest 19

3 Methodology 21

3.1 Fake Malware using HMM 21

3.2 Fake Malware using GAN 22

3.3 Feature Extraction 22

3.4 Evaluation 23

3.4.1 HMM Evaluation 23

3.4.2 GAN Evaluation 24

3.4.3 Accuracy, Precision and Recall 25

4 Implementation 26

4.1 Dataset 26

4.2 HMM Implementation 27

4.3 GAN Implementation 31

4.3.1 GAN Stabilizing Techniques 33

4.4 WGAN Implementation 34

4.4.1 Wasserstein Distance 37

4.5 WGAN with Gradient Penalty Implementation 37

5 Results and Discussion 41

5.1 HMM Results 41

5.1.1 Optimum M and HMM Training Results 41

5.1.2 HMM Classification Results 43

5.2 GAN Results 46

5.2.1 Best GAN Generative Model 46

5.2.2 GAN Classification Results 47

5.3 WGAN Results 49

5.3.1 Convergence and Loss Values 49

5.3.2 WGAN Classification Results 50
5.4 Wasserstein GAN with Gradient Penalty 53
5.4.1 Convergence and Loss Curves 54
5.4.2 WGAN-GP Classification Results 55
6 Conclusion and Future Work 59
6.1 Future Work 61
LIST OF REFERENCES 62
APPENDIX
A ROC Curves and bar plots for optimum M. 66
B Loss curves for different GAN architectures. 74
B.1 GAN Loss Curves 74
B.2 WGAN Loss Curves 76
B.3 WGAN with Gradient Penalty Loss Curves 77
C Code snippets 79
C.1 Wasserstein Loss 79
C.2 Gradient Penalty 79

LIST OF TABLES

1 HMM notation. 6

2 GAN notation. 9

3 Dataset summary 27

4 Unique Opcodes in observation sequences for each family 28

5 Best value of M for each family 41

6 Best 2 models for fake samples from each family. 43

7 SVM-HMM scores for each feature. 44

8 Naïve Bayes-HMM scores for each feature. 44

9 Random Forest-HMM scores for each feature. 45

10 kNN-HMM scores for each feature. 45

11 Best GAN generative model for each family 46

12 SVM-GAN scores for each feature. 47

13 Naïve Bayes-GAN scores for each feature. 48

14 Random Forest-GAN scores for each feature. 48

15 kNN-GAN scores for each feature. 49

16 Best WGAN generative model for each family. 51

17 SVM-WGAN scores for each feature. 52

18 Naïve Bayes-WGAN scores for each feature. 52

19 Random Forest-WGAN scores for each feature. 53

20 kNN-WGAN scores for each feature. 53

21 Best WGAN-GP generative model for each family. 56

22 SVM-WGAN with Gradient Penalty scores for each feature. . . . 56

23 Naïve Bayes-WGAN with Gradient Penalty scores for each feature. 56

24 Random Forest-WGAN with Gradient Penalty scores for each feature. 57

25 kNN-WGAN with Gradient Penalty scores for each feature. . . . 57

LIST OF FIGURES

1 GAN training pipeline [25]. 10

2 kNN classification example. [31] 18

3 Possible hyperplanes [33]. 18

4 Optimal hyperplane [33]. 18

5 Example usage of objdump. 26

6 Top 20 opcodes from the observation sequences and the number of families they are present in. 29

7 Top 40 opcodes from the observation sequences and the number of families they are present in. 30

8 GAN Discriminator Architecture. 32

9 GAN Generator Architecture. 32

10 WGAN Critic and Generator Architecture. 36

11 WGAN-GP Critic Architecture. 38

12 WGAN-GP Generator Architecture. 40

13 WinWebSec 22×22 models. 42

14 Zbot 20×20 models. 42

15 Renos 22×22 models. 42

16 OnLineGames 22×22 models. 42

17 VBInject 25×25 models. 42

18 WinWebSec WGAN critic loss. 50

19 WinWebSec WGAN generator loss. 51

20 WinWebSec WGAN-GP critic loss. 54

21 WinWebSec WGAN-GP generator loss. 55
A.22 AUC scores for different M values for WinWebSec. 66
A.23 AUC scores for different M values for Zbot. 66
A.24 AUC scores for different M values for Renos. 67
A.25 AUC scores for different M values for OnLineGames. 67
A.26 AUC scores for different M values for VBInject. 68
A.27 ROC curves for different M values for WinWebSec. 69
A.28 ROC curves for different M values for Zbot. 70
A.29 ROC curves for different M values for Renos. 71
A.30 ROC curves for different M values for OnLineGames. 72
A.31 ROC curves for different M values for VBInject. 73
B.32 WWS GAN discriminator loss. 74
B.33 WWS GAN generator loss. 74
B.34 Zbot GAN discriminator loss. 74
B.35 Zbot GAN generator loss. 74
B.36 Renos GAN discriminator loss. 75
B.37 Renos GAN generator loss. 75
B.38 OLG GAN discriminator loss. 75
B.39 OLG GAN generator loss. 75
B.40 VBInject GAN discriminator loss. 75
B.41 VBInject GAN generator loss. 75
B.42 Zbot WGAN critic loss. 76
B.43 Zbot WGAN generator loss. 76

B.44 Renos WGAN critic loss. 76
B.45 Renos WGAN generator loss. 76
B.46 OLG WGAN critic loss. 76
B.47 OLG WGAN generator loss. 76
B.48 VBInject WGAN critic loss. 77
B.49 VBInject WGAN generator loss. 77
B.50 Zbot WGAN-GP critic loss. 77
B.51 Zbot WGAN-GP generator loss. 77
B.52 Renos WGAN-GP critic loss. 77
B.53 Renos WGAN-GP generator loss. 77
B.54 OLG WGAN-GP critic loss. 78
B.55 OLG WGAN-GP generator loss. 78
B.56 VBInject WGAN-GP critic loss. 78
B.57 VBInject WGAN-GP generator loss. 78

CHAPTER 1

Introduction

Malicious software, or Malware in short, is a software that is specifically designed to harm computer systems by affecting devices, tampering with/stealing data and even harming people. 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.

Due to the exponential increase in the number of technological devices such as smart phones, laptops, tablets and many other devices encapsulated by "Internet of Things (IoT)", the number of malware attacks has grown rapidly. According to data collected by SonicWall, there were a total of 9.9 billion [1] malware attacks worldwide in 2019 alone.

Traditional malware classification and detection techniques can be broadly divided into 3 categories: behavior/anomaly-based detection, signature-based detection and heuristics-based detection [2].

In anomaly-based techniques, a detector requires prior knowledge of what "normal" behavior should be and uses that knowledge to decide whether the program being inspected is harmful or not. This involves 2 phases: training phase and monitoring phase. In the training/learning phase, the detector learns the system's healthy/normal behavior. In the monitoring phase, the detector flags programs that alter the behavior of the system. One big drawback of these techniques is the alarmingly high false positive rate [3].

In signature-based techniques, the detector is required to know the signature of a malicious program and can decide whether the current program is malicious or not [2] based on its signature. The signature is extracted from a set of contiguous bytes in the program. On identifying the signature as malware, the signature is stored in a

database. Although most commercial anti-viruses still use signature-based malware detection techniques, they have a major drawback that they can only identify known malware. This means that "zero-day" malware, which haven't been identified yet are very hard to catch.

In heuristics-based techniques, the suspected malicious file is either executed in a virtual environment and the set of instructions is observed (dynamic analysis) or decompiled before executing it and the code is examined (static analysis) [4]. Heuristics based analysis can help discover some zero-day attacks but their drawback is that they are unable to detect malware that use different or newer techniques to harm systems.

To combat these disadvantages, machine learning techniques have been researched and developed. During dynamic and static analysis of malicious files, features such as opcode sequences, API calls, bytes vectors and many other [5, 6, 7] are extracted. Machine Learning models are trained on these features and tested against potential malware files.

Although machine learning techniques have shown promising results, there are still some challenges to this technique. In recent years, malware authors have evolved even more and developed methods that make malware detection even harder even for machine learning based techniques. Methods such as malware obfuscation [8] in which the code of a malicious file is convoluted with dead code and some random code is inserted from benign files makes it hard to detect a particular file as malicious. Some other challenges to machine learning based techniques include the availability, or lack there of, of large public datasets for research purposes [9] and new techniques employed by malware authors such as Adversarial machine learning [10] to throw off machine learning models.

In this project, we use mnemonic opcodes extracted from malware executables

belonging to 5 different malware families. We focus on generating realistic fake malware samples by solving Problem 2 of Hidden Markov Models (HMM) (see Chapter 2) and thereby estimating the hidden states to get the opcodes sequence. In addition to using HMM to generate opcode sequence, we use Generative Adversarial Networks(GAN) [11] to generate fake opcodes sequence.

We use multiple machine learning classification techniques: Support Vector Machines, k-Nearest Neighbor, Random Forest and Naïve Bayes Classifier to differentiate between fake and real samples and compare the 2 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.

In Chapter 2, we go over some previous and related work. We also give a brief summary of the techniques and concepts we are using for this project. In Chapter 3 we explain our workflow and give a description of our malware generation pipeline. In Chapter 4 we go over the actual implementation and our experimental setup. In Chapter 5 we provide the results of our experiments and finally in Chapter 6 we conclude the project and discuss the future directions for our project.

CHAPTER 2

Background

In this chapter, we discuss the background of malware classification and use of generative modelling for the same. We lay out some of the drawbacks of malware as images classification and highlight the gap in the literature with respect to generated/synthetic malware opcode samples.
We also give a brief introduction to Hidden Markov Models, Generative Adversarial Networks and the Machine Learning techniques used to evaluate our results.

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. For example, in [12] S. Yajamanam et al. extract features from malware images known as gist-descriptors and classified malware samples using these features with k-Nearest Neighbors algorithm. Then they compare the performance of gist-descriptors versus deep neural networks. In both cases they received excellent results, with over 90% accuracy. In [13], the authors classified malware images using a Convolutional Neural Network (CNN) and achieved excellent classification results with 100% accuracy for 14 out 25 malware families that they considered.

The examples above show clear advantages of using deep learning techniques with image-based data. In terms of generative modelling, many researchers have used malware images to generate malware samples as that gives the advantage of boosting the dataset and even performing data augmentation to real samples so that they conform to some obfuscated malware samples. In [14], the authors used Variational Auto Encoder (VAE) and GANs to boost malware dataset and saw a 2% increase in

accuracy in case of VAE and 6% increase in case of GAN. They also used malware as images. A very similar research is [15] where 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. But the drawback of this technique is that converting malware files to images is computationally expensive. Moreover, training deep convolutional networks is also computationally expensive and it takes a long time to train and test the models. Using GANs with images has similar overheads.

In [16], Weiwei Hu and Ying Tan propose a GAN based model that is able to bypass black-box malware detection systems which almost 0 detection rate. They used API features extracted from the malware samples as they are executed in a virtual environment such as a sandbox. This model is called "MalGAN" and certainly performed very well. But executing malware in sandbox environment to get the API features is again an overhead.

There is a gap in the literature when it comes generating malware samples using non-image features or representations of malware. So we explore this gap by utilizing mnemonic opcodes extracted from malware files and generating mnemonic opcode samples using HMM and 3 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 timeseries or sequential data. They have been successfully used in speech analysis and recognition [17], malware classification [18] and genes sequence analysis [19].

Table 1: HMM notation.

Symbol	Description
M	Number of distinct observation symbols
N	Number of hidden states
T	Length of the observation sequence
$\mathcal{O}$	Observation sequence
Q	Set of distinct states
V	Set of all possible observations
A	State transition probability matrix
B	State-Observation probability matrix
π	Initial state probability matrix

2.2.1 HMM Introduction and Working

A Markov model is defined as a statistical model which has states and the transition probabilities from one state to another is known [20]. In a Markov Model, the states are known to an observer. On the other hand, in an HMM the underlying states are not known to the observer. The state transition probability between states is known and the probability distribution of observing a set of observation symbols for each state is know [20].

Some notations required to understand HMM are given in Table 1. From Table 1 we see that V is the set of possible observations and M is the total number of unique observation symbols. In other words, $|V| = M$ and $V = \{0, 1, ..., M-1\}$. Additionally, observation sequence, $\mathcal{O}$, of length T is comprised of V, i.e., $\mathcal{O}_i \in V$, for $i = 0, 1, .., T-1$.

An HMM model is denoted by λ. It is characterized by the matrices A, B and π. The matrix A denotes the state transition probability between 2 states. It's dimensions are $N \times N$.

$$a_{ij} = P(state\ q_j\ at\ time\ t+1 \mid state\ q_i\ at\ time\ t)$$

where, $A = \{a_{ij}\}$. The B matrix, $B = \{b_j(k)\}$ is of dimension $N \times M$ and denotes the probability of observing a symbol $\mathcal{O}_k$ at time t when in state q_i:

$$b_j(k) = P(observation\ k\ at\ time\ t \mid state\ q_j\ at\ time\ t)$$

We can use HMM to solve 3 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 [20].

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 [20].

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 we are training the model to match the observation sequence [20].

The solution to these problems is implemented via the Baum-Welch algorithm [21]. It uses a forward and backward algorithm to find the unknown parameters of the Hidden Markov Model, or in other words, to train the model (Problem 3). Additionally, the forward algorithm is used to solve Problem 1 and the backward algorithm is used to solve Problem 2. The γ matrix from the Baum-Welch algorithm contains a probability value for each state at time t, where $t \in \{0, 1, .., T-1\}$. So its dimensions are $T \times N$.

We find the most likely hidden state sequence by finding the states Q_i at t with the highest probability. So this gives us most likely state sequence of length T.

In our case, we need to solve all 3 of the problems. More details on why we need to solve all 3 problems are given in Chapter 4, Section 4.2.

HMM is a hill climb algorithm, that means that it finds the local maxima on the parameter space A, B and π. To ensure a global maximum, we need to train the model multiple times and each time with a random initialization of the parameters A, B and π. This is the concept of "random restarts" in HMM [20].

2.3 Generative Adversarial Networks

Generative Adversarial Network(GAN) is one the most exciting topic in the field of generative modelling and Machine Learning right now. GAN was introduced by Ian J. Goodfellow et al. [11] in 2014. The GAN model consists of 2 neural networks: the discriminator and the generator which participate in a zero-sum game to achieve Nash equilibrium. The objective of the 2 networks is different from each other but the overall objective of the network is to generate data samples that conform to a probability distribution p_g which is similar to the true data's probability distribution p_{true}. The generator tries to fool the discriminator by getting it to classify the generated samples as real and the discriminator tries to identify the samples as fake or real.

2.3.1 GAN Working and Architecture

Table 2 shows the notation used in defining the GAN Architecture. The discriminator, D, is trained such that on seeing a true sample, $x \sim p_t$, it classifies it accurately as a real sample by maximizing the log likelihood of $D(x)$. Conversely, D is trained such that on seeing a fake sample $z \sim p_g$, it classifies it as fake giving a probability as close to 0 as possible. This is achieved by maximizing the log likelihood of $1 - D(G(z))$:

$$\max_D \mathbb{E}_{x \sim p_t}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log 1 - D(G(z))] \tag{1}$$

Table 2: GAN notation.

Symbol	Description
G	Generator
D	Discriminator
z	Random noise belonging to probability distribution p_z
p_t	Probability distribution of true data samples
p_g	Probability distribution of G's output $G(z)$
$D(.)$	Output of D for any input .
$G(.)$	Output of G for any input .
$V(D,G)$	Cost function for the GAN

On the other hand, the generator, G, is trained such that D, on seeing a fake sample, $G(z)$, classifies it as real. This is achieved by minimizing the log likelihood of $1-D(G(z))$:

$$\min_G \mathbb{E}_{z\sim p_z}[\log 1 - D(G(z))] \tag{2}$$

Since the Generator's objective is independent of the Discriminator's predictions on true data sampled from p_t, we can combine (1) and (2). We get a combined cost function $V(D,G)$:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_t}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log 1 - D(G(z))] \tag{3}$$

This is the minimax game that the Generator and Discriminator play. The authors of the original paper showed that once the Generator is trained to optimum, p_g gets very close to p_t and consequently, D becomes very close to 1/2. Equation (3) is essentially finding the divergence between the probability distribution p_g and p_t minimizing the Jensen-Shannon(JS) divergence between these 2 probability distributions as shown in Theorem 1 in [11] and [22].

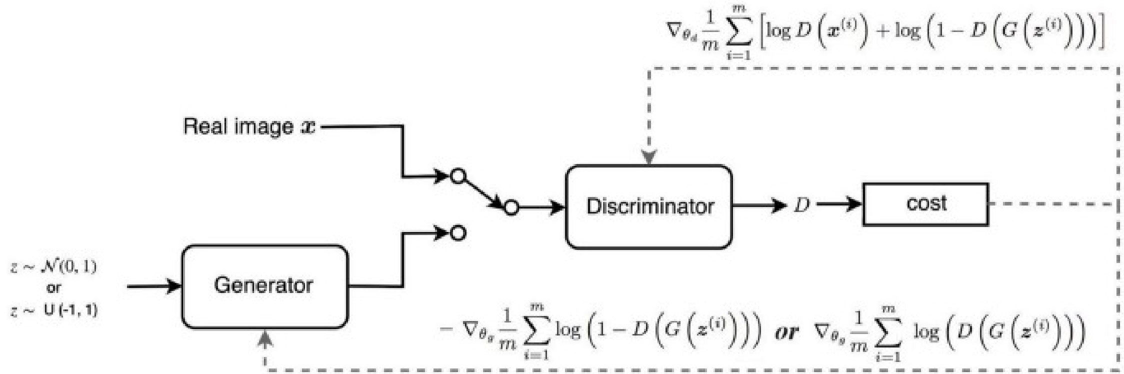

Figure 1: GAN training pipeline [25].

2.3.2 GAN Training

When actually training the model, the loss function used is Binary Crossentropy [23] which calculates the difference in the probability distribution of true samples, labelled 1, and false samples labelled as 0. The weights of both models are updated independently of each other using 2 loss functions on the models parameterized by their weights: $G(z : \theta_g)$ and $D(x : \theta_d)$. The formal algorithm for GAN training is given in Algorithm 1. Figure 1 shows the training steps graphically.

The authors of the paper mentioned that we can use any optimizer to update the weights of both the models. They suggested that momentum based optimizers, like Adam [24], with low learning rate $(1e - 3 \sim 1e - 5)$ work well as they allow for faster convergence of both the networks.

2.3.3 GAN Limitations

Although GANs excel in learning complex data distributions, there exist major challenges in training of GANs such as mode collapse, vanishing gradient, internal covariate shift, failure mode etc. To overcome these challenges, several novel variants and architectures of GANs have been researched and implemented. [26] and [27] provide a comprehensive analysis of the challenges in GAN training and pros and cons of various GAN architectures.

Algorithm 1 GAN Training [11].

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

2.4 Wasserstein GAN

Wasserstein GAN (WGAN) [28] was first proposed in 2017 by M. Arjovsky et al. as an improvement over the vanilla GAN. They first published a paper [22] highlighting the important theoretical implications of GAN training as proposed by Ian J. Goodfellow et al. [11] and laid out the mathematical reasoning and proofs for some of the issues surrounding GAN training, as explained in Section 2.3.3.

2.4.1 WGAN working

The main idea of the WGAN is that instead of optimizing the JS Divergence between 2 probability distributions p_θ and p_t, use of a different distance metric as the loss function is proposed which is called the Wasserstein distance or Earth-Mover distance is proposed (in this section we represent the generator's probability distribution with p_θ instead of p_g to conform to the notations used by the authors). The Wasserstein distance is a measure of distance between 2 probability distributions. It 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.

In [28], the authors explain why using JS Divergence, or some other distance metrics, such as KL Divergence is not conducive to optimally training the discriminator, In fact, they give an example, Example 1 in [28] which shows that in come cases the value of JS Divergence goes to 0 and hence, when the discriminator is trained to optimality, i.e., when it can perfectly classify fake vs real data samples, then the value of loss function in Equation (3) provides no meaningful feedback to the generator. No learning takes place from there on as the model saturates.

Equation (4) represents the formula for Wasserstein distance between 2 probability distributions in a continuous probability domain.

$$W(p_t, p_\theta) = \inf_{\gamma \in \prod(p_t, p_\theta)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|] \tag{4}$$

In Equation (4), $\prod(p_t, p_\theta)$ represents the set of all possible join probability distributions between p_t and p_θ and $\gamma \in \prod(p_t, p_\theta)$ indicates one out of many "dirt" transport plans. Finally, $\gamma(x, y)$ indicates the amount of "mass" to be moved from x to y in order to transform p_t to p_θ. The EM distance is the *infimum* or the smallest value of "cost" out of all the plans.

Exhausting all join distributions between p_t and p_θ is intractable. So, the authors applied a smart transformation to the formula in (4) using the Kantorovich-Rubinstein duality to get:

$$W(p_t, p_\theta) = \frac{1}{K} \sup_{\|f\|_L \leq K} \mathbb{E}_{x\sim p_t}[f(x)] - \mathbb{E}_{x\sim p_\theta}[f(x)] \tag{5}$$

where the *infimum* is replaced with the *supremum*, which means the maximum value and p_θ is the probability distribution for generator's outputs. The condition

$\|f\|_L \leq K$ implies that the function f is *K-Lipschitz continuous.* A function $f : \mathbb{R} \rightarrow \mathbb{R}$ is K-Lipschitz continuous if there exists a real number $K \geq 0$, s.t., $\forall(x_1, x_2) \in \mathbb{R}$:

$$| f(x_1) - f(x_2) | \leq K | x_1 - x_2 |$$

where K is known as the Lipschitz constant for function f. Now, suppose there is a family of functions $\{f_w\}_{w \in \mathcal{W}}$ where they are all K-Lipschitz continuous and w is the set of optimal weights and $\mathcal{W}$ is the set of possible weights, then the problem to solve i.e. the loss function, becomes:

$$L(p_t, p_\theta) = W(p_t, p_\theta) = \max_{w \in \mathcal{W}} \mathbb{E}_{x \sim p_t}[f_w(x)] - \mathbb{E}_{z \sim p_z}[f_w(G_\theta(z))] \quad (6)$$

where p_z is the noise distribution from which z is sampled, G_θ represents the generator parameterized by the weights θ and the "discriminator" is parameterized by the optimal set of weights $w \in \mathcal{W}$.

The "discrimiantor" is no longer predicting whether a sample is fake or real, it is simply trying to learn a K-Lipschitz continuous function that will help in calculating the Wasserstein distance between p_t and p_θ. Due to this, the authors of the paper call this model the "critic" instead of discriminator.

Since we already know that neural networks are designed to learn functions, we can train the neural network to learn the K-Lipschitz continuous function $\{f_w\}_{w \in \mathcal{W}}$ and train the critic to optimal. This will allow for an estimation of the Wasserstein distance up to a multiplicative constant (K). In the next section, we explain how the training of a WGAN works.

2.4.2 WGAN Training

We need to first understand how the generator, g_θ, updates its weights. We can get the gradient for θ using Equation (6) as:

$$\nabla_\theta W(p_t, p_\theta) = \nabla_\theta(\mathbb{E}_{x \in p_t}[f_w(x)] - \mathbb{E}_{z \in p_z}[f_w(G_\theta(z))])$$

which gives

$$\nabla_\theta W(p_t, p_\theta) = -\mathbb{E}_{z \in p_z}[\nabla_\theta f_w(G_\theta(z))]) \tag{7}$$

To train a WGAN we need to find the optimal f_w. To do this, for a fixed value of g_θ, we try to train the critic to optimality and obtain the optimal f_w for the Wasserstein distance. Once we have an optimal f_w, we update the generator's weights, θ, through back propagation as given in Equation (7). This process involves training the critic more times than the generator since our goal is to train the critic to optimality (formally, we have n_{critic} iterations per generator iteration and the authors recommend a value between 5 and 10). This way the gradient information that will be back propagated through the network will be very efficient and the generator can update it's weights effectively. The authors state the use of RMSProp optimizer instead of Adam [24] citing higher stability and better performance.

However, there is 1 constraint in the training algorithm which is to ensure that f_w is K-Lipschitz continuous throughout the training. To achieve this, the authors use a very simple but effective trick: The value of the weights, w, is clamped within a compact space $\mathcal{W}$ such as $[-0.01, 0.01]$. This way, f_w receives an upper and lower bound and is Lipschitz continuous in this space. The formal algorithm for WGAN training is given in Algorithm 2

2.4.3 WGAN limitations

The main drawback of the WGAN algorithm is the way K-Lipschitz continuity is enforced. 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. To quote the authors of the paper themselves: "Weight clipping is a clearly terrible way to enforce a Lipschitz constraint. If the clipping parameter is large, then it can take a long time for any weights to reach their limit, thereby making it harder to train the critic till optimality.

Algorithm 2 WGAN Training [28]. Default parameter values are: $\alpha = 0.00005$, $n_{critic} = 5$ and $c = 0.01$

Require:

α: 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_θ: generator parameterized by weights θ.

while θ 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

If the clipping is small, this can easily lead to vanishing gradients when the number of layers is big, or batch normalization is not used (such as in RNNs).''

2.5 WGAN with Gradient Penalty

Wasserstein GAN with Gradient Penalty (WGAN-GP) [29] was first introduced in 2017 by Ishaan Gulrajani et al. The main objective of this architecture is to overcome the drawback of WGAN which is the way Lipschitz continuity is enforced. As mentioned in Section 2.4.1, the value of the weights w is clamped in a compact space $\mathcal{W}$ so that f_w is bound and K-Lipschitz continuous in this range. This is a clear limitation in WGAN (see Section 2.4.3).

To overcome this, the authors in [29] propose an improved WGAN training method. They present Corollary 1 in [29] which claims that the optimal critic in WGAN has gradient norm equal to 1 and it is 1-Lipschitz continuous. Using this fact, a ``penalty'' is imposed on the critic if it's gradient's norm deviates from 1. The

objective function of WGAN-GP now changes to:

$$L(p_t, p_\theta) = \mathbb{E}_{x\in p_t}[f_w(x)] - \mathbb{E}_{z\in p_z}[f_w(G_\theta(z))] + \lambda\mathbb{E}_{\hat{x}\in p_{\hat{x}}}[(\|\nabla_{\hat{x}} f_w(\hat{x})\|_2 - 1)^2] \quad (8)$$

The last term on the right side is the gradient penalty. λ is the penalty coefficient and is set to 10 [29].

In Equation 8, the distribution $p_{\hat{x}}$ is defined implicitly as sampling uniformly from straight lines formed by pair of points belonging to p_t and p_θ, where p_θ is the generator's distribution as defined by $G_\theta(z)$. When actually implementing the algorithm, $\hat{x}$ is estimated by taking a weighted average between pair of data samples (x, y) such that $x \in p_t$ and $y \in p_\theta$ or $G_\theta(z)$. Usually $\hat{x}$ is known as the interpolated sample. The authors also suggested the use of Adam optimizer instead of RMSProp as it performed better. The formal algorithm is given in Algorithm 3.

Algorithm 3 WGAN with Gradient Penalty Training [29]. 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, λ: penalty coefficient, n_{critic}: number of critic iterations per generator iteration, α, β_1, β_2: Adam parameters, C_w: critic parameterized by weights w, G_θ: generator parameterized by weights θ.
  while θ 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

Algorithm 3 is very similar to WGAN's algorithm (Algorithm 2) minus the weight clipping part and the addition of the gradient penalty.

2.6 *k*-Nearest Neighbor

k-Nearest Neighbor (kNN) is a supervised classification algorithm. The k in kNN stands for the number of neighbors to consider. To classify an input, say x, the algorithm considers "k" points from the given training data such that the k points are nearest to x. Then a class is assigned to x depending on the majority of classes in the k neighbors. The distance metric used to calculate the distance between points can be Euclidean, Manhattan, Minkowski etc [30]. The most widely used metric is Euclidean distance.

An advantage of kNN algorithm is that there is no training or model fitting to be done. The labelled training data is just used to calculate the distance between input points and assign classes thereafter. The disadvantage is that although there is no training, testing data can take a long time especially with large datasets of high dimensions.

An example of kNN algorithm is given in Figure 2. There are 2 classes of data, class A (yellow) and class B (pink). When a new input, red, is to be classified and the value of k=3, then 3 nearest neighbors to red are calculated. We can see that 2 of the 3 belong to class B and 1 belongs to class A. So the red input will be assigned class B. On the other hand, when k=6, 4 of the nearest neighbors belong to class A and only 2 belong to B. In that case, the red input will be assigned to class A.

2.7 Support Vector Machines

Support Vector Machine (SVM) [32] is a supervised machine learning algorithm that tries to generate a decision boundary between 2 classes of data, known as the hyperplane. It is a binary classifier that can be extended for multi-class classification

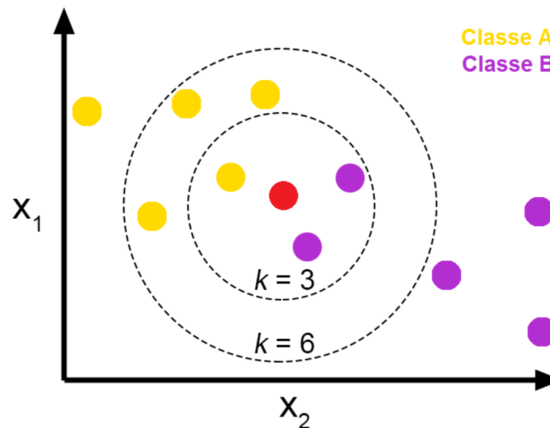

Figure 2: kNN classification example. [31]

by utilizing techniques such as One-vs-Rest. The main idea behind SVM is to find a hyperplane such that the distance between the hyperplane and the closest point from the 2 classes is maximized. This distance is called the margin. The hyperplane with the maximum margin is the optimal solution. Figure 3 shows the possible separation hyper-planes between the data points belonging to 2 different classes and Figure 4 shows the optimal hyperplane such that the margin is maximum. Sometimes when a

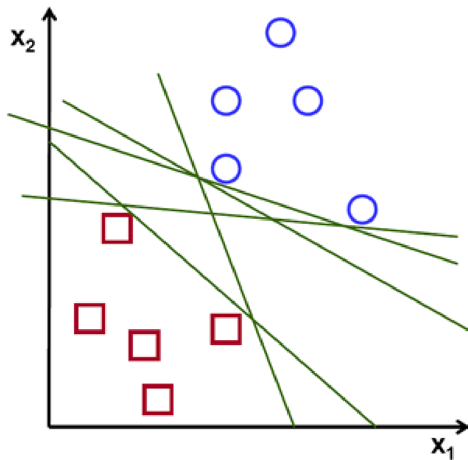

Figure 3: Possible hyperplanes [33].

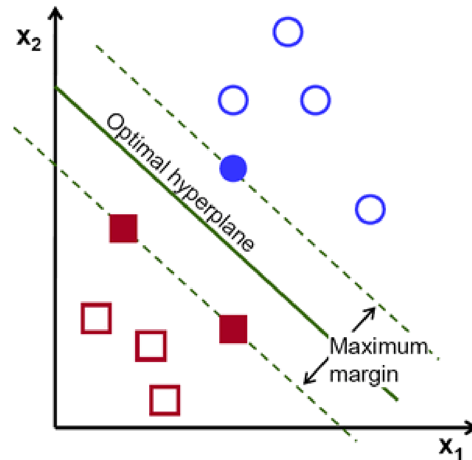

Figure 4: Optimal hyperplane [33].

clean hyperplane cannot be found, the data is projected into a higher dimension so that a clean separation boundary is possible, but this increases the complexity and

may make it intractable. In 1992, a new technique known as the kernel trick [34] was invented which allows SVM to work efficiently for higher dimensional data.

2.8 Naïve Bayes Classifier

Naïve Bayes Classifier [35] is a supervised learning algorithm that uses Bayes theorem to compute the probability of an observation belonging to a particular class. The Bayes theorem or formula is:

$$P(B|A) = \frac{p(A|B)\ p(B)}{p(A)} \tag{9}$$

where, $P(B|A)$ is the posterior probability, meaning the probability of hypothesis B given data A, $p(A|B)$ is the probability of data A given hypothesis B is true, $p(B)$ is called the prior probability, meaning the probability of B, $p(A)$ is the probability of data A.

The algorithm is called "Naïve" because it makes an assumption that the input features are independent of each other. During training, probability of observing a feature is calculated for each class using Equation (9). For classification, the probability of an input belonging to every class is calculated and the class with the highest probability is assigned to the input vector.

2.9 Random Forest

Random Forest is a supervised machine learning algorithm. It is an ensemble technique which means that it utilizes existing classification techniques and combines them to produce a stronger classifier.

The underlying principle behind Random Forest is decision tree. Random Forest consists of multiple decision trees which are created when fitting the training data. During testing, each decision tree outputs a prediction and the majority of these outputs is selected as the final class output by the Random Forest classifier. Each individual decision tree is built or grown as follows:

1. Select a subset of training data.

2. Select a set of features randomly from the subset of the data and start splitting the tree node.

3. The node is split into 2 child nodes so as to maximize the difference between the features of 2 new groups and minimize the difference between features inside a group.

4. Nodes are split until perfect groups are obtained or until the specified maximum depth is reached.

A combination of multiple decision trees each trained as explained above with a random subset of training data makes up the Random Forest classifier.

CHAPTER 3

Methodology

In this chapter, 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 below:

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

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. Chapter 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 4 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 2 highest scoring models from Step 4 and use their γ matrix to find out the most likely state sequence of the HMM model.

7. The most likely state sequence represents the fake samples. Score and evaluate these fake samples as explained in Section 3.4.

3.2 Fake Malware using GAN

The methodology adopted for generating fake malware samples using GANs is explained below:

1. We use 3 different GAN architectures to generate fake samples: Vanilla GAN (or just GAN), Wasserstein GAN and Wasserstein GAN with Gradient Penalty.
2. Train GAN models for each family and save generator models at an interval of 200 epochs for GAN and 500 for WGAN and WGAN with gradient penalty.
3. Generate fake samples in batches of 32 using the saved generative models.
4. Evaluate them against real data samples by simply testing the integer vectors (see 3.3) representing real samples and fake samples.
5. Repeat Step 4 5 times and the average the results.
6. Select the best scoring model as the final generative model for each family, giving a total of 5 generator models per architecture.
7. 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.
8. Repeat Steps 2-6 for WGAN and WGAN-GP architectures.

This process is explained in more detail in Section 3.4.2.

3.3 Feature Extraction

In this section we explain our feature extraction process and the types of features used for evaluation. We extract 3 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 and normalizing 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 evaluate all the HMM models by creating the Receiver Operating Characteristic (ROC) curve for each model and calculating the AUC.

3.4.1 HMM Evaluation

HMM scores represent the log likelihood value of an observation sequence belonging to the trained model. An ROC curve is created to show the classification status of a model at different thresholds. A threshold value is defined for probabilistic models such that values above a threshold are considered belonging to class A and values below the threshold are considered belonging to class B (or class NOT A).

Fixing a threshold is not good practice since it doesn't always lead to accurate classification. Instead, different values of threshold are considered one by one and the True Positive Rate (TPR) and False Positive Rate (FPR) are calculated for each threshold value. An ROC curve plots the TPR vs. FPR.

$True\ Positive\ (TP)$ = classified as true and are actually true.

$True\ Negative\ (TN)$ = classified as false and are actually false.

$False\ Positive\ (FP)$ = classified as true but are actually false.

$False\ Negative\ (TN)$ = classified as false but are actually true.

TPR and FPR are defined as follows:

$$TPR = \frac{TP}{TP + FN}$$
$$FPR = \frac{FP}{FP + TN}$$

Area Under the Curve (AUC) is simply the area under the ROC curve. It provides an overall measure of the classification performance. AUC is in range $[0, 1]$ and a larger AUC value means better classification with 0 meaning all predictions were wrong and 1 meaning all predictions were right.

3.4.2 GAN Evaluation

The most common application for GANs is in the image domain. Most researches use benchmark datasets such as MNIST, CIFAR10, ImageNet etc to evaluate the performance of their GANs. A batch of generated images is saved every few hundred epochs, like 200, and they are visually inspected. There are 2 common metrics used to evaluate the quality of generated images: Inception Score [36] and Fréchet Inception Distance (FID) [37].

In our case, we are generating opcode sequences and which can't be inspected visually and Inception Score and FID scores are defined only for images. So to evaluate our GAN models, we saved the generative model at every 200 epochs for GAN and 500 epochs for WGAN and WGAN with gradient penalty. 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 (0, 200, 500 and so on), that gave the lowest classification results is chosen as the best generative model from that architecture. The best model is then used for evaluation as explained in the next section.

3.4.3 Accuracy, Precision and Recall

To score and evaluate the quality of the fake samples (HMM and GANs), we trained 4 machine learning models and calculated the Accuracy, Precision and Recall for each model with each feature.

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

Accuracy

Accuracy is a measure of correct predictions vs total predictions made. For binary classification, it can be represented using TP, TN, FP and FN:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision

Precision is a measure of the data samples classified correctly as positive vs all samples classified as positive. In other words, in measures how "precise" the model is.

$$Precision = \frac{TP}{TP + FP}$$

Recall

Recall is a measure of the data samples classified correctly as positive vs all the actually positive samples. In other words, it measures how many positive data samples the model was able to "recall". It is the same as TPR.

$$Recall = \frac{TP}{TP + FN}$$

CHAPTER 4

Implementation

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

4.1 Dataset

Out dataset consists of 5 malware families and a benign dataset. Each malware family has over 1000 samples and the benign dataset has over 700 samples containing mnemonic opcode sequences. We began with the Malicia dataset [38] which has over 50 malware families with already extracted opcodes and selected WinWebSec and Zbot since these 2 families had more than 1000 samples each. The rest of the 3 families were collected from a huge dataset using VirusShare [39]. The dataset has over 120,000 malware executable and is around 100 Gigabytes in size. We selected Renos, VBInject and OnLineGames from this dataset.

We used `objdump` which is a command line program part of the GNU Binary Utilities library for Unix-like operating systems. It is used to disassemble executables into Assembly code and hence extract the mnemonic opcodes. An example of how this is used is shown in Figure 5.

For the sake of the example we limit the output to 5 opcodes using `head -n 5` argument. We ran a Python script which recursively extracted up to 8000 opcodes from every file belonging to the 3 families. A summary of our dataset along with each

```
hegitrehan@instance-1:~/opcode_extract$ objdump -d VirusShare_ExampleFile |
sed '/[^\t]*\t[^\t]*\t/!d' | cut -f 3 | sed 's/ .*$//' | head -n 5
mov
ret
int3
int3
int3
```

Figure 5: Example usage of objdump.

Table 3: 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

malware family's type is given in Table 3.

4.2 HMM Implementation

We needed to solve all 3 HMM problems for our tasks. The solution to problem 3 allowed us to train an HMM model so that it represents our observation sequence, which is opcodes from each family. The solution to problem 1 allowed us to score data samples and evaluate the quality of our model. Finally, the solution to problem 2 allowed us to use the γ matrix so that we can find the hidden state sequence.

The HMM algorithm was implemented as per the algorithm given in [20]. We wrote the code in C++ since training HMM is an expensive task and high-level programming languages such as Python are slow compared to C++. We wrote a Python script to preprocess our data and create our 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 5 families in our dataset.

Table 4 shows the number of unique opcodes in each observation sequence. The number of unique opcodes for each family is very high and setting M to such large values makes training of HMM models computationally infeasible. So we experimented

with selecting the top n most frequent opcodes from the observation sequence, where, $n \in \{20, 21, ..., 40\}$. Figure 6 shows the top 20 opcodes from our combined observation

Table 4: Unique Opcodes in observation sequences for each family

Malware Family	Unique Opcodes
OnLineGames	284
Renos	208
VBInject	365
WinWebSec	122
Zbot	175

sequences and the number of families a particular opcode is present in. As we can see, out of the top 20 opcodes, 19 of them are present in all 5 families. Similarly, Figure 7 shows this distribution for the top 40 opcodes. Out of the top 40 opcodes, 26 of them are present in all 5 families. So we decided that the range for $M \in \{20, 21, ..., 40\}$ is the best choice.

Next, we converted the mnemonic opcodes to integers so that we can use them to solve Problem 3 of HMM, which is training the model to best fit the observation sequence. For each value of M as described above, we took the $M - 1$ most frequent opcodes and mapped them to integer $\{0, 1, .., M - 2\}$. Every other opcode which is not present in the top $M - 1$ was labelled as "other", i.e., $M - 1$. For example, when $M = 20$, we mapped the top 19 opcodes to integers $\{0, 1, .., 18\}$ and then the opcodes not present in the top 19 were marked as 19. This gives us 20 unique observation symbols $\in \{0, 1, .., 19\}$.

We set the value of $N = 2$ for our initial experiments which were to figure out the best value of M. Now, we have different values of $M \in \{20, 21, ..., 40\}$ and value of $N = 2$. For each family, this gives us 21 different HMM models of different dimensions

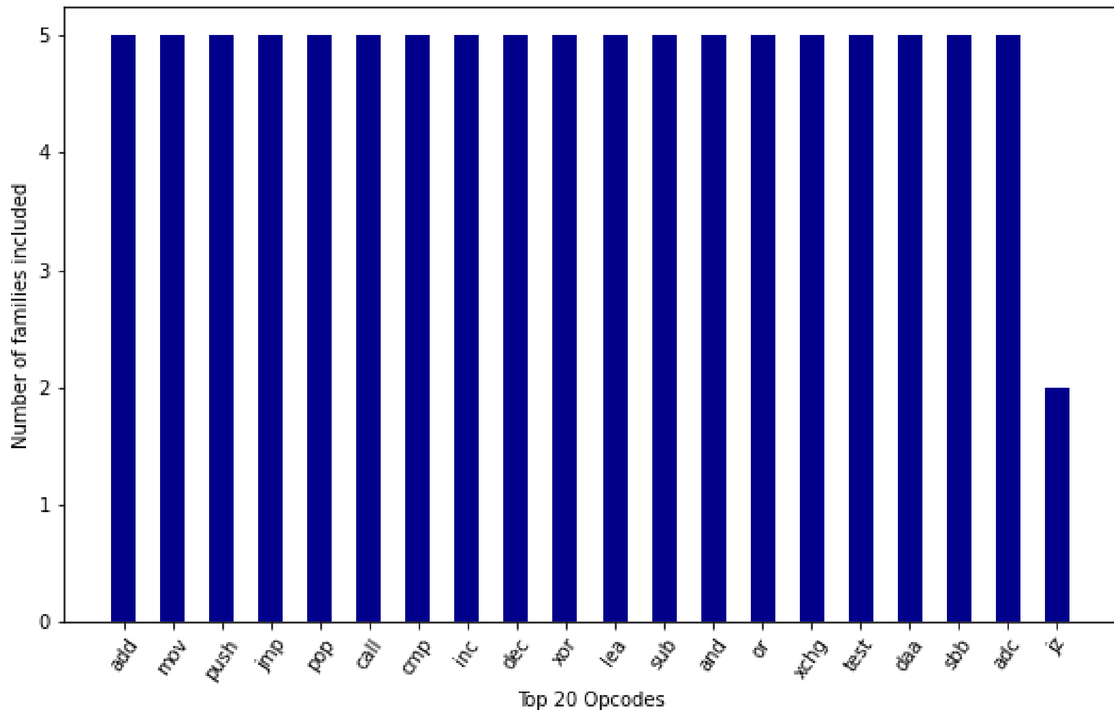

Figure 6: Top 20 opcodes from the observation sequences and the number of families they are present in.

for each family. The models were trained with 50000 random restarts so that we can maximize the chance of finding the global maxima. After training the models for a family, we tested them as follows:

- We sampled 500 malware samples (their extracted opcode sequences) from the family that the HMM model belongs to. These are the true samples so we labelled them as +1.
- We sampled 100 samples each from the rest of the 4 families, giving us a total of 400 samples. These are the false samples so we labelled them as -1.
- We sampled 100 samples from the benign dataset and labelled them as -1 as well.
- Finally, we have 500 true samples (labelled 1) and 500 false samples (labelled -1), giving a total of 1000 samples to score. Every sample consisting of opcode

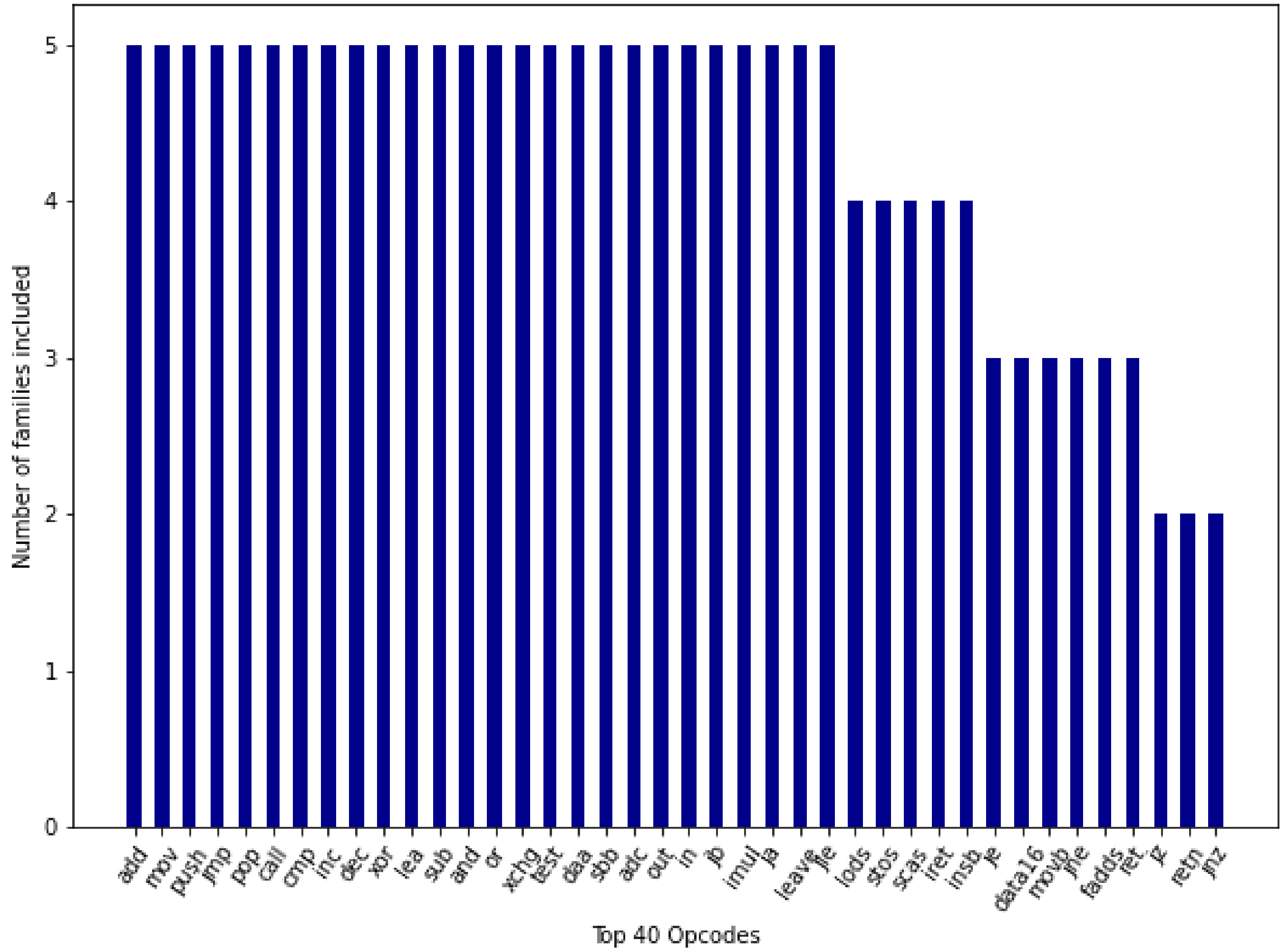

Figure 7: Top 40 opcodes from the observation sequences and the number of families they are present in.

sequences was truncated to a length of 1200 because each sequence has a different length.

- We calculated the log likelihood of each sample belonging to a model and plotted the ROC curve. Finally, the AUC was calculated from these ROC curves.
- The model with the highest AUC was chosen and the value of M, say M', from that model was used for the rest of the experiments.

The above steps were repeated for each family. The optimum of M for each family

served as the dimensions of our HMM model in the next set of experiments: $N = M = M'$.

Our next experiments were to solve Problem 2 of HMM so that we can 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' is the best value of M for each individual family.

We trained 10 different HMM models, each with 5000 random restarts for each malware family. All 10 of these models were scored the same way as explained above, using 500 true samples and 500 false samples. Out of these 10 models, we selected the 2 best models with the highest AUC value. The γ matrix from these 2 models was used to find the most likely hidden state sequence. Each model gives us a sequence of 30,000 length. 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 3 GAN architectures in Python using TensorFlow and Keras with TensorFlow backend. For vanilla GAN or just 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 so they were chosen. The loss function used was Binary Crossentropy as it is equivalent to the loss function for GAN as given in Equation (3). The models were trained for 10000 epochs.

For GANs the use of Batch Normalization [40] 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

dense_input: InputLayer | input: [(?, 600)] | output: [(?, 600)]
dense: Dense | input: (?, 600) | output: (?, 400)
dense_1: Dense | input: (?, 400) | output: (?, 200)
dense_2: Dense | input: (?, 200) | output: (?, 1)

Figure 8: GAN Discriminator Architecture.

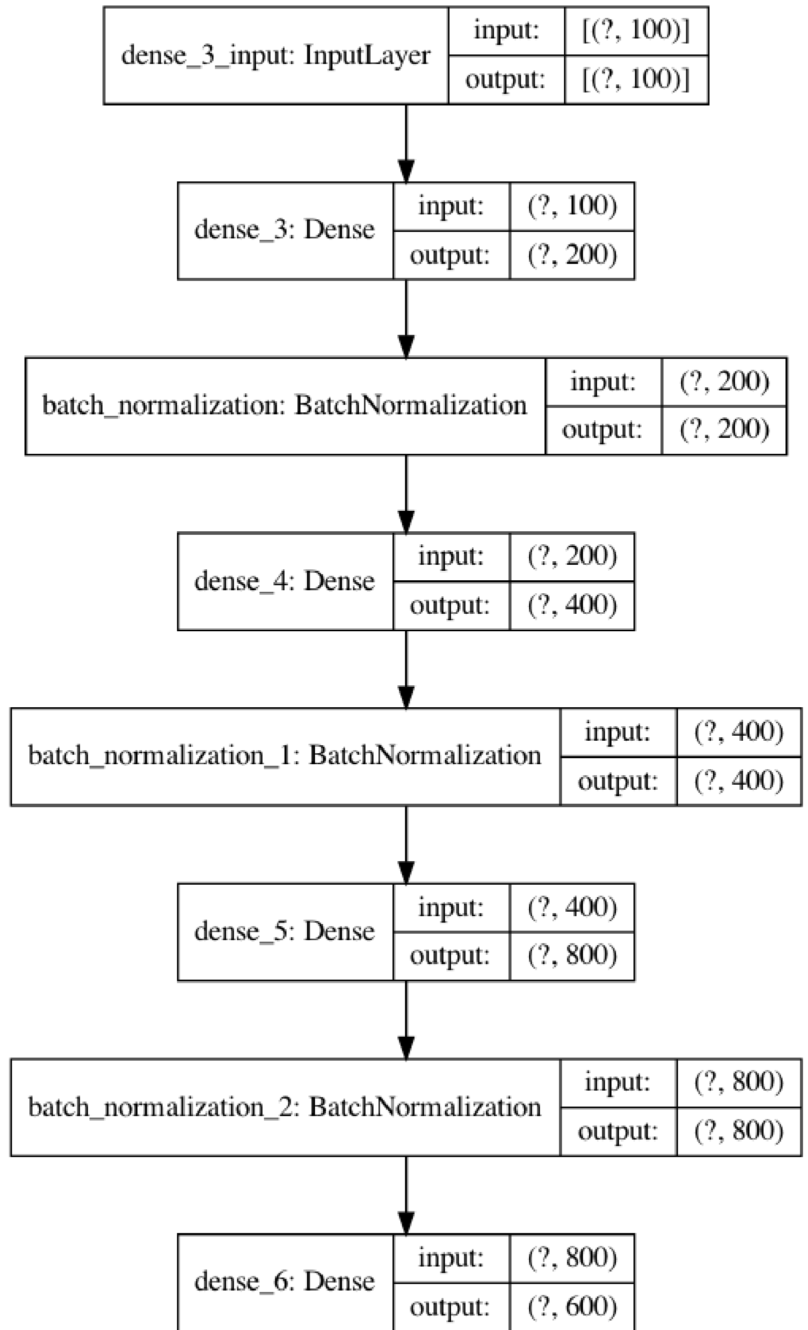

Figure 9: GAN Generator Architecture.

minibatch to minibatch. The overall architecture of our discriminator and generator models is given in Figures 8 and 9.

For the discriminator we have 1 input layer, 2 fully connected hidden layers and

an output layer with just 1 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. LeakReLU has a hyperparameter called alpha which is used to scale negative outputs. We used $\alpha = 0.2$ for our experiments. LeakyReLU activation function is:

$$f(x) = \begin{cases} x, \ x \geq 0 \\ \alpha x, \ x < 0 \end{cases}$$

The generator has 1 input layer, 3 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 want from the generator. We experimented with different layers for both the networks including Convolutional 1D layers and fully connected Dense layers had the best performance.

4.3.1 GAN Stabilizing Techniques

We further utilized stabilizing techniques to improve GAN training. All the techniques are discussed in [36] which was published in 2016 by some of the co-authors of the original 2014 paper on GANs [11]. The techniques were:

1. **Minibatch Discrimination:** In minibatch discrimination, instead of training the discriminator on a batch of real and fake samples combined, the training is split into 2 steps where 1st we train the discriminator on a batch of real samples only and then on a batch of fake samples only. This techniques helps to prevent

the generator from collapsing.

2. **Label Smoothing:** Instead of using just "1" as the ground truth labels for real samples, labels are scaled between a range close to 1, for example between $[0.7, 1.2]$. This is a common technique now days for deep neural network training since it prevents over fitting of data.

3. **Label Switching:** In label switching, after every few epochs the labels for real and fake samples are swapped for 1 epoch and then swapped back to normal. This helps to keep the discriminator from becoming too confident. If the discriminator is over confident, then the loss dips to 0 and generator stops receiving any feedback to update it's weights. This is known as failure mode.

4.4 WGAN Implementation

For WGAN, we used RMSProp optimizer. RMSProp is recommended by the paper authors 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. Figure 10 shows the architecture for both the critic and generator. We trained each WGAN model for 100000 epochs using minibatches of data.

The actual models are compiled and trained separately for the critic and generator. They are together in the figure just to show the architecture. 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 is done so that the loss function in Equation (6) can be computed

easily when implementing the WGAN algorithm given in Algorithm 2. These layers and networks gave the best result so we chose these as our final networks.

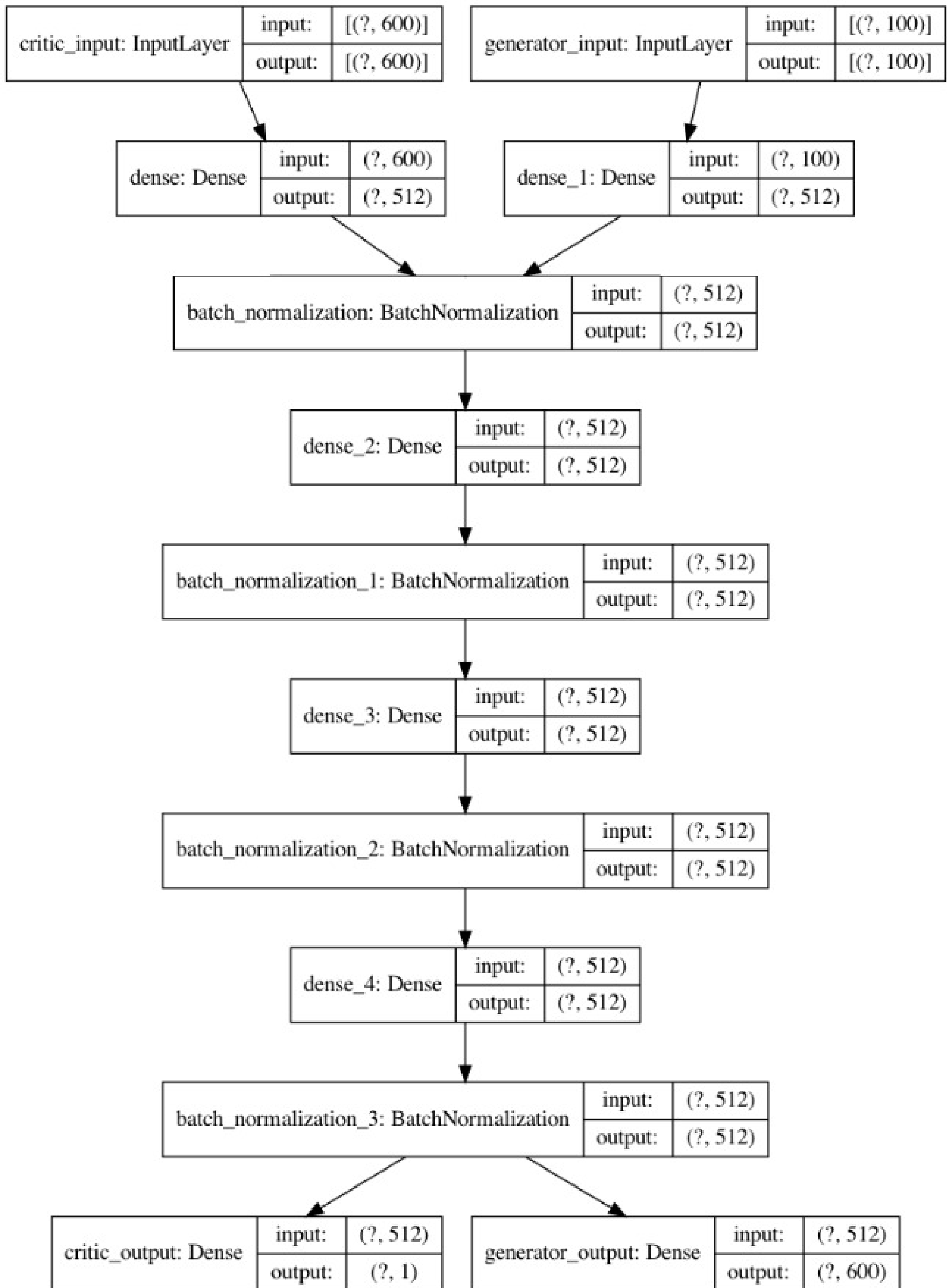

Figure 10: WGAN Critic and Generator Architecture.

4.4.1 Wasserstein Distance

The loss function or the Wasserstein distance between real and fake samples given in Equation (6) 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, then 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 (Equation (6)) 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 fake samples. The code snippet for this implementation can be found in Appendix C.

4.5 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:

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

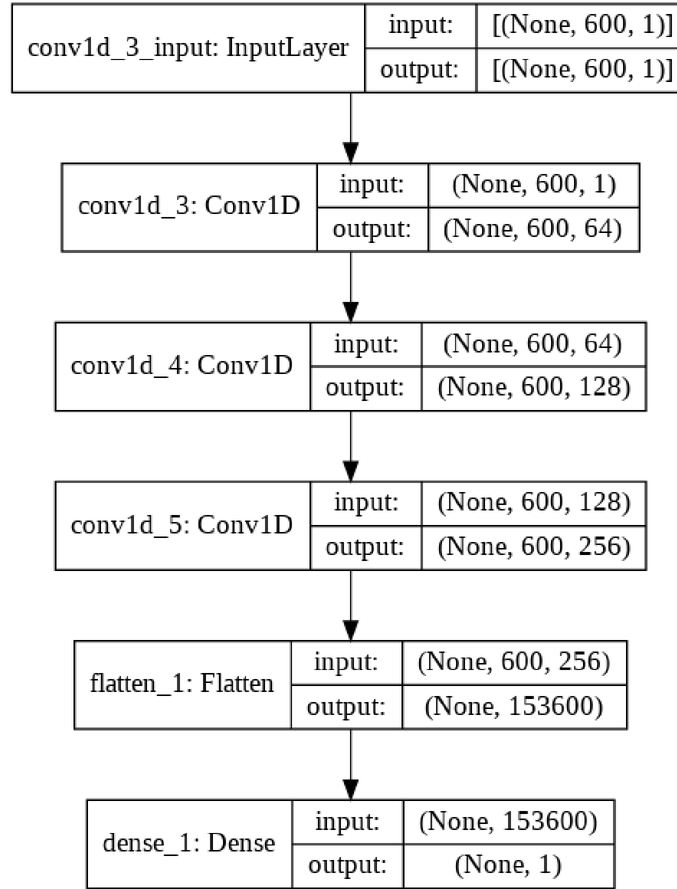

Figure 11: WGAN-GP Critic Architecture.

We trained each WGAN-GP model using minibatches for 100000 epochs. The architectures for the critic and generator model are given in Figures 11 and 12. 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 3 hidden Conv1D layers with 64, 128 and 256 filters and filter size 3. In the generator network also we used 3 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 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. Calculating the Wasserstein loss is part of the WGAN-GP loss function given in Equation (8). The first 2 operands on the right hand side are the Wasserstein distances which are calculated the same way as mentioned in Section 4.4.1. The last or the 3rd operand on the right side is the actual gradient penalty.

In the actual implementation, the calculation of the penalty is very straightforward. Using Keras backend ops we get the gradients for the critic network with respect to the averaged/interpolated image. Then we calculate the Euclidean norm or L2-norm of these gradients. Finally, the penalty is calculated: $(1 - L2_norm)^2$. Appendix C has the code snippet which shows the actual implementation of the gradient penalty loss in Python using TensorFlow and Keras.

Finally, the authors advised against the use of Batch Normalization in the critic network. They suggested that if required, Layer Normalization layer could be used. We experimented with Layer Normalization layer but the performance degraded so we decided not to use it. For the generator we still used Batch Normalization layer.

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

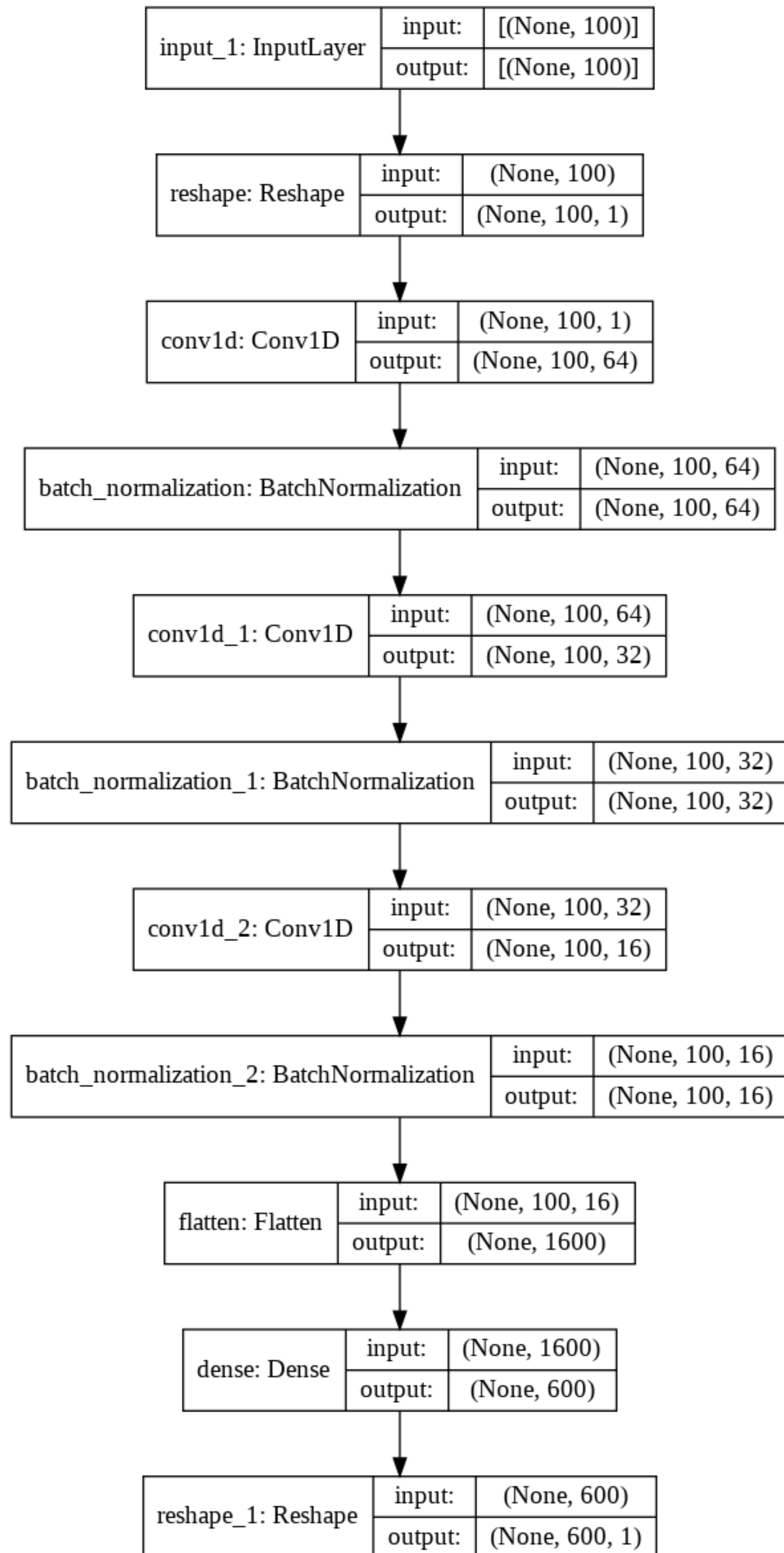

Figure 12: WGAN-GP Generator Architecture.

CHAPTER 5

Results and Discussion

In this chapter, we give the results of our experiments.

5.1 HMM Results

The first set of experiments were conducted to determine the optimum 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.

5.1.1 Optimum M and HMM Training Results

The summary of the results and the best value of M chosen for each family is given in Table 5. Appendix A contains ROC curves for each value of M for each family and also bar plots comparing the AUC scores of the 21 models from each family.

Table 5: 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 5.

Our next experiments were to train 10 different HMM models with dimensions as mentioned above and choose the 2 best models out of 10. Figures 13, 14, 15, 16 and 17 show the AUC scores for 10 models from each family. The models chosen

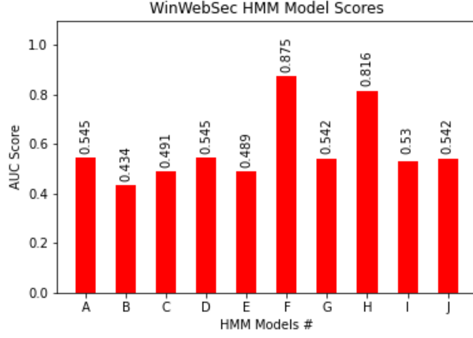

Figure 13: WinWebSec 22×22 models.

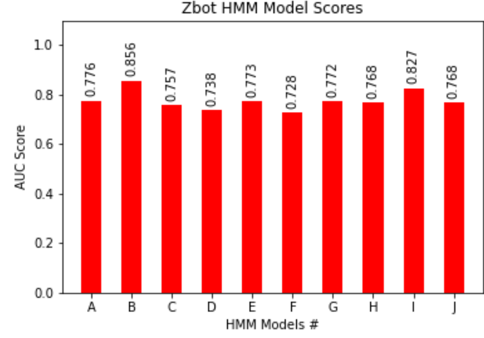

Figure 14: Zbot 20×20 models.

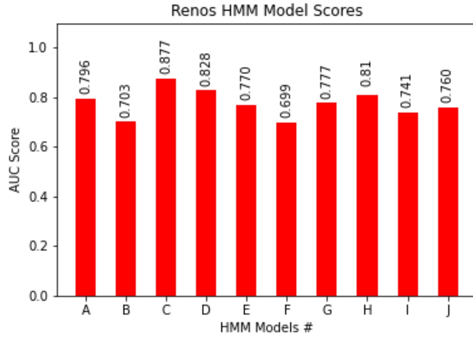

Figure 15: Renos 22×22 models.

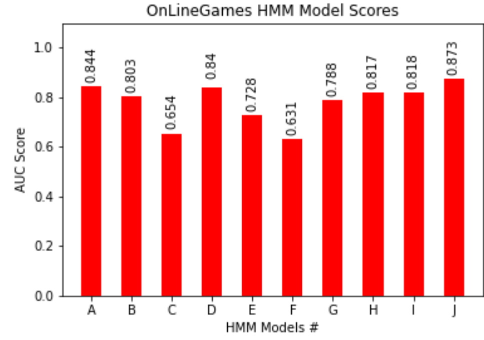

Figure 16: OnLineGames 22×22 models.

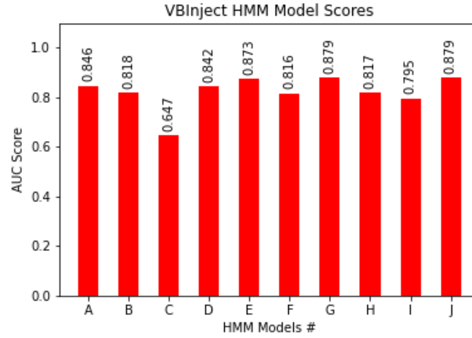

Figure 17: VBInject 25×25 models.

and their AUC scores are summarized in Table 6. We chose the 2 highest scoring models and calculated their most likely hidden state sequence using the γ matrix from the models. After breaking the 2 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. Low accuracy, precision and recall scores mean that the model

Table 6: Best 2 models for fake samples from each family.

Malware Family	Models			
	Model#	AUC	Model#	AUC
WinWebSec	F	0.875	H	0.816
Zbot	B	0.856	I	0.827
Renos	C	0.877	D	0.828
OnLineGames	A	0.844	J	0.873
VBInject	E	0.873	G	0.879

isn't able to differentiate between real and fake samples. Results from each of the 4 algorithms are given in the following section.

5.1.2 HMM Classification Results

We first performed hyperparameter tuning for the 4 machine learning algorithms and fixed the best parameters for the rest of the experiments.

1. **SVM:** Grid search on values of C, kernel and degree with values: $C \in \{1, 2, \ldots, 10\}$, $kernel \in \{rbf, poly, linear\}$ and $degree \in \{2, 3, 4, 5\}$. We found that polynomial kernels were overfitting the data so 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 number of decision trees to use and maximum depth of trees with values: number of trees $\in \{10, 20, \ldots, 80\}$ and 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 number of neighbors to consider, k, with values: $k \in \{4, 5, \ldots, 20\}$. $k = 8$ worked well and the distance metric chosen was Euclidean.

The classification results for each feature for each machine learning technique are given below. We used 5-fold cross validation and the scores given are the average scores from 5-fold cross validation. Table 7 gives the classification results for SVM. Table 8 gives the classification results for Naïve Bayes. Table 9 gives the classification results for Random Forest. Table 10 gives the classification results for k-NN.

Table 7: SVM-HMM scores for each feature.

Malware Family	Features								
	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	1.00	1.00	1.00	0.99	1.00	0.98	0.99	1.00	0.99
Zbot	0.97	0.99	0.95	0.98	1.00	0.97	0.93	0.95	0.91
Renos	1.00	1.00	1.00	0.98	0.99	0.98	0.97	0.97	0.97
OnLineGame	1.00	1.00	1.00	0.99	1.00	0.99	0.99	0.99	1.00
VBInject	1.00	1.00	1.00	0.98	0.98	0.98	0.96	0.93	0.99

Table 8: Naïve Bayes-HMM scores for each feature.

Malware Family	Features								
	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	0.99	1.00	0.98	0.99	0.99	1.00	0.99	1.00	0.98
Zbot	0.79	0.80	0.73	0.99	0.98	1.00	0.72	0.77	0.63
Renos	0.97	0.99	0.94	0.97	0.97	0.96	0.78	0.82	0.72
OnLineGame	0.87	0.97	0.76	0.96	0.93	0.99	0.93	0.92	0.94
VBInject	0.99	1.00	0.98	0.96	0.96	0.97	0.94	0.90	1.00

Using Word2Vec features, SVM, Random Forest and k-NN classifiers were able to differentiate between real and fake samples efficiently. However, Naïve Bayes classifier had low recall rates for Zbot (73%) and OnLineGames (76%). We attribute

Table 9: Random Forest-HMM scores for each feature.

Malware Family	Features								
	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	0.99	1.00	0.99	1.00	1.00	1.00	0.99	1.00	0.99
Zbot	0.97	0.96	0.99	0.99	1.00	0.99	0.92	0.95	0.88
Renos	0.98	0.98	0.99	0.98	1.00	0.97	0.95	0.95	0.96
OnLineGame	0.99	1.00	0.98	0.99	1.00	0.99	0.96	1.00	0.91
VBInject	1.00	1.00	1.00	0.99	0.99	1.00	0.93	0.96	0.89

Table 10: kNN-HMM scores for each feature.

Malware Family	Features								
	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	0.98
Zbot	0.92	0.91	0.93	0.96	0.95	0.97	0.82	0.95	0.69
Renos	1.00	1.00	1.00	0.97	0.95	1.00	0.71	1.00	0.42
OnLineGame	0.97	1.00	0.95	0.96	0.93	1.00	0.68	1.00	0.36
VBInject	1.00	1.00	1.00	0.97	0.96	0.99	0.59	1.00	0.19

this result to the ineffectiveness of the classifier rather than the quality of fake samples because the other 3 algorithms were able to differentiate very well.

Using Bigram features, all 4 classifiers were able to differentiate between real and fake samples very effectively with high precision and recall.

Using integer vectors, SVM and Random Forest precision and recall rates dipped a little but they were still able to classify real and fake samples effectively. Naïve Bayes and k-NN scores were quite low. As we have seen, Naïve Bayes is a weak classifier as compared to the other 3 and we attribute these low scores to integer

vectors being a weaker feature representation for the data. The low recall rates for k-NN and Naïve Bayes means that a lot of real samples were classified as fake (high false negative). On the other hand, the high precision rates for these 2 classifiers means that very few fake samples were classified as real (low false positive).

5.2 GAN Results

We experimented with the stabilizing techniques mentioned in Section 4.3.1. Although the training stabilized across all 5 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 don't necessarily indicate or correspond to the model's performance or quality of the generated samples. The discriminator and generator loss for GAN for each family are given in Appendix B.

5.2.1 Best GAN Generative Model

We used a subset of the techniques for each family to get the best results in generating fake samples. Minibatch discrimination was used for all 5 families. Label smoothing and label switching were only used for Zbot, Renos and VBInject. The best generative model was chosen for each family as discussed in Section 3.4.2. Table 11 summarizes the model chosen identified by the epoch number for each family.

Table 11: Best GAN generative model for each family

Malware Family	Epoch Number
WinWebSec	1000
Zbot	400
Renos	1800
OnLineGames	200
VBInject	200

5.2.2 GAN Classification Results

Fake samples were generated using the chosen models in Table 11 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 HMM results section (Section 5.1.2) and tested the fake samples using all 3 features mentioned above. Table 12 gives the results for SVM model. Table 13 gives the results for Naïve Bayes model. Table 14 gives the results for Random Forest model. Table 15 gives the results for k-NN model.

Table 12: SVM-GAN scores for each feature.

Malware Family	Features								
	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	1.00	1.00	1.00	1.00	1.00	1.00	0.80	0.82	0.81
Zbot	1.00	1.00	1.00	0.91	0.91	0.94	0.83	0.91	0.77
Renos	0.98	1.00	0.97	1.00	1.00	1.00	0.89	0.82	0.94
OnLineGame	1.00	1.00	1.00	0.95	0.97	0.93	0.73	0.89	0.62
VBInject	0.98	1.00	0.97	0.97	0.95	1.00	0.75	0.79	0.72

Using Word2Vec and Bigram features, the scores for all 4 families dipped a little as compared to the HMM results. SVM and Random Forest have accuracy, precision and recall all above 90% for these 2 features, except for OnLineGames with 88% precision with Random Forest. Low precision rate means high false positive rate which is the most desirable result for us. Naïve Bayes has low overall scores for Word2Vec and Bigram features on account of it being a weaker classifier. Interestingly, k-NN has the lowest overall scores for these 2 features. This can be attributed to the way k-NN algorithm works and that the generated data's distribution is slightly closer to

Table 13: Naïve Bayes-GAN scores for each feature.

Malware Family	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	0.87	0.86	0.92	0.98	0.96	1.00	0.69	0.70	0.66
Zbot	0.98	0.96	1.00	0.83	0.81	0.84	0.70	0.71	0.65
Renos	0.98	1.00	0.97	0.92	0.88	0.98	0.75	0.73	0.73
OnLineGame	0.97	1.00	0.94	0.89	1.00	0.80	0.50	0.53	0.57
VBInject	0.95	0.91	1.00	0.84	0.93	0.75	0.73	0.77	0.72

Table 14: Random Forest-GAN scores for each feature.

Malware Family	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	0.97	0.96	1.00	0.98	1.00	0.96	0.67	0.68	0.63
Zbot	1.00	1.00	1.00	0.95	0.96	0.97	0.86	0.91	0.80
Renos	0.98	1.00	0.96	0.97	0.93	1.00	0.92	0.93	0.92
OnLineGame	1.00	1.00	1.00	0.98	0.97	1.00	0.85	0.84	0.84
VBInject	1.00	1.00	1.00	0.92	0.88	0.98	0.89	0.93	0.84

the real data's distribution as compared to HMM fake samples.

For integer vectors, we see that all 4 classifiers were not able to differentiate very well between real and fake samples. As seen with HMM, 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 isn't able to distinguish between fake and real at all based on just the integer vectors.

Table 15: kNN-GAN scores for each feature.

Malware Family	Features								
	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	0.88	0.90	0.79	0.97	1.00	0.95	0.61	0.80	0.24
Zbot	0.94	0.97	0.93	0.89	0.90	0.90	0.77	0.96	0.59
Renos	0.97	1.00	0.92	0.92	0.96	0.89	0.50	0.00	0.00
OnLineGame	0.94	1.00	0.87	0.94	0.91	0.98	0.65	0.83	0.40
VBInject	1.00	1.00	1.00	0.88	0.82	0.92	0.64	1.00	0.28

5.3 WGAN Results

Unlike vanilla GAN, the loss values when training WGAN give reliable information about the model's progress and convergence. So for WGAN and WGAN with Gradient Penalty we first discuss the loss curves and convergence and then give the classification results for the 4 machine learning techniques.

5.3.1 Convergence and Loss Values

The loss values for all 5 WGAN models that we trained; 1 for each family; were interesting. Figures 18 and 19 show the critic and generator loss curves respectively for WinWeSec. Appendix B contains the loss curves for the rest of the 4 families.

The loss value for the critic and the generator converges very fast, in the first few epochs and then stays same for the rest of the 100k epochs. We tried a lot of different hyperparameters, such as changing the value of "n_critic", that is the number of critic iterations per generator iteration, different clipping value and different learning rates. Even changing the networks entirely and using Convolutional 1D instead of fully connected Dense layers didn't help. The value of loss didn't change after the first few epochs. This shows that clipping the weights is a major drawback in WGAN (Section 2.4.3) as it saturates the model, and the weights don't update after a point.

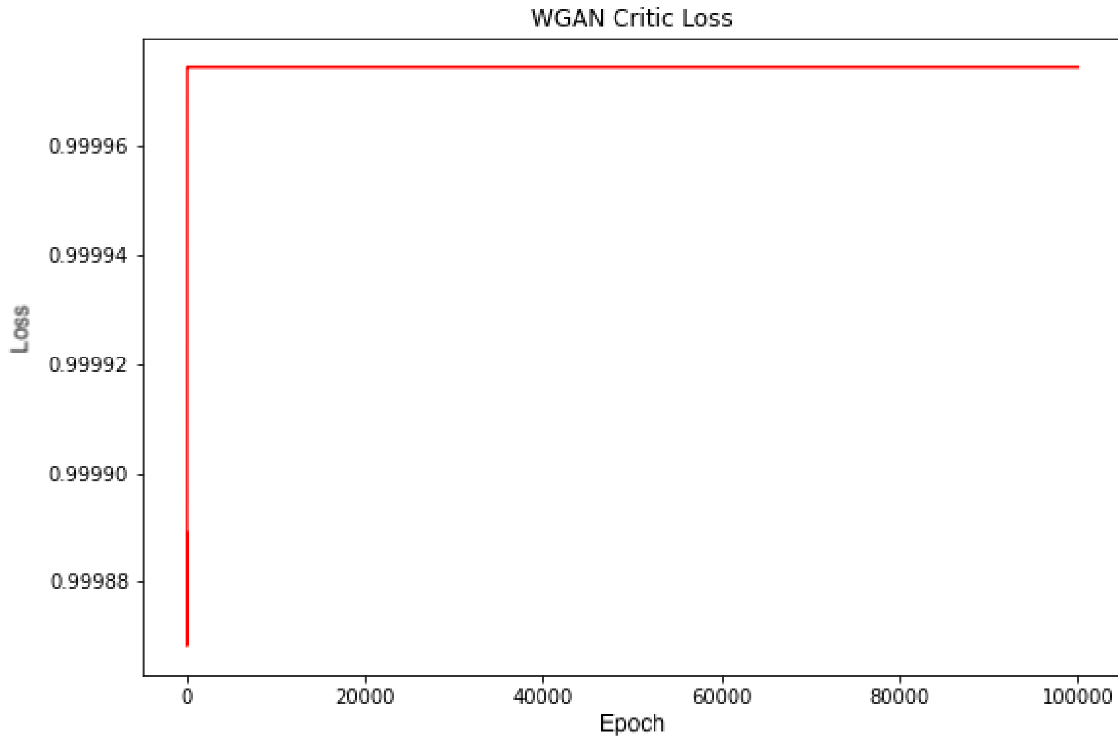

Figure 18: WinWebSec WGAN critic loss.

Any change in weight is nullified by the clipping step. Interestingly, all 4 families converge to the same loss value for the critic and generator. The clipping step stops the training since the weights can't change beyond the clipping range and don't respond to the gradient updates that are back propagated through the network.

5.3.2 WGAN Classification Results

The best generative model from WGANs was chosen for each family as discussed in Section 3.4.2. Table 16 summarizes the best model chosen for each family. The classification results for the fake samples generated by the selected WGAN models are given below. We used the same hyperparameters as discussed in the HMM results section (Section 5.1.2) and tested the fake samples using all 3 features in batches of 32. Table 17 gives the results for SVM model. Table 18 gives the results for Naïve Bayes model. Table 19 gives the results for Random Forest model. Table 20 gives the

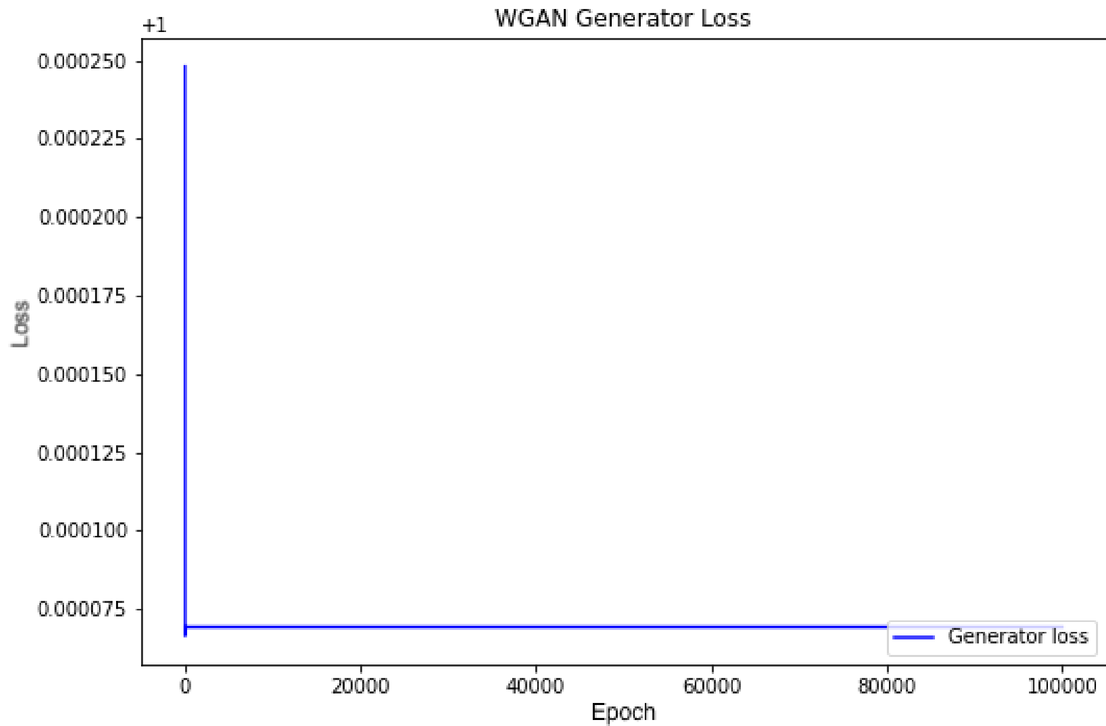

Figure 19: WinWebSec WGAN generator loss.

results for k-NN model.

Table 16: Best WGAN generative model for each family.

Malware Family	Epoch Number
WinWebSec	19000
Zbot	29500
Renos	97000
OnLineGames	11500
VBInject	45000

Using Word2Vec and Bigram features SVM and Random Forest were able to very effectively differentiate between real and fake samples generated by WGAN. Interestingly, even Naïve Bayes and k-NN performed very well and we have seen from previous results that they are the 2 weaker classifiers. This means that the WGAN

Table 17: SVM-WGAN scores for each feature.

Malware Family	Features								
	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	1.00	1.00	1.00	0.98	1.00	0.97	0.97	1.00	0.94
Zbot	1.00	1.00	1.00	0.98	0.98	1.00	0.86	1.00	0.72
Renos	0.98	1.00	0.96	0.98	0.97	1.00	0.98	1.00	0.97
OnLineGame	1.00	1.00	1.00	0.97	1.00	0.94	1.00	1.00	1.00
VBInject	1.00	1.00	1.00	1.00	1.00	1.00	0.95	0.96	0.96

Table 18: Naïve Bayes-WGAN scores for each feature.

Malware Family	Features								
	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	1.00	1.00	1.00	0.97	1.00	0.93	0.95	1.00	0.91
Zbot	1.00	1.00	1.00	0.98	1.00	0.98	0.84	1.00	0.73
Renos	1.00	1.00	1.00	0.98	1.00	0.95	0.95	1.00	0.88
OnLineGame	1.00	1.00	1.00	0.98	1.00	0.97	0.97	0.93	1.00
VBInject	0.94	0.96	0.93	0.98	0.97	1.00	0.62	1.00	0.25

fake samples are of inferior quality compared to HMM and GAN.

Using integer vectors, the results for SVM and Random Forest were very high. Again, integer vectors have proven to be weak feature representations that make classification hard but in the case of WGAN even they are easily differentiable. For k-NN and Naiïve Bayes with integer vectors, we see extremely low recall rates for some families such as 25% for VBInject, 37% for OnLineGames and 53% for Renos. But these low recall rates are accompanied by high precision rates, almost 100% across the board for all families.

Table 19: Random Forest-WGAN scores for each feature.

Malware Family	Features								
	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	1.00	1.00	1.00	1.00	1.00	1.00	0.98	1.00	0.96
Zbot	1.00	1.00	1.00	0.95	0.97	0.94	0.94	1.00	0.85
Renos	0.97	1.00	0.96	0.92	0.92	1.00	0.92	1.00	0.85
OnLineGame	0.98	1.00	0.97	0.98	1.00	0.96	0.97	1.00	0.93
VBInject	1.00	1.00	1.00	0.94	0.91	0.98	0.97	1.00	0.93

Table 20: kNN-WGAN scores for each feature.

Malware Family	Features								
	Word2Vec			Bigrams			Integer Vectors		
	Acc.	Prec.	Recall	Acc.	Prec.	Recall	Acc.	Prec.	Recall
WinWebSec	1.00	1.00	1.00	0.89	0.88	0.92	1.00	1.00	1.00
Zbot	0.97	1.00	0.94	0.98	1.00	0.96	0.81	1.00	0.65
Renos	0.98	1.00	0.97	0.94	0.92	0.98	0.75	1.00	0.53
OnLineGame	0.97	1.00	0.93	1.00	1.00	1.00	0.67	1.00	0.37
VBInject	0.98	1.00	0.96	0.98	0.98	1.00	0.62	1.00	0.25

5.4 Wasserstein GAN with Gradient Penalty

As with WGAN, the critic's loss value helps monitor the model's performance for WGAN with gradient penalties. The WGAN-GP paper mentions that the the critic's loss (negative of it) should start at a large number and then converge towards 0. The generator's loss is not very insightful and can fluctuate. So, first we discuss the loss curves and then give the classification results.

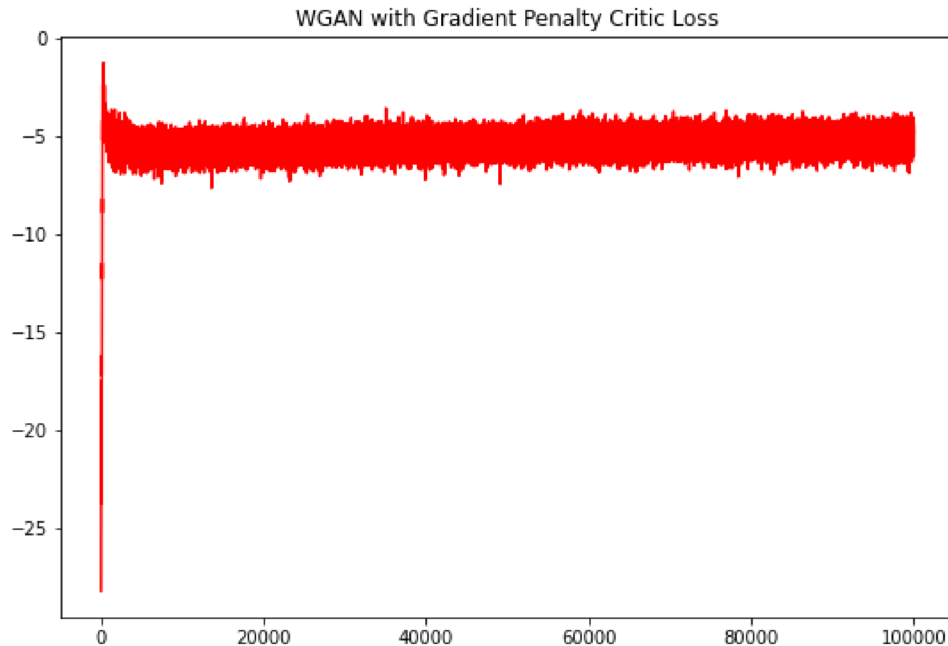

Figure 20: WinWebSec WGAN-GP critic loss.

5.4.1 Convergence and Loss Curves

The loss curves for all 5 families have a similar shape. Figures 20 and 21 show the loss curves for WinWebSec critic and generator respectively. Appendix B contains the loss curves for the rest of the 4 families.

We can see that the loss curve for the critic starts at around -28 and then slowly converges to around -4. This is the expected behavior and means that our model is training properly. Usually WGAN with Gradient Penalty take a long time to train, around 200k-300k epochs. We trained till 100k epochs since we were training 5 different models, 1 for each family.

The critic loss curves for the other 4 families (Appendix B) also have similar shapes but with slightly different values of convergence. Training the models for more epochs, around 200k-300k would be ideal for full convergence.

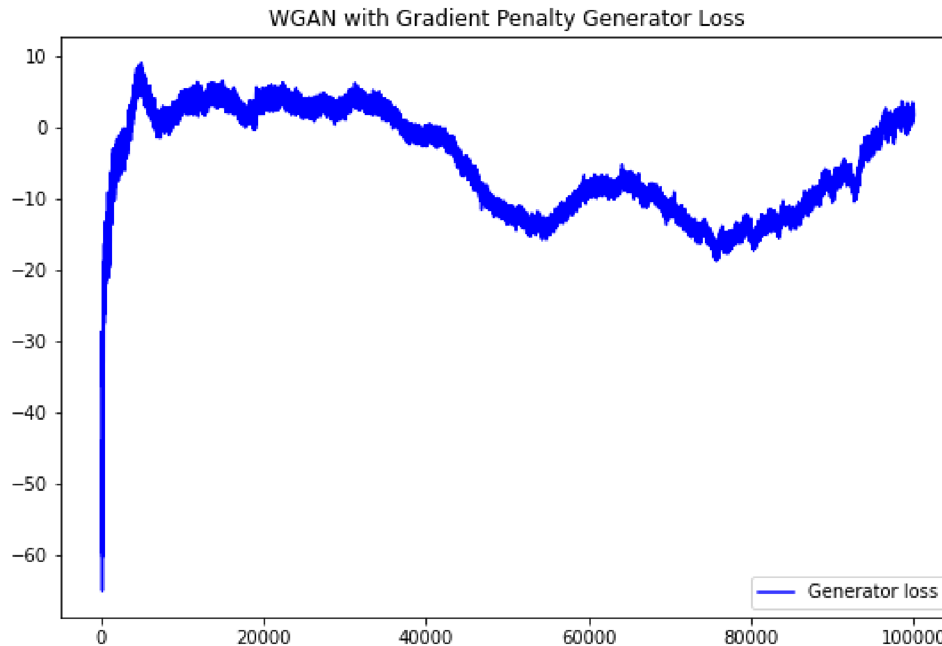

Figure 21: WinWebSec WGAN-GP generator loss.

The loss curve for the generator is not very informative about the model's performance and training and we can see that the loss values oscillate.

5.4.2 WGAN-GP Classification Results

The best generative model from WGAN-GPs was chosen for each family as discussed in Section 3.4.2. Table 21 summarizes the best model chosen for each family. The classification results for the fake samples generated by the selected WGAN-GP models are given below. We used the same hyperparameters as discussed in the HMM results section (Section 5.1.2) and tested the fake samples using all 3 features in batches of 32. Table 22 gives the results for SVM model. Table 23 gives the results for Naïve Bayes model. Table 24 gives the results for Random Forest model. Table 25 gives the results for k-NN model.

Using Word2Vec and Bigram features all 4 machine learning techniques weren't

Table 21: Best WGAN-GP generative model for each family.

Malware Family	Epoch Number
WinWebSec	86000
Zbot	77000
Renos	64000
OnLineGames	66000
VBInject	29000

Table 22: 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

Table 23: 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 24: 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 25: kNN-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

able to give very good classification results. Compared to WGAN and GAN the accuracy, precision and recall rates are much lower. This means that the quality of 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 4 classifiers we used. For Zbot, Renos and VBInject the accuracy for Random Forest is low $\sim 70\%$. For WinWebSec and OnLineGames the accuracy is also low at 82% and 81% for Word2Vec and even lower for Bigram at 81% and 74%. This is a promising result

since we have seen that classifying real vs fake samples using these 2 features has been very effective and we have gotten high accuracy and precision scores previously.

Using integer vector features the scores for SVM, Naïve Bayes and k-NN classifiers are very low. Basically these 3 models are not able to distinguish between real and fake samples just based on the integer representation. This is confirmed by accuracy scores in range of 50%-60% and even lower for Naïve Bayes at less than 50% for WinWebSec, Zbot, Renos and VBInject families. Random Forest did a better job as compared to the other 3 techniques but still the accuracy is around 70% for WinWebSec, Zbot, Renos and around 60% for OnLineGames and VBInject. This again shows that the quality of fake samples generated by WGAN-GP generative model is much better than the other GAN architectures and HMM.

CHAPTER 6

Conclusion and Future Work

In this project, we aimed at utilizing different generative modelling techniques to generate fake malware mnemonic opcode sequences. We utilized 4 different techniques: Hidden Markov Models, Generative Adversarial Networks, Wasserstein Generative Adversarial Networks and Wasserstein Generative Adversarial Networks with Gradient Penalty.

Previous work has shown that using malware as images, GANs can be utilized successfully to generate fake malware images but 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 as converting malware files to images and training GANs on images is computationally expensive. Opcode sequences provide a much simpler representation of the data.

We used 3 different feature extractions from malware opcode sequences: Word2Vec, Bigram and Integer Vectors. Classification results showed that Word2Vec and Bigram features give a very good representation of the malware data since for all 4 generative models the classification results were very high when tested with these 2 extracted features. Integer vectors, on the other hand are not a very good representation since they don't 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. Using Word2Vec and Bigram features these 3 classifiers had accuracy well above 90% for all 5 of the families. For integer vector features, the accuracy was slightly lower. Naïve Bayes classifier, on the other hand had much lower scores for all 3 features. This implies that Naïve Bayes is a weaker

classifier comparatively for malware classification using opcode sequences.

Using generative models from GAN we saw an improvement in the results as it was difficult for the classifiers to tell apart real and fake samples indicated by slightly lower classification scores. For WGAN on the other hand, the results were worse as compared to GAN. This is attributed to the weight clipping step in the WGAN algorithm. Weight clippings inhibits the critic network's ability to properly learn the real data's representation.

For WGAN with gradient penalty algorithm, we got the best results. We saw that the classification results for Word2Vec and Bigram features were the worst as compared to the other 3 generative models. For all 4 classifiers we got accuracy around 70%-80% with Word2Vec and Bigram features. From previous results we know that HMM, GAN and WGAN generative models were not able to properly learn the true data's distribution especially the Word2Vec and Bigram features. But WGAN-GP clearly performed better. For integer vectors the results were even better as the accuracy score dipped to 50%-60%. SVM classifier had accuracy of 41%, 48% and 44% for Renos, OnlineGames and VBInject families.

So, we conclude that we can use WGAN with Gradient Penalty algorithm to successfully generate fake malware opcode sequences such that they are close to the real data's distribution. This serves as a "proof of concept" that different GAN algorithms can be successfully applied to generate malware opcode sequences and that GANs are not succesful only for image data. The generative models can be used to boost malware datasets for some families that have very few data samples. Moreover, testing the quality of fake samples with Word2Vec and Bigram features is a better option as compared to integer vectors.

6.1 Future Work

We have shown that we can utilize GAN algorithms to generate malware opcode sequences. There are a lot of different directions that this research project can be expanded to:

- We experimented with 5 malware families: WinWebSec, Renos, Zbot, OnLineGames and VBInject. The dataset can be expanded and the experiments can be done on more malware families.
- We trained individual GAN models for each family. We can experiment with multi-class generative models.
- Trained generative models can be used to boost or augment the datasets for families that have less data samples. Lack of large datasets sufficient for deep neural networks is a common obstacle in malware research. Classifying malware families using existing datasets and comparing the results after classifying malware families with boosted datasets is a natural next step for this project. This will give insights to the actual effectiveness and provide an important use case for generated malware opcodes.
- Malware obfuscation is a pressing issue and the fake samples generated from our WGAN-GP algorithm can be treated as obfuscated malware because of the presence of noise in the generated opcode sequence. We can also add opcodes from benign samples into the generated samples and then experiment with actual obfuscated malware samples to boost the detection rate.
- Experiments with LSTM-GAN can be conducted since stateful networks can provide better results.

LIST OF REFERENCES

[1] SonicWall, ‘‘Sonicwall 2020 Cyber Threat Report,’’ https://www.sonicwall.com/news/2020-sonicwall-cyber-threat-report, 2020.

[2] N. Idika and A. Mathur, ‘‘A survey of malware detection techniques,’’ *Purdue University*, 03 2007.

[3] K. Pal and J. Verma, ‘‘A survey on anomaly based malware detection and demolition in false alarm rate,’’ 2015.

[4] A. K. Lab, ‘‘Heuristics Analysis,’’ https://usa.kaspersky.com/resource-center/definitions/heuristic-analysis, 2020.

[5] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, ‘‘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*, 2016, pp. 183--194.

[6] I. Santos, Y. K. Penya, J. Devesa, and P. G. Bringas, ‘‘N-grams-based file signatures for malware detection.’’ *ICEIS (2)*, vol. 9, pp. 317--320, 2009.

[7] Z. Sun, Z. Rao, J. Chen, R. Xu, D. He, H. Yang, and J. Liu, ‘‘An opcode sequences analysis method for unknown malware detection,’’ ser. ICGDA 2019. Association for Computing Machinery, 2019, p. 15–19.

[8] P. O’Kane, S. Sezer, and K. McLaughlin, ‘‘Obfuscation: The hidden malware,’’ *IEEE Security Privacy*, vol. 9, no. 5, pp. 41--47, 2011.

[9] D. Gibert, C. Mateu, and J. Planes, ‘‘The rise of machine learning for detection and classification of malware: Research developments, trends and challenges,’’ *Journal of Network and Computer Applications*, vol. 153, p. 102526, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804519303868

[10] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar, ‘‘Adversarial machine learning,’’ in *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, 2011, pp. 43--58.

[11] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, ‘‘Generative adversarial networks,’’ 2014.

[12] S. Yajamanam, V. R. S. Selvin, F. Di Troia, and M. Stamp, ‘‘Deep learning versus gist descriptors for image-based malware classification.’’ in *Icissp*, 2018, pp. 553--561.

[13] M. Jain, ‘‘Image-based malware classification with convolutional neural networks and extreme learning machines,’’ https://scholarworks.sjsu.edu/etd_projects/900/, Dec 2019.

[14] R. Burks, K. A. Islam, Y. Lu, and J. Li, ‘‘Data augmentation with generative models for improved malware detection: A comparative study*,’’ in *2019 IEEE 10th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, 2019, pp. 0660--0665.

[15] Y. Lu and J. Li, ‘‘Generative adversarial network for improving deep learning based malware classification,’’ in *2019 Winter Simulation Conference (WSC)*, 2019, pp. 584--593.

[16] W. Hu and Y. Tan, ‘‘Generating adversarial malware examples for black-box attacks based on gan,’’ 2017.

[17] L. Rabiner, ‘‘A tutorial on hidden markov models and selected applications in speech recognition,’’ *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257--286, 1989.

[18] C. Annachhatre, T. Austin, and M. Stamp, ‘‘Hidden markov models for malware classification,’’ *Journal of Computer Virology and Hacking Techniques*, vol. 11, 05 2014.

[19] A. Krogh, ‘‘An introduction to hidden markov models for biological sequences,’’ in *Computational methods in molecular biology*, S. Salzberg, D. Searls, and S. Kasif, Eds. United Kingdom: Elsevier, 1998, pp. 45--63.

[20] M. Stamp, ‘‘A revealing introduction to hidden markov models,’’ *Science*, pp. 1--20, 01 2004.

[21] M. Stamp, *Introduction to Machine Learning with Applications in Information Security*, 1st ed. Chapman & Hall/CRC, 2017.

[22] M. Arjovsky and L. Bottou, ‘‘Towards principled methods for training generative adversarial networks,’’ 2017.

[23] S. Mannor, D. Peleg, and R. Rubinstein, ‘‘The cross entropy method for classification,’’ in *Proceedings of the 22nd International Conference on Machine Learning*, ser. ICML ’05. Association for Computing Machinery, 2005, p. 561–568.

[24] D. P. Kingma and J. Ba, ‘‘Adam: A method for stochastic optimization,’’ 2014.

[25] J. Hui, "Gan - what is generative adversarial networks gan?" Dec 2019. [Online]. Available: https://jonathan-hui.medium.com/gan-whats-generative-adversarial-networks-and-its-application-f39ed278ef09

[26] A. Jabbar, X. Li, and B. Omar, "A survey on generative adversarial networks: Variants, applications, and training," *ArXiv*, vol. abs/2006.05132, 2020.

[27] P. M R and P. Jayagopal, "Generative adversarial networks: a survey on applications and challenges," *International Journal of Multimedia Information Retrieval*, vol. 10, 03 2021.

[28] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein gan," 2017.

[29] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, "Improved training of wasserstein gans," 2017.

[30] "K Neighbors Classifier," https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html, Accessed on: (04/20/2021).

[31] "KNN (K-Nearest Neighbors)," https://towardsdatascience.com/knn-k-nearest-neighbors-1-a4707b24bd1d, Accessed on: (04/20/2021).

[32] C. Cortes and V. Vapnik, "Support-vector networks," *Mach Learn 20*, pp. 273--297, 1995. [Online]. Available: https://doi.org/10.1007/BF00994018

[33] R. Gandhi, "Support vector machine," https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47, 2018.

[34] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, ser. COLT '92. New York, NY, USA: Association for Computing Machinery, 1992, p. 144–152. [Online]. Available: https://doi.org/10.1145/130385.130401

[35] S. Sawla, "Introduction to Naiïve Bayes for classification," https://medium.com/@srishtisawla/introduction-to-naive-bayes-for-classification-baefefb43a2d, 2018.

[36] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," 2016.

[37] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "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*, ser. NIPS'17. Curran Associates Inc., 2017, p. 6629–6640.

[38] A. Nappa, M. Z. Rafique, and J. Caballero, ‘‘The malicia dataset: identification and analysis of drive-by download operations,’’ *International Journal of Information Security*, vol. 14, no. 1, pp. 15--33, 2015.

[39] J.-M. Roberts, ‘‘VirusShare.com - Because Sharing is Caring,’’ http://www.virusshare.com, 2011.

[40] S. Ioffe and C. Szegedy, ‘‘Batch normalization: Accelerating deep network training by reducing internal covariate shift,’’ 2015.

APPENDIX A

ROC Curves and bar plots for optimum M.

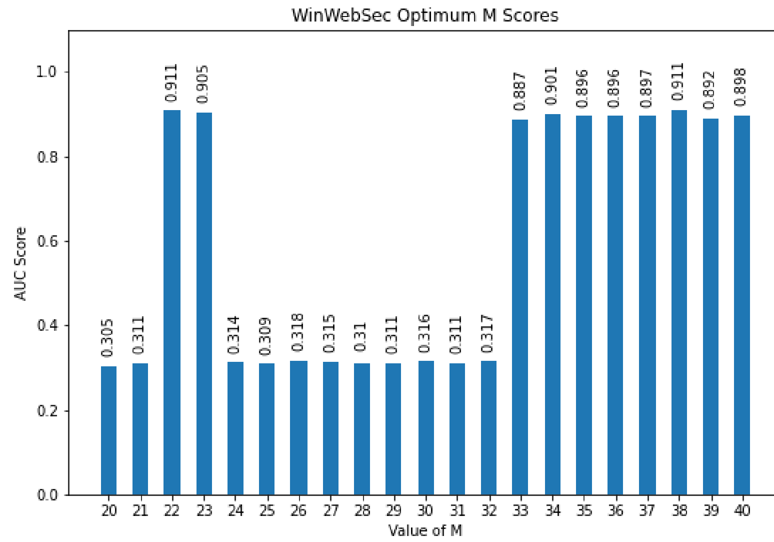

Figure A.22: AUC scores for different M values for WinWebSec.

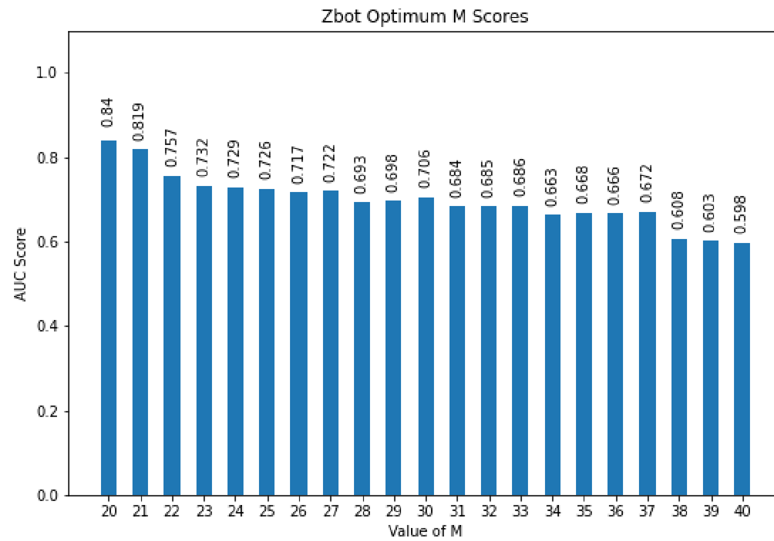

Figure A.23: AUC scores for different M values for Zbot.

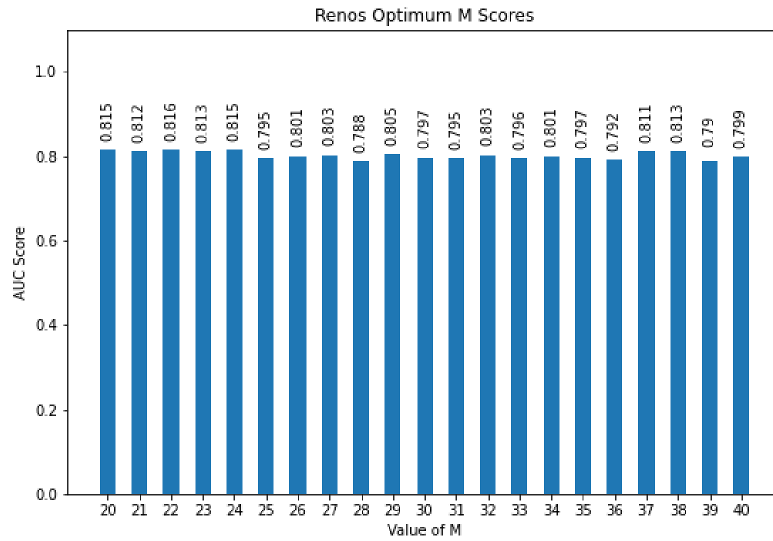

Figure A.24: AUC scores for different M values for Renos.

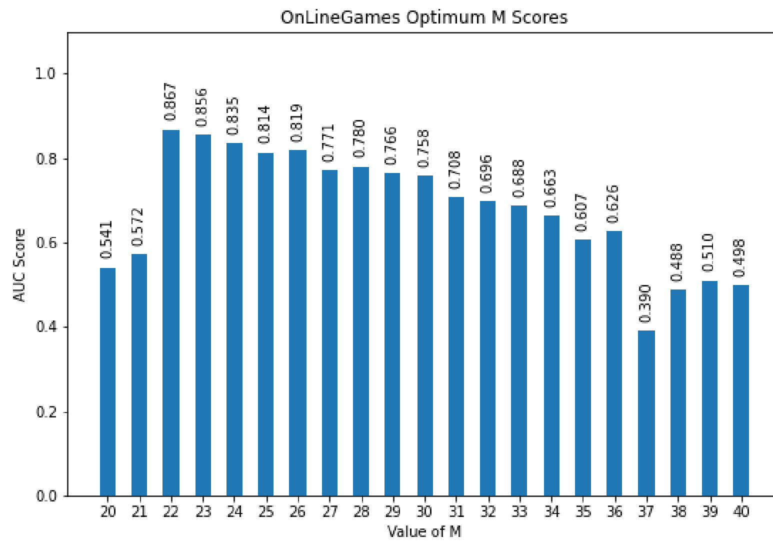

Figure A.25: AUC scores for different M values for OnLineGames.

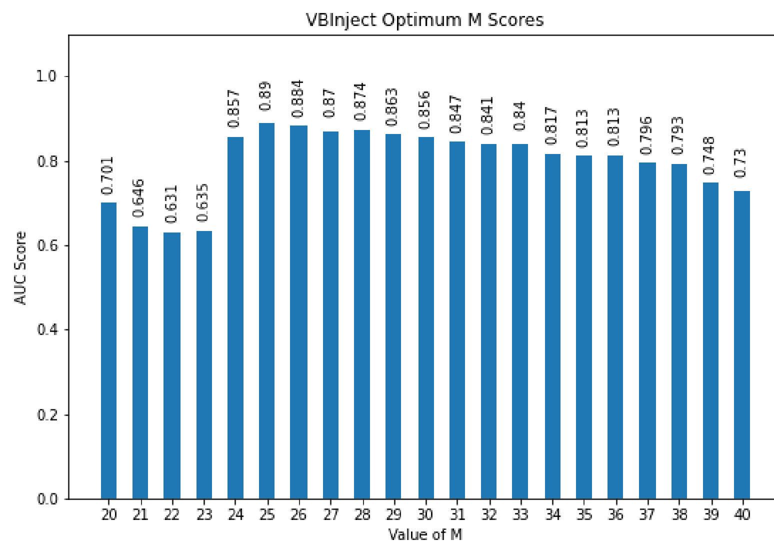

Figure A.26: AUC scores for different M values for VBInject.

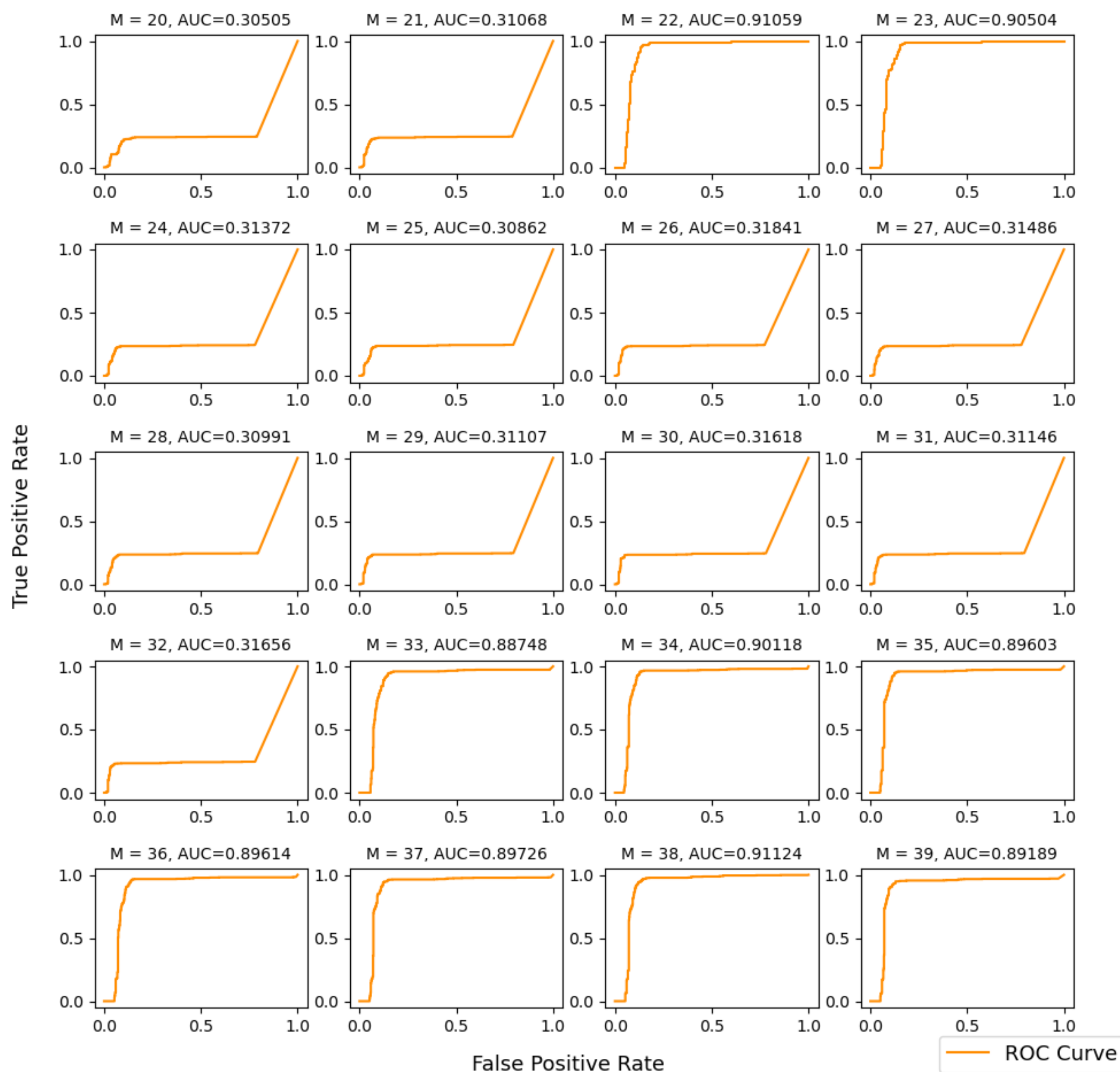

Figure A.27: ROC curves for different M values for WinWebSec.

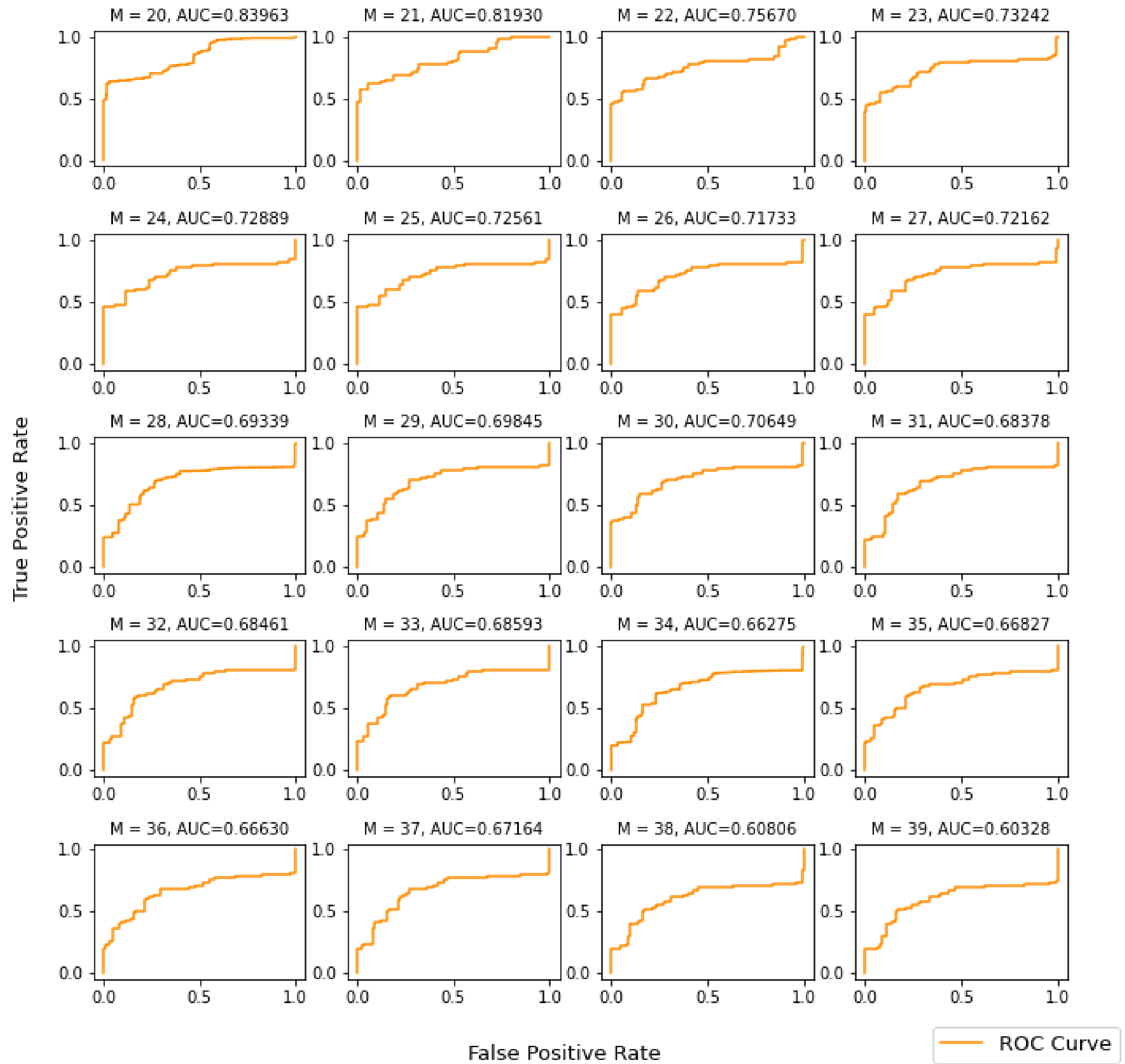

Figure A.28: ROC curves for different M values for Zbot.

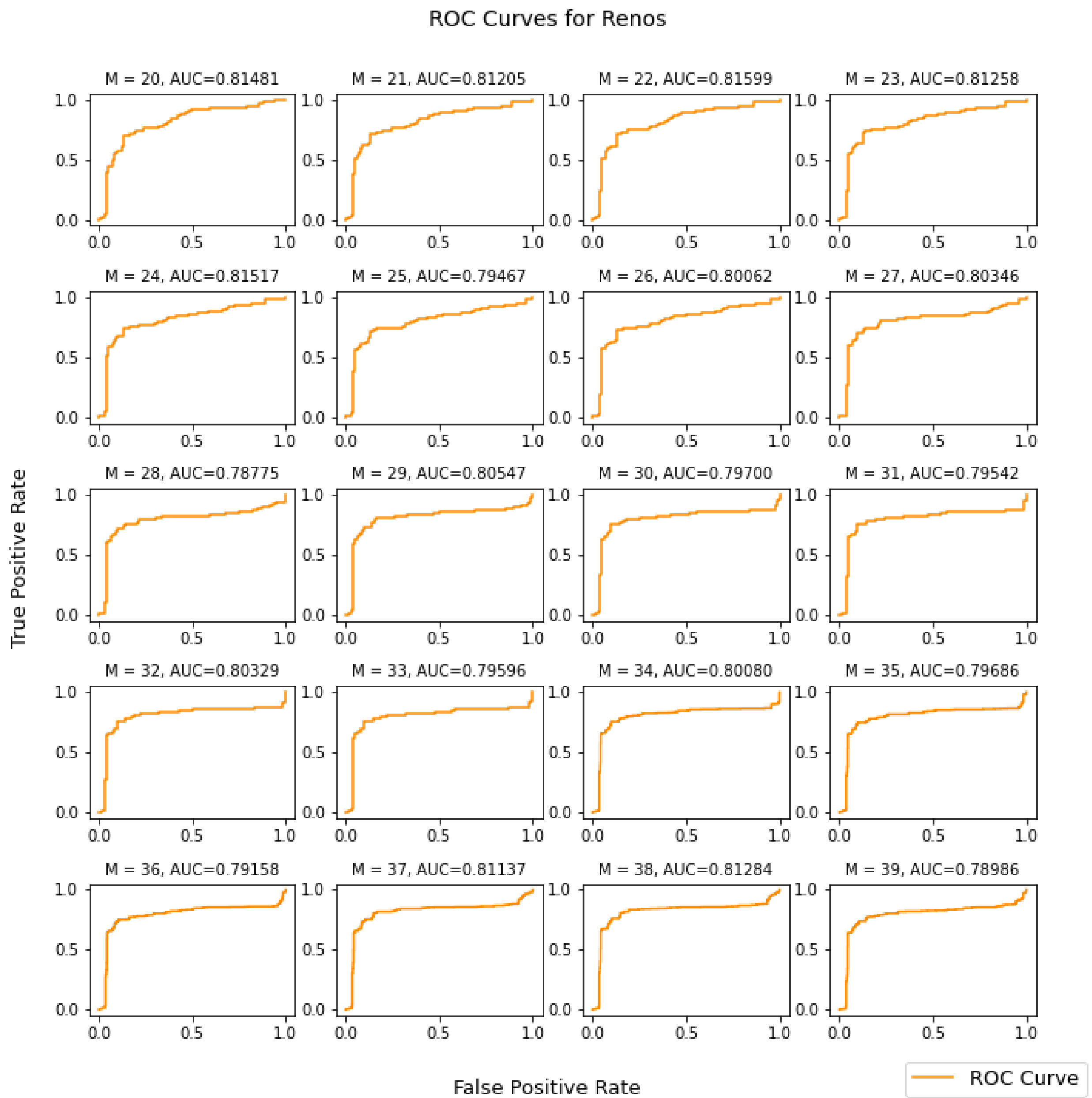

Figure A.29: ROC curves for different M values for Renos.

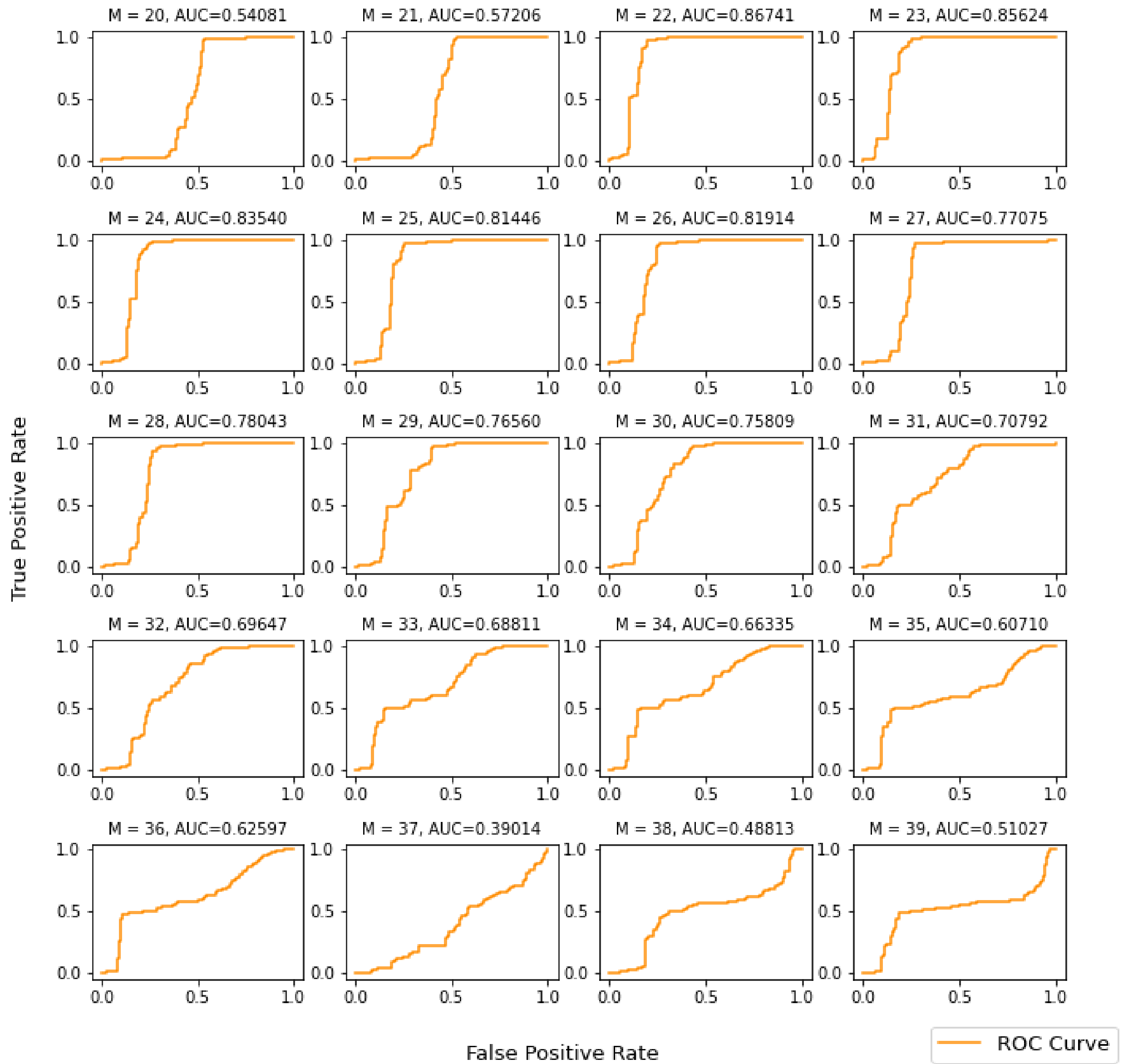

Figure A.30: ROC curves for different M values for OnLineGames.

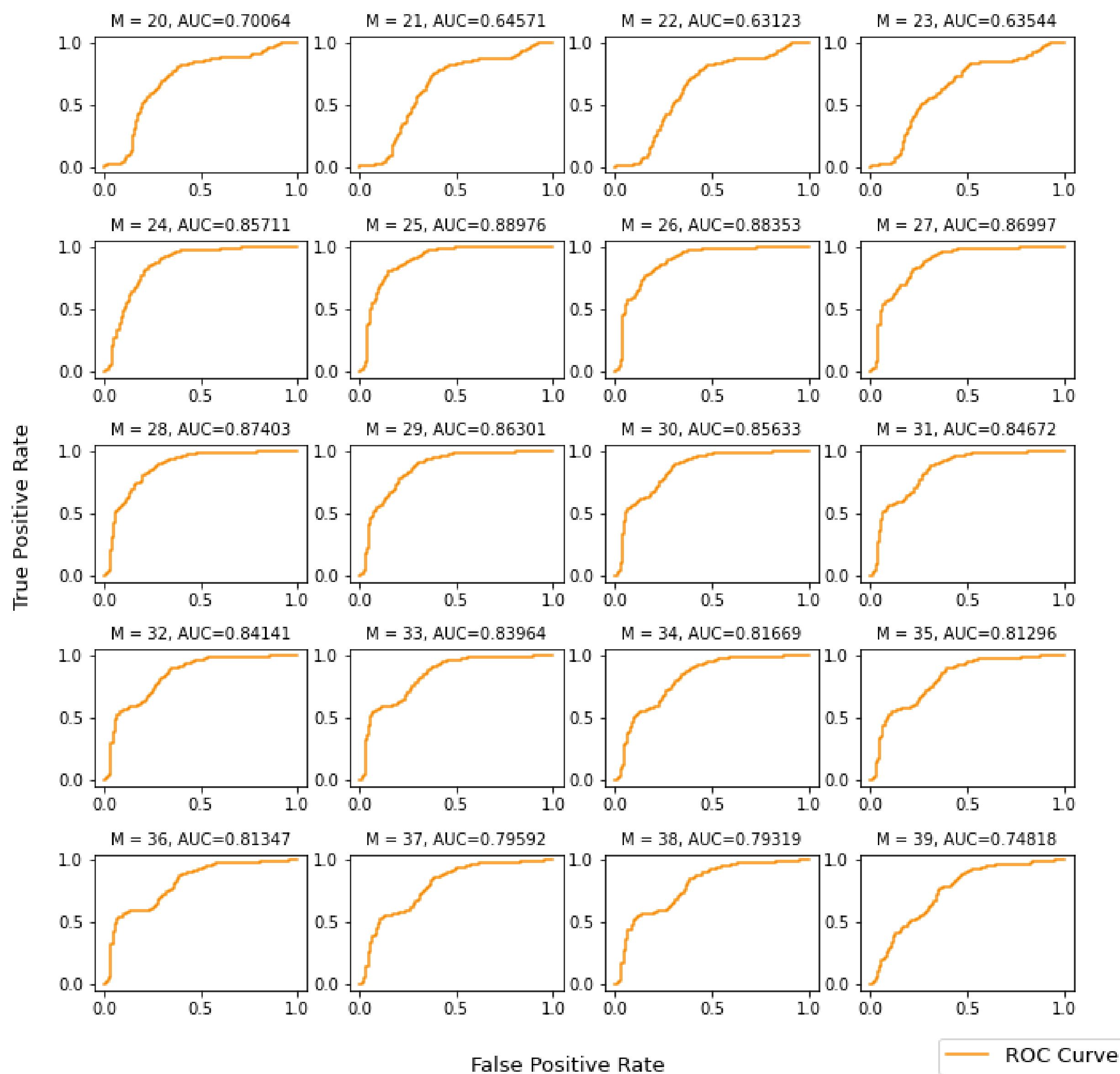

Figure A.31: ROC curves for different M values for VBInject.

APPENDIX B

Loss curves for different GAN architectures.

B.1 GAN Loss Curves

The loss curves for all 5 families are given here. We can see that the loss values don't necessarily correspond to the generative model's performance. For WinWebSec and OnLineGames the training is unstable as we see the loss values oscillating. But the results given in section 5.2 show that the results for all 5 families are almost the same with sub par quality of fake samples.

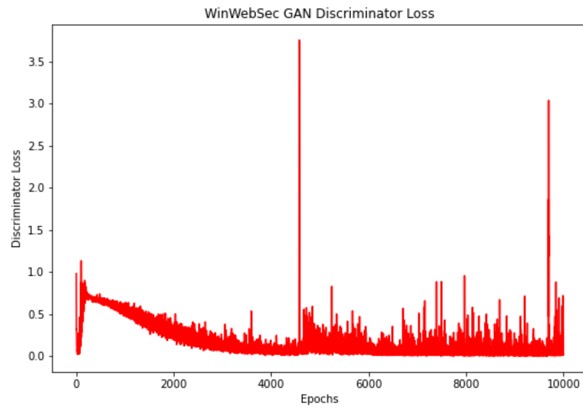

Figure B.32: WWS GAN discriminator loss.

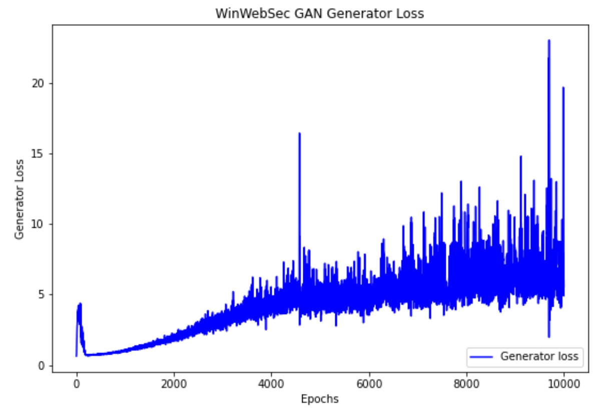

Figure B.33: WWS GAN generator loss.

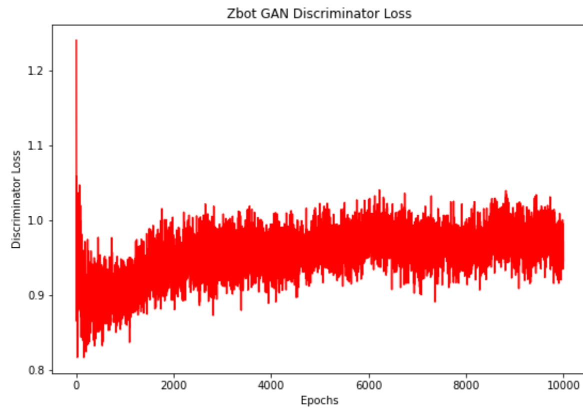

Figure B.34: Zbot GAN discriminator loss.

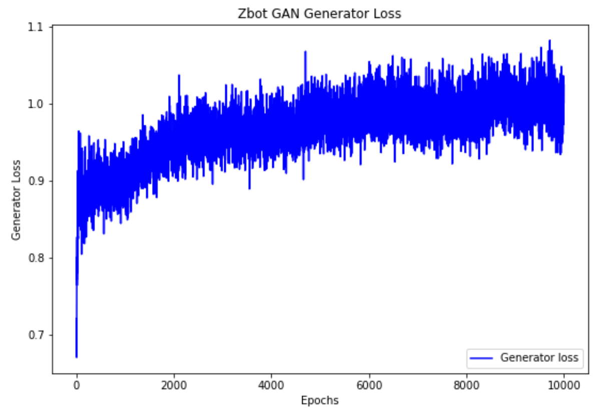

Figure B.35: Zbot GAN generator loss.

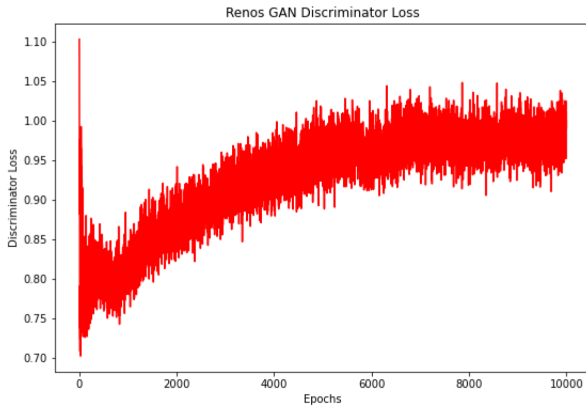

Figure B.36: Renos GAN discriminator loss.

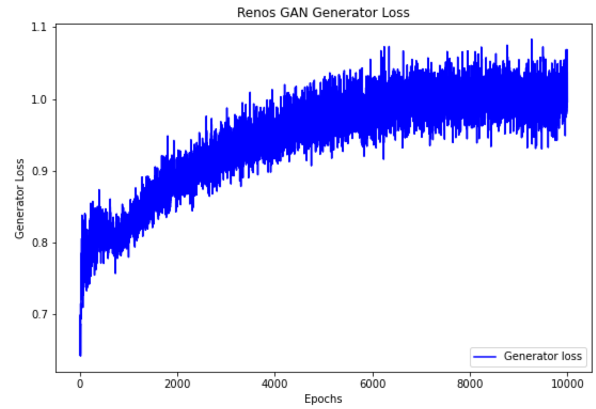

Figure B.37: Renos GAN generator loss.

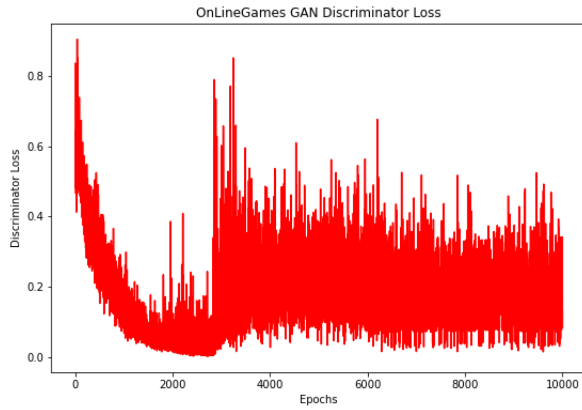

Figure B.38: OLG GAN discriminator loss.

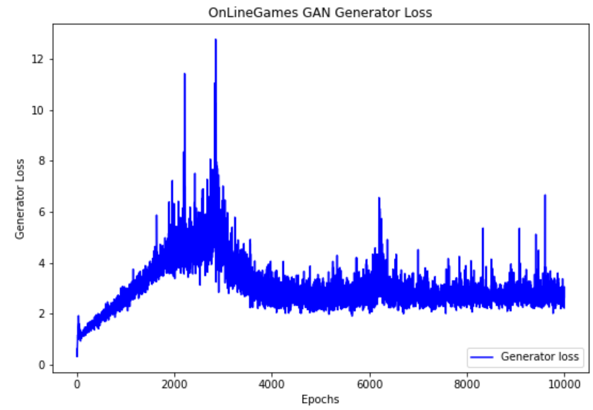

Figure B.39: OLG GAN generator loss.

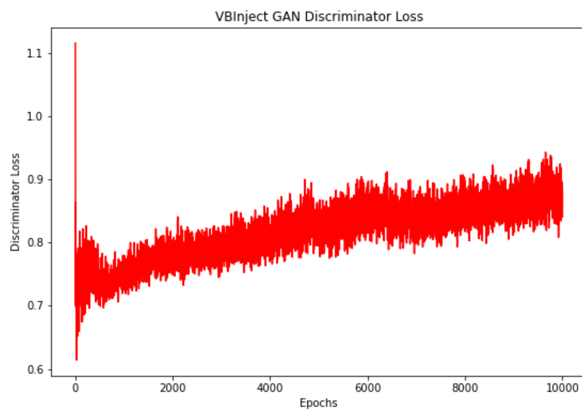

Figure B.40: VBInject GAN discriminator loss.

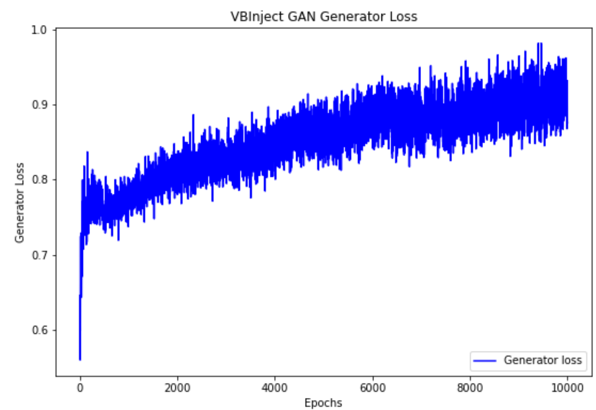

Figure B.41: VBInject GAN generator loss.

B.2 WGAN Loss Curves

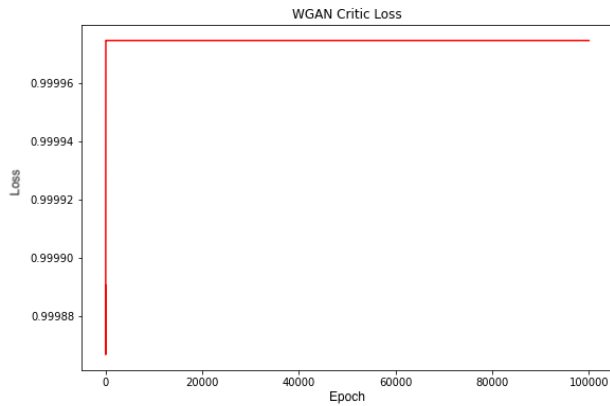

Figure B.42: Zbot WGAN critic loss.

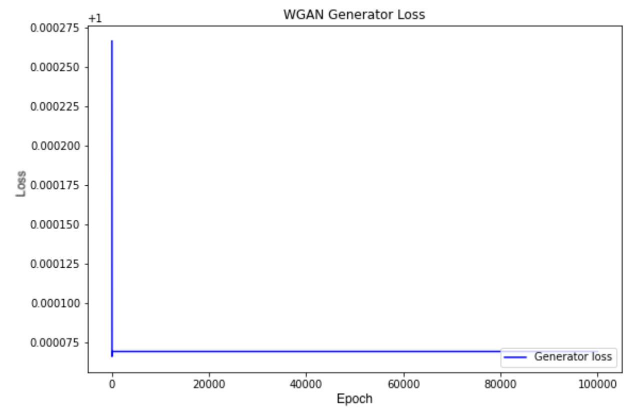

Figure B.43: Zbot WGAN generator loss.

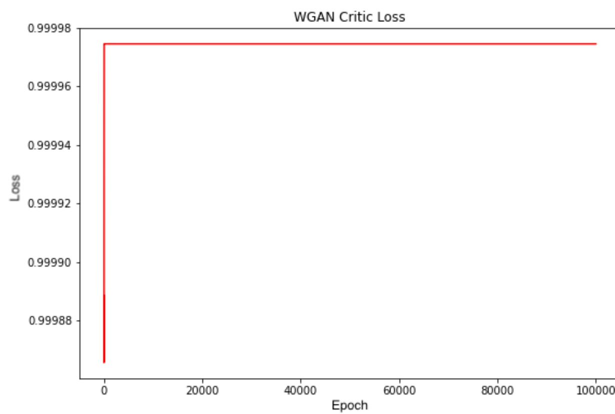

Figure B.44: Renos WGAN critic loss.

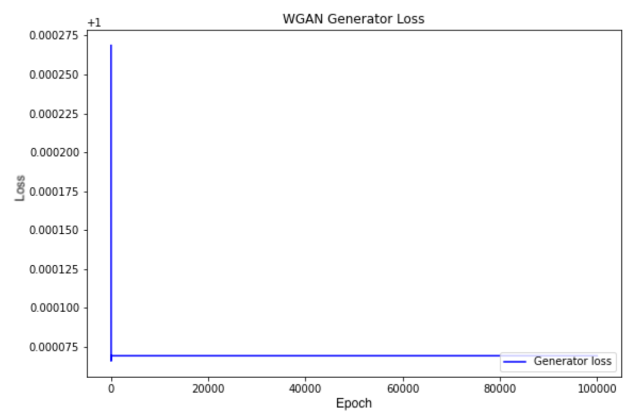

Figure B.45: Renos WGAN generator loss.

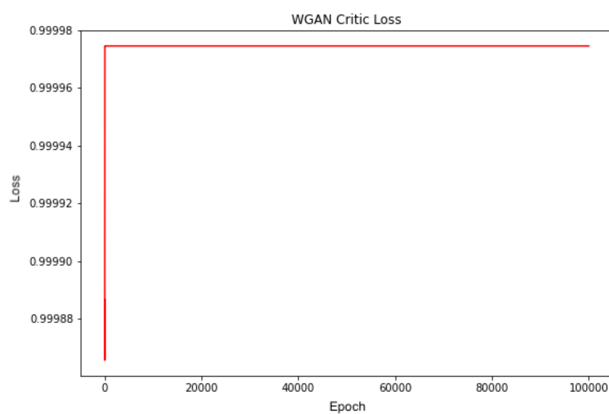

Figure B.46: OLG WGAN critic loss.

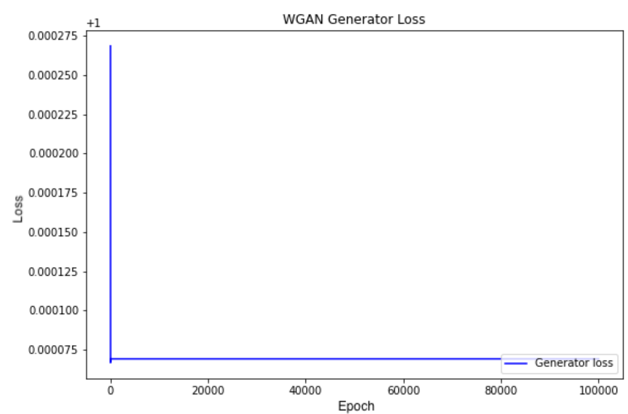

Figure B.47: OLG WGAN generator loss.

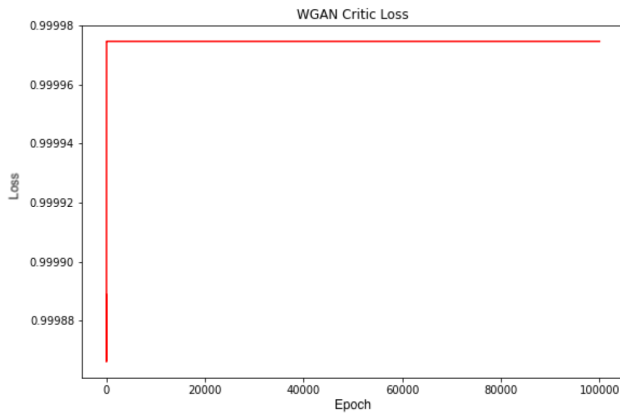

Figure B.48: VBInject WGAN critic loss.

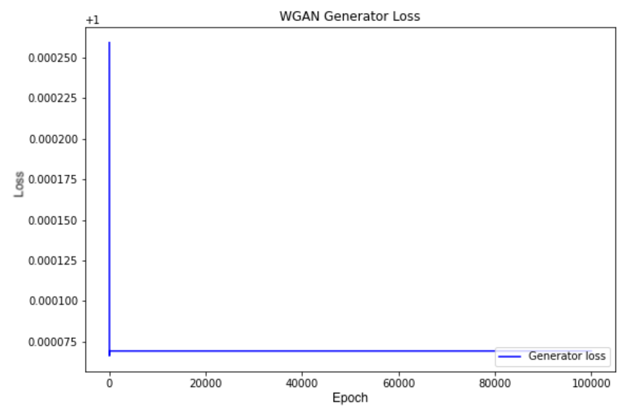

Figure B.49: VBInject WGAN generator loss.

B.3 WGAN with Gradient Penalty Loss Curves

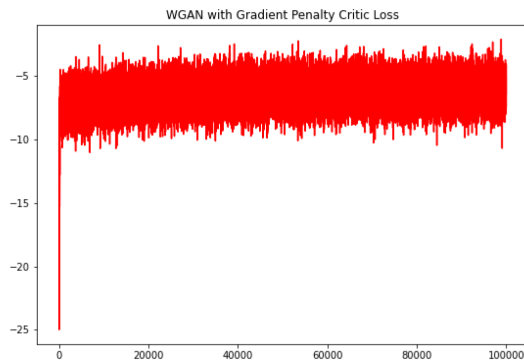

Figure B.50: Zbot WGAN-GP critic loss.

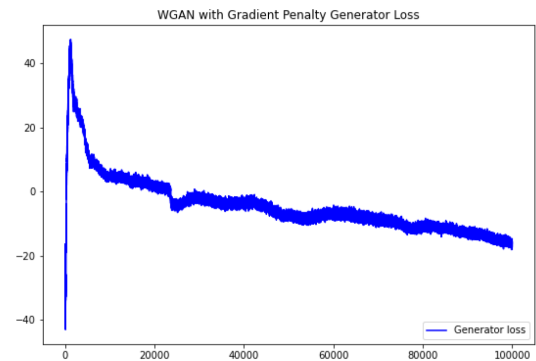

Figure B.51: Zbot WGAN-GP generator loss.

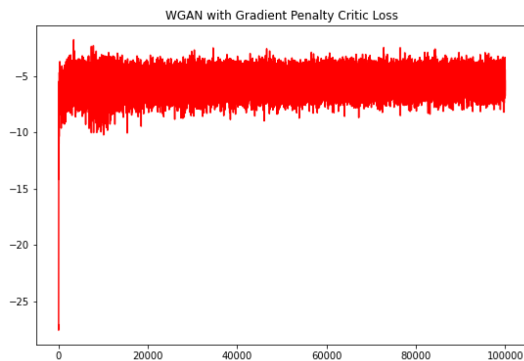

Figure B.52: Renos WGAN-GP critic loss.

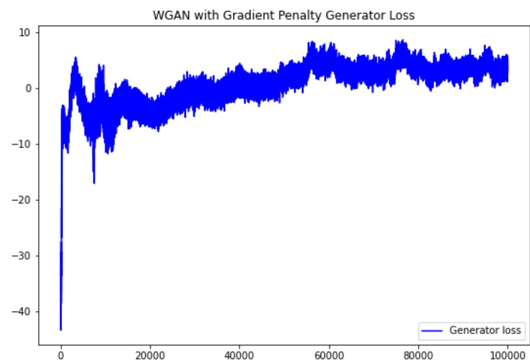

Figure B.53: Renos WGAN-GP generator loss.

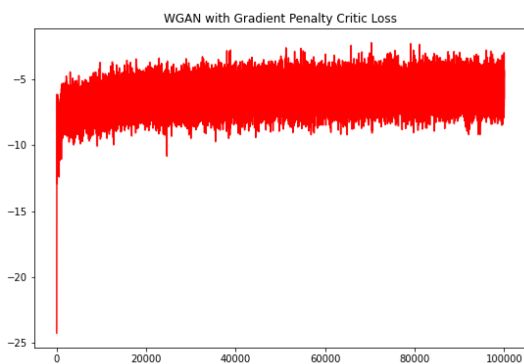

Figure B.54: OLG WGAN-GP critic loss.

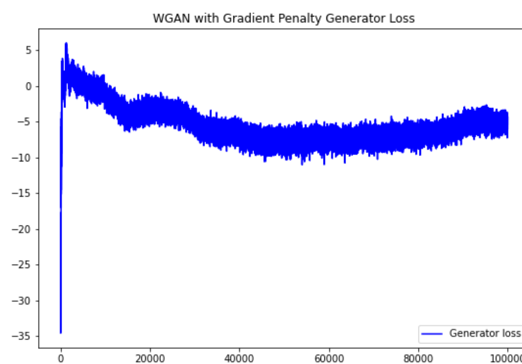

Figure B.55: OLG WGAN-GP generator loss.

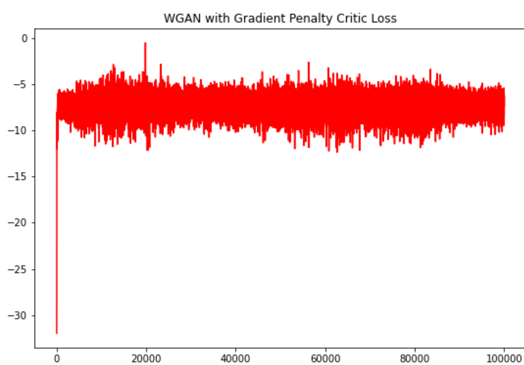

Figure B.56: VBInject WGAN-GP critic loss.

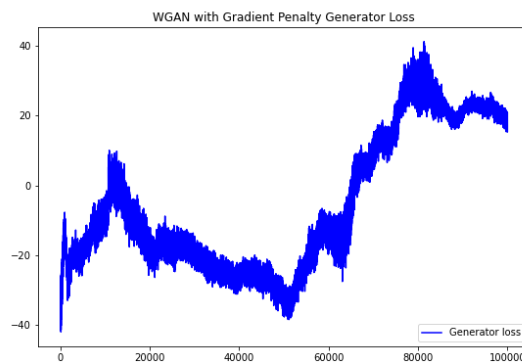

Figure B.57: VBInject WGAN-GP generator loss.

APPENDIX C

Code snippets

Code snippets for Wasserstein distance and gradient penalty calculation are given below.

C.1 Wasserstein Loss

Using -1 as ground truth labels for fake images and 1 as ground truth labels for real images, the Wassersetin loss/distance is calculated as follows:

```
def wasserstein_distance(y_true, y_pred):
    return tensorflow.keras.backend.mean(y_true * y_pred)
```

C.2 Gradient Penalty

The gradient penalty is implemented as follows:

```
import tensorflow.keras.backend as kb


def grad_penalty(y_true, y_pred, interpolated_samples):
    '''
    L2 norm or Eucledian norm calculation:
    1. Square the gradients
    2. L2 norm = Sum over rows and take square root
    3. Gradient penalty = lambda * (l2_norm - 1)^2
    '''
    gradients = kb.gradients(y_pred, interpolated_samples)[0]
    grad_squared = kb.square(gradients)
    grad_norm = kb.sqrt(kb.sum(grad_squared, axis=np.arange(1, len(
                                            grad_squared.shape))))
    grad_penalty = kb.square(1 - grad_norm)
    return kb.mean(grad_penalty)
```

We define lambda in the computational graph for the critic model when defining the weights for each loss. For Wasserstein loss we assign a weight of 1 and for gradient

penalty loss we assign a weight of 10 which is lambda. This is a hyperparameter and we experimented with different values: $\{5, 8, 10, 12, 15\}$