Ch4os: Discretized Generative Adversarial Network for Functionality-preserving Evasive Modification on Malware

Christopher Molloy, Furkan Alaca, and Steven H. H. Ding

Queen's University, Kingston ON K7L 3N6, Canada
`{chris.molloy,furkan.alaca,steven.ding}@queensu.ca`

Abstract. Rapid advancements in Artificial Intelligence (AI) have led to applications in varying domains. Due to the exponential growth of cyberspace in recent years, the domain of cybersecurity has seen substantial integrations of AI to aid in handling large amounts of data. The discipline of malware analysis within cybersecurity has leveraged AI to develop advanced analysis techniques. Within malware analysis, AI has been applied to both malware detection and evasive malware generation. Adversarial Learning on Malware (ALM) is the study of evasive modifications that is focused on AI-based detection tools. Most of the existing ALM evasive modification methods produce samples that are not valid executables. Solutions that produce effective valid executables are limited to injecting random code from a finite set of benign samples. Instead of using known code, we aim to optimize the injected bytes to increase evasion probability through adversarial learning. We propose Ch4os, a malware modification system trained in a Generative Adversarial Network setting. We introduce the Valid Machine Code Execution (VaME) activation function, guaranteeing functionality of modified malware samples while preserving differentiability of the learning process. As well, to address the challenge of learning efficiency and stability, we introduce the Binary Copier Pre-training (BCP) method. We conduct experiments on a dataset of chronologically separated malware for a simulated real-world detection scenario and show Ch4os can generate 152% more evasive samples compared to the state-of-the-art.

Keywords: Machine Learning · Adversarial Learning · Malware Analysis.

1 Introduction

Due to notable advancements, Artificial Intelligence (AI) has garnered widespread attention and adoption worldwide. One area that has seen a substantial integration of AI is cybersecurity. With the current number of internet-connected devices sharing data with one another, human-based analysis for malware detection is now impractical. Notably, the production of novel malware has reached unprecedented levels in recent years. According to the AVTest institute, there have been more than 79 million new unique malware samples indexed in 2022 that attack Microsoft systems[1]. At this scale, it is impracticable to assume human investigation can suffice. The conventional signature-based method is also unable to cope with the dynamic and versatile characteristics of malware variants. To augment human investigation at scale, multiple AI approaches have been proposed for different challenges within cybersecurity.

One research area within the cybersecurity space is attacking AI-based anti-malware tools. Given some malware and an AI-based anti-malware tool, attackers modify incoming samples or feature vectors of malware with the intent of having the sample classified as benign. This area of research, as well as the practice of creating AI methods that are more challenging to evade, is known as Adversarial Learning on Malware (ALM). Multi-agent systems have been used for ALM in recent works to better simulate real-world defender-attacker environments [13].

Generative Adversarial Networks (GANs) are multi-agent neural networks designed to create approximations to some data space with higher resolution than conventional generative networks. GANs have been applied to the cybersecurity space in both malware generation and detection [6, 8, 11, 14, 15]. [6] used a GAN to generate adversarial feature vectors for malware samples against a black box system. A drawback of this work is the GAN was only trained to produce a modified feature vector of the sample. Reinforcement Learning (RL) is also used for adversarial sample generation. Current RL functional adversarial sample generation methods train against AI-based anti-malware engines with a finite set of actions for malware sample modification [1, 13]. Molloy *et al.* show that a two-party game environment improves the evasiveness of generated samples, as well as the detection ability of an AI-based anti-malware tool [13]. Molloy *et al.* [13] found that the most effective method for producing functional adversarial samples was appending benign machine code to the malicious executable. One drawback of this method is that it relies on a repository of pre-written benign machine code that can be sampled from to modify malicious executable samples.

We propose a different route, Ch4os, a functionality-preserving adversarial sample generation tool. Ch4os generates and directly optimizes functionality-preserving machine code injected into a malicious file's overlay, augmenting its evasiveness against detection tools. Ch4os is trained in a GAN architecture against a pre-trained state-of-the-art Deep Learning based anti-malware tool for adversarial machine code optimization. Unlike the previous methods that have proposed GAN architectures for malware generation, we introduce the VaME activation function, a discretized function which maps continuous values to machine code. To address the learning efficiency and stability challenge of adversarial machine code optimization from conventionally initialized network weights, we propose a pre-training method that prepares the network for adversarial byte generation by optimizing to learn the data distribution of benign machine code. Our system has shown successful evasive sample generation against a state-of-

[1] https://portal.av-atlas.org/

the-art anti-malware engine trained on a research standard dataset. The main contributions of this paper are as follows:

1. We propose the first GAN for functional evasive malware modification with the novel VaME activation function, enabling differentiability of the complete learning process.
2. To address the challenge of learning efficiency and stability, we propose the Benign Copier Pre-training (BCP) method, which accelerates the initial convergence problem.
3. We compare our training method against contemporary solutions using a real-world dataset of malware samples and show our method can produce 152% more evasive samples than the state-of-the-art.

2 Related Work

GANs are a neural network architecture proposed by Goodfellow *et al.* GANs use two networks, a generator and a discriminator, to train off one another for better results. GANs have been designed against adversarial attacks, attacks in which data samples are modified to be misclassified by the network [10]. This idea has extended to cybersecurity in the domains of ALM and zero-day malware detection [8,11,14,15]. Kim *et al.* proposed the tGAN (transferred GAN) for classifying malware [8]. The proposed system used an autoencoder to generate images of malware from rescaled assembly code visualizations. The discriminator is then transferred to a malware detector with a family classification accuracy of 96.39%. Moti *et al.* proposed a GAN that simulated potential zero-day malware Binary header information [14]. The simulated headers were incorporated into a malware dataset and trained through multiple classifiers. Training the classifier with the generated data showed an improvement in malware classification accuracy from 97.12% to 98.14%. Lu *et al.* proposed a method of increasing a malware dataset size by simulating malware samples with a GAN [11]. Lu *et al.* showed that training a network with the simulated data could raise the classification accuracy by 6%.

Hu *et al.* proposed MalGAN, a GAN architecture for generating adversarial samples against a black box malware detector [6]. The system proposed by Hu *et al.* was able to achieve a true positive percent of zero in some test datasets. The generator in this work modifies each malware feature vector and does not generate a functional piece of obfuscated malware. Kargaad *et al.* built on top of this system by proposing a malware detection network that is trained on the resulting data from MalGAN [7]. Nazari *et al.* proposed a CGAN-based upsampling method to produce software data samples for balancing malware datasets [16]. Trehan *et al.* showed that multiple different GAN architectures can be used to generate malware mnemonic opcode sequences for model training [20].

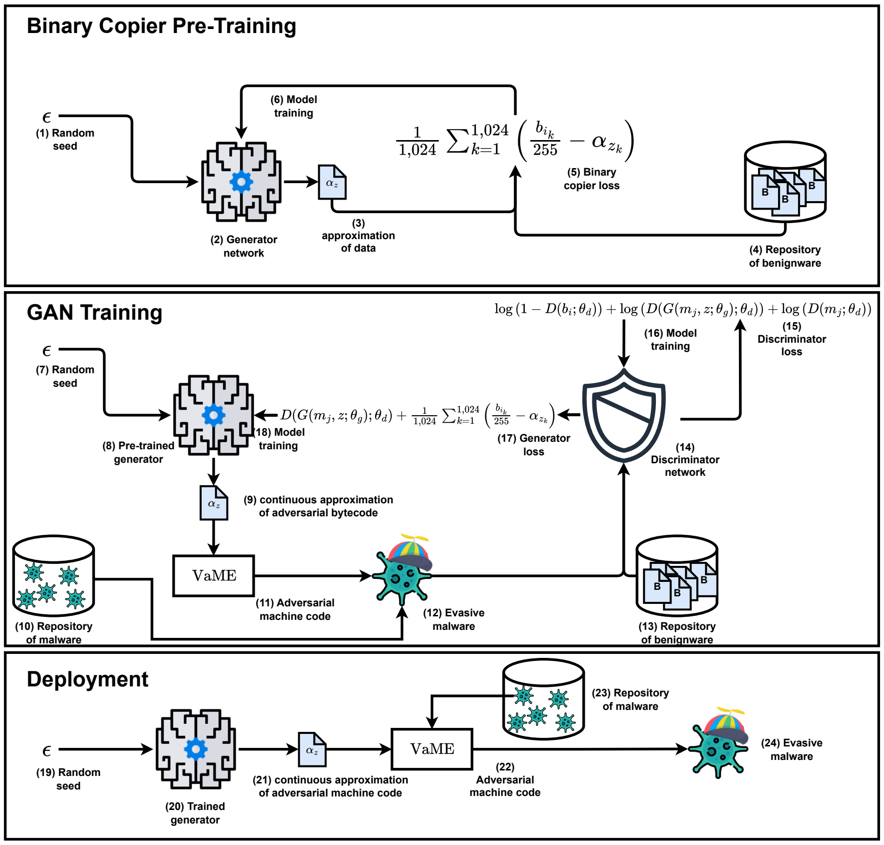
Binary Copier Pre-Training
ϵ
(1) Random seed
(2) Generator network
α_z
(3) approximation of data
(6) Model training
$\frac{1}{1{,}024}\sum_{k=1}^{1{,}024}\left(\frac{b_{i_k}}{255}-\alpha_{z_k}\right)$
(5) Binary copier loss
B
(4) Repository of benignware
GAN Training
$\log\left(1-D(b_i;\theta_d)\right)+\log\left(D(G(m_j,z;\theta_g);\theta_d)\right)+\log\left(D(m_j;\theta_d)\right)$
(16) Model training
(15) Discriminator loss
ϵ
(7) Random seed
(8) Pre-trained generator
$D(G(m_j,z;\theta_g);\theta_d)+\frac{1}{1{,}024}\sum_{k=1}^{1{,}024}\left(\frac{b_{i_k}}{255}-\alpha_{z_k}\right)$
(18) Model training
(17) Generator loss
(14) Discriminator network
α_z
(9) continuous approximation of adversarial bytecode
VaME
(11) Adversarial machine code
(12) Evasive malware
(10) Repository of malware
(13) Repository of benignware
Deployment
ϵ
(19) Random seed
(20) Trained generator
α_z
(21) continuous approximation of adversarial machine code
VaME
(22) Adversarial machine code
(23) Repository of malware
(24) Evasive malware

Fig. 1: An overview of the training and deployment process of Ch4os. Nodes represent an action, and edges represent the next action that requires such action. (1) Generate a random vector from prior distribution. (2) Pass vector as input to generator network. (3) Generate an approximation of data b. (4) Retrieve a sample of benignware. (5) Generate benign copier loss from benign sample and approximate data. (6) Train generator on loss. (7) Generate a random vector from prior distribution. (8) Pass random vector into generator network. (9) Generate continuous approximation of adversarial machine code. (10) Retrieve a sample of malware. (11) Generate adversarial machine code. (12) Generate evasive malware. (13) Retrieve a sample of benignware. (14) Evaluate samples with discriminator network. (15) Generate discriminator loss. (16) Train the discriminator network on discriminator loss. (17) Calculate generator loss. (18) Train generator network on generator loss. (19) Generate a random vector from prior distribution. (20) Pass random vector into trained generator network. (21) Generate continuous approximation of adversarial machine code. (22) Generate adversarial machine code. (23) Retrieve a sample of malware. (24) Generate evasive malware.

3 Methodology

The Ch4os system is a deep generative neural network for adversarial machine code generation. An overview of the Ch4os system can be seen in Fig. 1. Given a random vector input, a kilobyte of adversarial machine code is generated to be repeatedly appended to the end of the malware machine code. The resulting malware sample contains machine code that has been optimized to evade static analysis-based anti-malware engines. The adversarial machine code is generated from the generator network and goes through two steps of training optimization. The first step of training is optimizing the generator network to create benign machine code from a random vector. The second step of training is optimizing the generator to create evasive bytes in a multi-agent environment against a pre-trained malware detection network.

3.1 Data

We define data b as benign Microsoft PE machine code. The PE file format is the most used executable file format for Microsoft Windows operating systems. PE Executables are written in machine code and are comprised of various headers and sections [17]. Headers contain important information such as the intended machine type, the number of sections, and the number of symbols. PE sections contain code blocks for PE-specific execution. Also, a PE Executable file may contain an overlay, optional code machine code added to the end of the file that is not mapped to anywhere in memory [5].

Our data m is malicious Microsoft PE machine code. Malware is defined as any piece of code that has been designed to cause harm or subvert a computer's intended function when executed [12]. Malicious PE files have the same format as benign files but are designed to perform different actions on a user's computer than benign code.

3.2 Adversarial Networks

We build on the work of Goodfellow *et al.* for defining our adversarial network [4]. In this work, we optimize the distribution p_g of the generator network G over data b. We define data m as input to the generator. We also define a prior noise variable z from distribution p_z as input to the generator. We define $G(m, z; \theta_g)$ as a mapping from the space of data m and the noise variable z to the b data space. G is a deep neural network with parameters θ_g. We define $D(\cdot; \theta_d)$ that outputs a single scalar between 0 and 1. $D(\cdot; \theta_d)$ represents the probability that the input data came from p_g rather than b. This can be interpreted as if the input data is untrue to the data b. We train D to maximize correct assignment of probability to both training samples from b and data generated from G. While D is trained, we concurrently train G to minimize the correct assignment from D for data generated from G. It follows that G and D play a two-player minimax game with the value function $V(G, D)$:

$$\min_G \max_D V\left(G, D\right) = \mathbb{E}_{b \sim p_{data}(b)}\left[1 - D\left(b\right)\right] + \mathbb{E}_{z \sim p_z(z)}\left[D\left(G\left(z\right)\right)\right] \tag{1}$$

3.3 Adversarial Attack

The target of the Ch4os system is deep learning models trained in malware detection [18]. Given an input sample executable, some preprocessing may be required for feature set generation, and some models perform inference directly from the software binary code [18]. These networks have a sigmoid activation head for malware prediction. Malicious software has a true positive value of 1, and benign software has a true positive value of 0. Based on experimentation results, a classification threshold $t \in 0 \leq \mathbb{R} \leq 1$ is chosen for the network to optimize the area under the ROC curve [19]. The Ch4os system aims to modify malware binaries at the machine code level to reduce the classification prediction below the target detection network's threshold without implicit knowledge of the threshold. This is done by adding machine code generated by $G(m, z; \theta_g)$ to the overlay of the attacking malware sample. As is the case with benign PE executables, PE malware executables can also have machine code added to the overlay that does not disrupt intended functionality. The adversarial attack is conducted as follows:

1. A sequence of bytes fitted to data b is generated by the generator network $G(m, z; \theta_g)$.
2. The sequence of generated bytes is concatenated to the end of a malware sample executable.
3. The concatenation is repeated with the same machine code until the modified sample is over the maximum input size of the target anti-malware engine.

It is intended that the adversarial machine code at the end of the malware sample reduces the malicious probability under the detection threshold against the target anti-malware engine.

Table 1: Structure of generator network.

Layer	Size-in	Size-out	Kernel	Params
random sample		1		0
dense1a	1	16		32
dense1b	16	256		4352
conv1a	256	256	1	65,792
lnorma	256	256		512
conv1b	256	512	1	131,584
lnormb	512	512		1024
conv1c	512	1024	1	525,312
lnormc	1024	1024		2048
flatten	1024	1024		0
Total				730,656

3.4 Generator

A challenge for designing the generator network $G(m, z; \theta_g)$ is mapping output from a continuous differentiable neural network to the discrete space of machine code. Deep Learning optimization and training require a continuous and differentiable network for backpropagation. A challenge in this context arises from the limitation that machine code is exclusively comprised of integers, but we require the real-valued output of a continuous activation head for model optimization. Due to this, we propose the Valid Machine Code Execution (VaME) activation function. We define VaME : $0 \leq \mathbb{R}^n \leq 1 \Rightarrow 0 \leq \mathbb{Z}^n \leq 255$, where $n \in \mathbb{N}$ is arbitrary, as

$$\text{VaME}(x) = \lfloor 255x \rfloor . \tag{2}$$

The VaME activation head is selectively employed in the generation of a functional adversarial sample, while its utilization is absent in the computation of the generator network's loss. This utilization technique poses no challenge to model efficacy, as the VaME activation function has no trainable parameters.

We will now describe the generator network $G(m, z; \theta_g)$. The input to $G(m, z; \theta_g)$ is some malware machine code m and a randomly generated noise z. We sample z from the standard normal distribution $p_z(z) \sim N(0, 1)$. The random noise is upsampled by two fully connected neuron layers to a vector of length 256. The vector is then upsampled further through three deep convolution blocks of one-dimensional convolution and layer normalization. The result of the final convolution block is then flattened to a vector, α_z, of length 1,024. The vector α_z is then passed through a sigmoid activation function

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{3}$$

The sigmoid activation function transforms α_z such that $0 \leq \alpha_z \leq 1 \in \mathbb{R}^{1,024}$.

The vector α_z is used for model loss calculation as well as adversarial sample generation. We will now explain adversarial sample generation. Given the input dimension of the discriminator function, q, we concatenate the executable code generated by α_z, to the input malware sample m with starting length r, until the length of the malware sample m is larger q. We denote this vector as

$$\beta = m \,\|_{i=0}^{\left\lceil \frac{q-r}{1,024} \right\rceil} \text{VaME}(\alpha_z). \tag{4}$$

We then remove the trailing bytes until the length of β is the correct input length for the discriminator. The resulting machine code is the output of the generator network, thus

$$G(m, z; \theta_g) = (\beta_i)_{1 \leq i \leq q} . \tag{5}$$

The network architecture can be seen in Table 1.

Table 2: Structure of discriminator network.

Layer	Size-in	Size-out	Kernel	Params
embedding	1,048,576	1,048,576×8		2,056
conv1a	1,048,576×8	2097×128	500	512,128
conv1b	1,048,576×8	2097×128	500	512,128
matrix multiply	(2097×128,2097×128)	2097×128		0
mpool	2,097×128	128		0
dense1a	128	128		16,512
dense1b	128	1		129
Total				1,042,953

3.5 Discriminator

We require a pre-trained anti-malware network to act as the discriminator of the Ch4os system. For the discriminator, we train with the modified MalConv detection network trained on the EMBER dataset [2,18]. This pre-trained network was tested in [2], resulting in an ROC of 0.998 on the testing set.

MalConv is a gated deep convolution network for malware detection proposed by Raff *et al.* [18]. Table 2 shows the architecture of the discriminator model. The input to the discriminator is a bytestring of the first megabyte of the executable's machine code. The bytestring is a vector $b \in \mathbb{Z}^{1,048,576}, 0 \leq b \leq 256$. There are a total of 257 possible values for each byte value due to the use of 256 as a padding character if the software sample is less than one megabyte in length. The bytestring is first embedded into a space that represents each byte in the byte string as a vector of length eight. This space is represented by $D^{|257| \times 8}$ due to the vocabulary of bytes in the executable being 257.

The input to the gated convolution unit is the embedded bytestring. The gated convolution unit takes the input of the embedding and passes it through two parallel one-dimensional convolution units. One of the convolution units acts as an attention layer, and the other a filter layer [18]. Both convolution layers have the same hyper-parameters of a filter size of 128, a kernel size of 500, and a stride of 500, but it is important to note there is a difference in layer activation. One layer uses the rectified linear unit (ReLU) activation function, and the other layer uses a sigmoid activation. The output of the two convolution layers are multiplied with one another to yield the gated unit result. The sigmoid activation is used on one of the convolution layers to create an attention scalar, which is multiplied by the second layer. The output matrix of the attention is max pooled on the embedded dimension. The output of the pooling is sent through two fully connected neuron layers to reduce with output dimensions of 128 and one, respectively. The final neuron has a sigmoid activation function, making the output of the MalConv network a probability. The output of the MalConv network is the probability that the input is malicious. The network architecture can be seen in Table 2.

The most significant difference between the MalConv network originally proposed by Raff *et al.* and the modified version used in [2] and this work is the input size. The input size to the original MalConv network was the first two megabytes of machine code from the software executable, whereas the input size to the modified MalConv is the first single megabyte of the software sample. This input size was chosen to accommodate the memory capacity of state-of-the-art Graphics Processing Units [2].

We use Transfer learning for the pre-trained discriminator. As previously discussed, the original training task of the MalConv network is classifying if an incoming sample is malicious. Due to software being only benign or malicious, it follows that the MalConv network is trained to classify if an incoming sample is not benign. This can be understood as assigning the probability to if the sample is not from data b. In this study, we consider non-functional software benign due to the inability of non-functional software to cause harm to a user's computer. We transfer this to the task of giving the probability that the incoming information is not from our data b but originates from $G(m, z; \theta_g)$.

3.6 GAN Loss

We will now describe the loss of the discriminator D. We optimize D to maximize the probability of correct assignment of training samples from b and data generated from G. As well we also optimize D to maximize the correct assignment of samples from m. This additional optimization procedure is implemented with the aim of ensuring discriminator D retains the capacity to assign whether a given sample originates from data b based on malicious machine code. For a single train step, we require some random vector z, some data b_i sampled from b, and some data m_j sampled from m. First, we find the probability that sample b_i is from data b. This is done by finding the log loss of assigning b_i to data b calculated as

$$L_1 = \log\left(1 - D(b_i; \theta_d)\right). \tag{6}$$

We then find the probability that our adversarial sample $G(m_j, z : \theta_g)$ is from data b. This is done by finding the log loss of assigning $G(m_j, z : \theta_g)$ to data b calculated as

$$L_2 = \log\left(D(G(m_j, z; \theta_g); \theta_d)\right). \tag{7}$$

Subsequently, we calculate the probability that m_j from data m is assigned to data b. Similarly to the above, this is done by finding the log loss of assigning m_i to data b. This is calculated as

$$L_3 = \log\left(D(m_j; \theta_d)\right). \tag{8}$$

Finally we compute the loss for the discriminator as the sum as the previously discussed losses as $L_D = L_1 + L_2 + L_3$.

We will now describe the loss of generator G. Following the same train step in calculating the loss for the discriminator D, the following is how the generator loss is calculated for the same z, b_i, and m_j. As described in the minimax game, we are optimizing the generator to minimize the correct assignment from the discriminator generated from G. To incorporate this into the generator loss, we create the reward r, where

$$r = 1 - D(G(m_j, z; \theta_g); \theta_d). \tag{9}$$

The reward is 1 if the discriminator assigns the adversarial sample to not in data b with probability 0, and the reward is 0 if the discriminator assigns the adversarial sample to not in data b with probability 1.

With the discriminator correct assignment minimization loss, we also find the mean squared error of the adversarial bytes α_z against the data b_i to further optimize the generator in mapping z to the data b. For the mean squared error calculation, we multiply all bytes in data b_i by $\frac{1}{255}$ to map the machine code to the range of α_z. This is calculated as

$$L_4 = \frac{1}{1{,}024} \sum_{k=1}^{1{,}024} \left(\frac{b_{i_k}}{255} - \alpha_{z_k} \right). \tag{10}$$

Finally, the calculation of the total loss of the generator network is performed as $L_G = 1 - r + L_4$.

In experiments, we perform GAN training with the Adam Stochastic Gradient Descent method [9] for both the generator and discriminator. Both models are trained with a learning rate of 10^{-9} for 1,000 epochs.

3.7 Benign Copier Pre-training

We use the weight initialization method proposed by Glorot and Bengio [3] as the method of conventional weight initialization. When initialized with random weights, $G(m, z; \theta_g)$ proves inept at generating an adversarial β. To prepare the generator $G(m, z; \theta_g)$ for adversarial sample generation, the network is pre-trained in a benign machine code generation task. We refer to this training process as Benign Copier Pre-training (BCP).

The input to the generator network $G(m, z; \theta_g)$ is some malware machine code m and a randomly generated noise z. As well, we require a set of benign machine code as a training set. Similar to GAN training, the noise z is sampled from the standard normal distribution. For BCP, malware machine code is not used for training, so the input is some arbitrary machine code m. We optimize the network to generate machine code that is similar to samples from data b. Given some random noise z, we compare the generated machine code $\text{VaME}(\alpha_z)$ to a vector of benign machine code of the same length from our training set. The advantage of this pre-training is that before any training in the GAN environment, our network can already generate machine code that is similar in distribution to data b. In BCP training, we optimize the generator to generate machine code that is similar to data b whereas in GAN training, we optimize the generator to generate an adversarial malware sample that is similar to data b.

3.8 Benign Copier Loss

We will now describe the loss of the BCP. In BCP, we optimize the generator $G(m, z; \theta_g)$ in generating machine code, α_z, that is similar to data b. For a single train step, we require a random noise z and some benign machine code b_i sampled from data b. First, we find the continuous output of the generator for the random noise z as α_z. We then optimize our network on the comparison of each continuous value of α_z to each value in b_i that is mapped from the machine code space in the same continuous space of α_z. This mapping is simply done by dividing each byte of b_i by 255. We compare these bytes by finding the mean squared error for each value in the two vectors α_z and b_i. We treat b_i as the true positive value and α_z as the predicted value. We calculate this loss as

$$L_{BCP} = \frac{1}{1{,}024} \sum_{k=1}^{1{,}024} \left(\frac{b_{i_k}}{255} - \alpha_{z_k} \right). \tag{11}$$

The loss L_{BCP} has the same form as L_4. The loss derived from L_{BCP} is then propagated through the network to optimize for benign machine code generation.

In experiments, we perform the BCP with the Adam Stochastic Gradient Descent method [9]. We train the model with a learning rate of 10^{-5} for $10,000$ epochs.

4 Experiments

For our experiments, we required four datasets. All samples used for training and testing were Microsoft PE Executable files. All samples used were collected from various online repositories.

The first dataset was used to conduct the BCP generator training. For BCP training, we used a dataset of 1,000 benignware samples upsampled from a set of 915 unique benign software binaries through sampling with replacement. For the BCP training, we used a train-validation split of $0.8 - 0.2$.

The second dataset used was for GAN training. For GAN training, we used a dataset of 15,000 malicious-benign pairs. Within the entire dataset, there were 15,000 unique malware samples first identified in 2021 from 173 unique families and 4,873 unique benign samples that were up-sampled to the required 15,000 pairs.

The third dataset was required for testing the evasive ability of the Ch4os system. This dataset was referred to as the Detection Testing Set, and was used to determine an optimal detection threshold of the target detection network. This dataset was comprised of 6,800 malware and benignware samples with 3,413 malicious and 3,387 benign. All malware samples in the Detection Testing Set were first identified in 2021.

The fourth dataset was a holdout set of malware for novel generation. The holdout set was comprised of 6,000 unique malware samples first identified in 2022 from 63 unique families.

We conducted two experiments to demonstrate the efficacy of the Ch4os system. The first experiment validated the BCP method. We performed GAN training with a generator that had weights initialized by the Glorot and Bengio method [3] as well as a generator that was first optimized with the BCP method. As described in section 3.6, for GAN training, both models trained for 1,000 epochs on the GAN training dataset. Prior to GAN training, the generator that was optimized with the BCP method was trained for 10,000 epochs on the BCP training dataset. We refer to the model that was pre-trained with BCP as Ch4os and the model that was not pre-trained as GAN-NO-BCP.

For our first experiment, we used four metrics from the training and validation sets to measure the results of training the GAN-NO-BCP model compared to the Ch4os model. The first metric used was the generator loss. This was the loss of the generator network on the final epoch of training. The optimal generator loss value was 0. The second metric was the discriminator loss. This was the loss of the discriminator function on the final epoch of training. The higher the discriminator loss, the better the performance of the generator. The third metric used was the generator reward. The optimal value for the generator reward was 1. The fourth metric used was the detection accuracy of the generator given based on a threshold of 0.5. The closer the training reward was to 0 indicated a higher performance in the generator.

Table 3: Results of training models with different weight initialization in the GAN architecture. We refer to training loss as T-Loss and validation loss as V-Loss. We refer to training reward as T-Reward and validation reward as V-Reward. We refer to training accuracy as T-Accuracy and validation accuracy as V-Accuracy.

Model	**Gen. T-Loss**	**Disc. T-Loss**	**Gen. T-Reward**	**Disc. T-Accuracy**
GAN-No-BCP	0.6611	4.2006	0.5875	0.7015
Ch4os	**0.1332**	**6.8089**	**0.8916**	**0.6016**
Model	**Gen. V-Loss**	**Disc. V-Loss**	**Gen. V-Reward**	**Disc. V-Accuracy**
GAN-No-BCP	0.6544	4.2718	0.594	0.6966
Ch4os	**0.1158**	**7.0781**	**0.909**	**0.5956**

The second experiment validated the evasive ability of the Ch4os system. We validated the evasive ability of the Ch4os system by generating an evasive set of samples using the holdout set of malware. We then evaluated the classification accuracy of the MalConv network trained on the EMBER dataset against the evasive samples [2, 18]. To simulate a real-world malware triage environment, we used the Detection Testing Set to find the Optimal Threshold (OT) for the successful classification of the MalConv network. We determined the OT by maximizing the Youden's J statistic of the ROC curve. We then evaluated our evasive set using the found OT. We compared our results against two state-of-the-art RL-based function malware generation systems [13] and [1] with the same training and holdout sets as benchmarks. We also tested the evasive ability of randomly generated bytes and the unmodified malware for further benchmarks for the Ch4os system. Finally, we tested the evasive ability of a network only trained through BCP and the BCP-NO-GAN networks to further study the efficacy of Ch4os. Three metrics were used to measure the evasive experiment. The first metric used was accuracy. This accuracy refers to the accuracy of the MalConv network that was used as the target anti-malware engine. The lower the accuracy, the higher the evasive ability of the system. The second metric used was False Negative (FN). FN indicate the number of malware samples that were incorrectly classified as benign. This metric was used to measure the rise or fall in the number of evasive samples compared to the unmodified malware. As well, for measuring the MalConv network on the Detection testing set we used the metrics Area Under the ROC curve (AUC), Accuracy, F1, Precision, Recall, False Negatives, and Optimal Threshold.

Table 4: MalConv-Ember model on results on the Detection Testing Set.

Model	**AUC**	**Accuracy**	**F1**	**Precision**	**Recall**	**FN**	**OT**
MalConv-Ember [2]	0.9134	0.9134	0.9129	0.9217	0.9042	327	0.0007

As can be seen in Table 3, Ch4os outperformed GAN-No-BCP on both the training and validation sets. In both training and validation, the Ch4os generator had a significantly lower loss (82% decrease and 79% decrease, respectively). Ch4os also had a higher reward than the GAN-No-BCP generator on both sets. For the discriminator of the GAN, the loss was much higher against Ch4os compared to GAN-No-BCP, and the accuracy was much lower against the Ch4os generator.

Table 5: Results of MalConv-Ember network against different adversarial sets.

Evasion Method	**Accuracy**	**FN**	**% Increase of FN**	**OT**
Unmodified	0.8958	625	-	0.0007
Random	0.9705	177	28%	0.0007
Anderson *et al.* [1]	0.8965	621	99%	0.0007
Molloy *et al.* [13]	0.8903	658	105%	0.0007
GAN-NO-BCP	0.9677	194	31%	0.0007
BCP	0.7947	1232	197%	0.0007
Ch4os	**0.7323**	**1606**	**257%**	0.0007

The results of the MalConv-Ember [2] network can be seen in Table 4. From these results, we found that the optimal classification threshold was 0.0007. This was the OT used to compare Ch4os to other methods.

The results of the evasive experiment can be seen in Table 5. As described above, the OT from Table 4 is used as the OT for this experiment. Ch4os performed best against all other baselines in generating adversarial samples. These results showed that using a generative network to create adversarial bytes was more effective than the current state-of-the-art method of choosing benign bytes from a finite set.

5 Conclusion

In this work, we propose Ch4os, the first functionality-agnostic GAN system for problem-space evasive malware generation. A limitation of the Ch4os system is that it only appends generated bytes to the end of the malware sample. Future work for Ch4os includes testing the system against industry anti-malware engines and adding adversarial machine code throughout sample executables. Future work also includes testing the Ch4os system against different anti-malware ML systems with varying datasets of malware and benignware.

References

1. Hyrum S. Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. Learning to evade static PE machine learning malware models via reinforcement learning. *CoRR*, abs/1801.08917, 2018.
2. Hyrum S. Anderson and Phil Roth. EMBER: an open dataset for training static PE malware machine learning models. *CoRR*, abs/1804.04637, 2018.
3. Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, volume 9 of *JMLR Proceedings*, pages 249–256. JMLR.org, 2010.
4. Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 2672–2680, 2014.
5. Katja Hahn and INM Register. Robust static analysis of portable executable malware. *HTWK Leipzig*, 134, 2014.
6. Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on GAN. In *Data Mining and Big Data - 7th International Conference, DMBD 2022, Beijing, China, November 21-24, 2022, Proceedings, Part II*, volume 1745 of *Communications in Computer and Information Science*, pages 409–423. Springer, 2022.
7. Joakim Kargaard, Tom Drange, Ah-Lian Kor, Hissam Twafik, and Emlyn Butterfield. Defending it systems against intelligent malware. In *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, pages 411–417, 2018.

8. Jin-Young Kim, Seok-Jun Bu, and Sung-Bae Cho. Malware detection using deep transferred generative adversarial networks. In *Neural Information Processing - 24th International Conference, ICONIP 2017, Guangzhou, China, November 14-18, 2017, Proceedings, Part I*, volume 10634 of *Lecture Notes in Computer Science*, pages 556–564. Springer, 2017.
9. Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
10. Guanxiong Liu, Issa Khalil, and Abdallah Khreishah. Gandef: A GAN based adversarial training defense for neural network classifier. In *ICT Systems Security and Privacy Protection - 34th IFIP TC 11 International Conference, SEC 2019, Lisbon, Portugal, June 25-27, 2019, Proceedings*, volume 562 of *IFIP Advances in Information and Communication Technology*, pages 19–32. Springer, 2019.
11. Yan Lu and Jiang Li. Generative adversarial network for improving deep learning based malware classification. In *2019 Winter Simulation Conference, WSC 2019, National Harbor, MD, USA, December 8-11, 2019*, pages 584–593. IEEE, 2019.
12. Gary McGraw and Greg Morrisett. Attacking malicious code: A report to the infosec research council. *IEEE Software*, 17(5):33–41, 2000.
13. Christopher Molloy, Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. H4rm0ny: A competitive zero-sum two-player markov game for multi-agent learning on evasive malware generation and detection. In *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 22–29. molloy-3.
14. Zahra Moti, Sattar Hashemi, and Amir Namavar. Discovering future malware variants by generating new malware samples using generative adversarial network. In *2019 9th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 319–324, 2019.
15. Rakesh Nagaraju and Mark Stamp. Auxiliary-classifier GAN for malware analysis. *CoRR*, abs/2107.01620, 2021.
16. Ehsan Nazari, Paula Branco, and Guy-Vincent Jourdan. Using CGAN to deal with class imbalance and small sample size in cybersecurity problems. In *18th International Conference on Privacy, Security and Trust, PST 2021, Auckland, New Zealand, December 13-15, 2021*, pages 1–10. IEEE, 2021.
17. Matt Pietrek. An in-depth look into the win32 portable executable file format, part 2. *MSDN Magazine, March*, 2002.
18. Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K. Nicholas. Malware detection by eating a whole EXE. In *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, volume WS-18 of *AAAI Workshops*. AAAI Press, 2018.
19. Shun Tobiyama, Yukiko Yamaguchi, Hajime Shimada, Tomonori Ikuse, and Takeshi Yagi. Malware detection with deep neural network using process behavior. In *40th IEEE Annual Computer Software and Applications Conference, COMPSAC Workshops 2016, Atlanta, GA, USA, June 10-14, 2016*, pages 577–582. IEEE Computer Society, 2016.
20. Harshit Trehan and Fabio Di Troia. Fake malware generation using HMM and GAN. In *Silicon Valley Cybersecurity Conference - Second Conference, SVCC 2021, San Jose, CA, USA, December 2-3, 2021, Revised Selected Papers*, volume 1536 of *Communications in Computer and Information Science*, pages 3–21. Springer, 2021.