A Generative Adversarial Network Based Approach to Malware Generation Based on Behavioural Graphs

Ross A. J. McLaren, Kehinde O. Babaagba$^{[0000-0003-0786-2618]}$, and Zhiyuan Tan$^{[0000-0001-5420-2554]}$

School of Computing, Edinburgh Napier University, Edinburgh EH10 5DT, United Kingdom `40174116@live.napier.ac.uk`,{`K.Babaagba`,`Z.Tan`}`@napier.ac.uk`

Abstract. As the field of malware detection continues to grow, a shift in focus is occurring from feature vectors and other common, but easily obfuscated elements to a semantics based approach. This is due to the emergence of more complex malware families that use obfuscation techniques to evade detection. Whilst many different methods for developing adversarial examples have been presented against older, non semantics based approaches to malware detection, currently only few seek to generate adversarial examples for the testing of these new semantics based approaches. The model defined in this paper is a step towards such a generator, building on the work of the successful Malware Generative Adversarial Network (MalGAN) to incorporate behavioural graphs in order to build adversarial examples which obfuscate at the semantics level. This work provides initial results showing the viability of the Graph based MalGAN and provides preliminary steps regarding instantiating the model.

Keywords: Malware, Malware Detection, Adversarial Examples, Generative Adversarial Network (GAN), Behavioural Graphs.

1 Introduction

Malware attack landscape has evolved over time as malware authors and attackers now employ a number of sophisticated techniques in launching attacks. Some of these include the use of encryption [9], polymorphism [16] among others to evade detection. Consequently, most traditional detection mechanisms are vulnerable in defending against these new intrusive techniques.

As more and more of the world becomes connected through the internet, or automated through technological advancements, defence against attacks on these systems becomes a growing priority. As attackers become more sophisticated it can be a struggle for defence vendors to keep up as it is not enough if they can fix the issue after it occurs, preemptive defence of systems will always be superior. Due to this, a crucial area in software security research is trying to anticipate what attackers will do to keep systems safe. One such way to do this is to generate adversarial examples of malware in a secure environment with the sole purpose of training automated detectors to be able to detect a wide range of new, never before seen samples.

Machine learning has shown encouraging results as a useful tool in the detection of malicious software. It has been used to learn patterns within header files, raw bytes and instruction sequences which identify a piece of software as malicious or benign [18]. Commonly, these machine learning solutions are created as a black box in an effort to increase the security of the underlying system. This follows an idea that if the attacker cannot access the underlying machine learning algorithm then it is far more difficult for them to exploit it for their own ends. This, however, has presented a weakness in such thinking as attackers are able to probe the network and from that, determine which design features will be flagged as malicious allowing them to design software that evades detection by the machine learning models [13].

In addressing the aforementioned weakness, adversarial learning [3,4,11] was introduced. The aim of an adversarial method is to create malicious software which exploits the loopholes in other machine learning models. To this effect, they collect data using models such as deep, convolutional neural networks used in malware analysis in order to categorize what they will allow as a benign piece of software. Another network is then trained to create malware which should be considered benign by the first network, initially this yields poor results but over numerous iterations the generated malware signature should successfully evade detection. This is known as gradient based adversarial generation [10] and has been done to great success with minuscule amounts of the malicious software needing to be changed to evade detection.

In recent years, attackers have been able to stay ahead of defenders by utilising numerous obfuscation techniques. To combat this, steps are being taken to move towards a broader semantics based, higher level approach to detection [1]. Malware behavioural graphs are a step towards this, as they utilise a Control Flow Graph (CFG) as a signature. CFGs are used because the domain of a CFG is at least as complex as the domain of strings in the same function [5]. A CFG consists of linked nodes with each node being a different element of the program. In assembly, these could be the different calls such as jmp, call or end [7]. This allows a mapping of the flow of data and functions within a greater piece of malware [8]. This not only provides the information on the number of times a function is called, but the order in which they were called thus mapping out the actual behaviour of the malware [1]. The structures of the created graphs can be compared to identify crossovers and similarities in the behaviour of two programs, allowing the classification of new malware into existing families based on the behavioural patterns they exhibit [5]. Unlike other classical detection techniques, CFG detection improves its detection accuracy as the size of the program increases [8]. This is due to a larger program creating a more detailed end graph. It is also resistant to common obfuscation techniques due to not being based on checks of specific vector or string details of a signature [7].

Generative Adversarial Networks (GANs) [15] have already been used to much success in the area of adversarial example generation, however, these attempts suffer from a lack of diversity within the malware used and the limited features they use for generation. This paper seeks to outline a solution to these problems by building the examples based on behavioural graphs rather than simple feature semantics. Two research questions are addressed in this work and they include:

1. To what extent can a Graph based Malware Generative Adversarial Network (MalGAN) be used in creating adversarial samples?
2. How does the Graph based MalGAN compare with the original MalGAN?

The rest of the paper is structured as follows. The second section presents related work. Section III describes the research methodology. The results and evaluation are discussed in Section IV and we present the conclusion and future work in Section V.

2 Related Work

Adversarial examples of malware have been utilized to show weaknesses in machine learning based malware detection systems [13]. These examples are capable of bypassing the traditional black-box malware detection systems by allowing attackers to infer the features likely to be flagged as malware [14]. Till very recently, neural network based models for generating adversarial examples have been primarily gradient based. These have had some success but struggle to reach a detection rate of zero. They are also able to be quickly retrained against by most defensive methods [17].

MalGAN has been introduced as a new model for generating adversarial examples [13]. It is based on the GAN model for generators and uses a system which comprises a generator and discriminator. It takes API features as an example of how to represent a program. The main difference between this given model and current models is the fact that the generator can update dynamically in relation to the feedback it receives from the black-box detector. Currently, most models use a static gradient to produce examples instead. The generator transforms an API feature vector into an adversarial malware example. It creates a binary version of the feature vector of a piece of malware, showing a zero if the API feature does not exist and a one if it does. This is concatenated with a random noise function, altered to return values between 0 and 1 in an effort to add non-malicious features to the example. This is fed into a multi-layer feed-forward perceptron with the last layer's activation function being sigmoid with a restricted output range of 0 to 1 [13]. During the generation, only irrelevant features are considered when adding features to the malware. The removal of features is not considered as this could result in a cracked malware. The black-box detector used is also a multi-layered feed-forward neural network that takes in a feature vector as an input and outputs if the vector is malicious or benign [13]. The training data used for the black box detector consists of a mix of the generator's examples, and benign pieces of software. MalGAN is also significantly more dynamic than a gradient based approach which allows it to keep up with advances in security. This is because MalGAN only needs to be retrained on the new detectors in order to be able to create adversarial examples against it.

It has a limitation of only currently generating feature vector examples, which could make it difficult to produce examples which can fool higher-level, broader semantics based detectors which utilize the behaviour of the malware rather than its API features [14]. That is why this paper seeks to present a new generator based on MalGAN that generates graph-based examples rather than feature vectors and evaluate how this affects the detection rate against its examples. MalGAN has also been suggested to have some issues which limit its functionality in a real-world application [14]. For example, by using a set of the features in a piece of malware rather than the entirety of the feature list, it limits to what degree the adversarial examples it generates can be properly used to actually harden machine learning based malware detection approaches. Also, by having the generator and discriminator built and trained within the same process as the generator, it creates an unrealistic advantage that traditionally attackers would not have [14]. In the case of both of these versions of MalGAN, they are able to reach under three percent detection rates based on the True Positive Rate (TPR).

3 Methodology

3.1 Overview

The implementation of the Graph based MalGAN comprises of multiple components. It required the collection of API features from analysed examples, the connection of dependencies between these calls and the encoding into a format a neural network can utilise. These steps led on to the model construction and neural network training and testing. This section gives a brief overview of these parts and presents an explanation of how the model works.

Dynamic Analysis This was used in order to get the correct API calls that were used at run time for each program, rather than relying on all possible API calls as is present in static analysis. To facilitate this, a Windows Operating System was procured and loaded onto a Virtual Box Machine. This machine was then hardened and set up for malware analysis, being completely cut off from being able to connect to the host or the Internet and had all of the Windows Defender settings turned off. The host machine was also prepared as per the specifications listed by the Cuckoo[1] sandbox website and finally the Cuckoo agent was loaded onto the Virtual Machine to allow for secure communication between the host and machine. After the malware binaries were analysed on the Virtual Machine, several python scripts were developed which first extracted the API calls from the Cuckoo log which consists of a JSON file, then removed any exact duplicate calls before assembling them into a behavioural graph that

[1] Cuckoo - https://cuckoosandbox.org/

represents the initial program. One-hot encoding was then used to have valid inputs for the neural network.

The Neural Network (NN) Model The NN model put forward for Graph based MalGAN consists of a generator and discriminator as in a typical GAN architecture. The generator takes a concatenated input consisting of a malware example and a noise vector and outputs an adversarial example that the discriminator takes as an input. The discriminator then determines whether or not the example is malicious based on the classification given to it from an outside detector, the detector an attacker is trying to bypass in a real world scenario. The generator and discriminator train with the goal being to minimise the number of samples correctly identified by the external detector as shown in Figure 1.

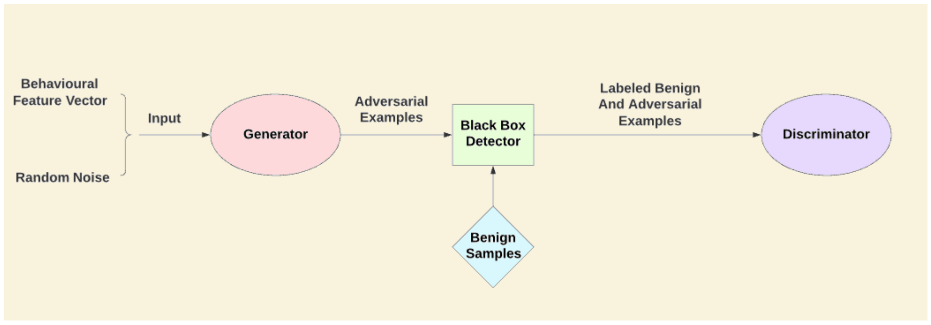


Fig. 1: A simple overview of the entire network model

3.2 Input Formatting and Encoding

Behaviour-based malware detection focuses on security critical operations, and this was taken into consideration when deciding what operations and API calls to focus on for the API graphs. Unlike in the original MalGAN paper, none of the API call graphs were capped. The resulting behavioural graph for each program only counts unique API call graphs, as duplicate behaviours are rare and overall unlikely to affect the behavioural obfuscation displayed in this paper.

Whilst all of the calls selected appear in benign samples, the combination of them can lead to intentional malicious design. From this, the generator can learn what combination of behaviours are likely to be assessed as malicious and how to alter them to avoid detection without cracking the underlying malware.

Behavioural graphs are an un-ordered set of API call graphs for a given program and are used as the basis of behaviour-based malware detection. By cleaning and processing a set of API calls from a given program, dependencies can be built between calls which act upon the same Operating System resource, this lets a user view the API calls in a graph structure. This graph structure presents the function of a piece of code within the program in a human friendly format, allowing for better understanding of the behaviour of the code. A behavioural graph is the next step up, and instead of displaying the behaviour of a particular section of code, it combines all of them into an un-ordered graph which describes the behaviour of the whole program.

Both the API call and its relevant Operating System resources are extracted from an analysed program and used to construct the API call graph. The API calls are assigned a numerical value in order to aid in the labelling during the encoding stage. The graph is directed and acylic with the nodes being defined as the API calls and the edges between them being their connected Operating System resource and the dependence between them. It can be defined as:

$$G = (V, E, f) \tag{1}$$

Wherein, API call graph G has a set of nodes, V, and a set of edges represented by E which is a member of $V \times V$ and an incidence function f, mapping E to V.

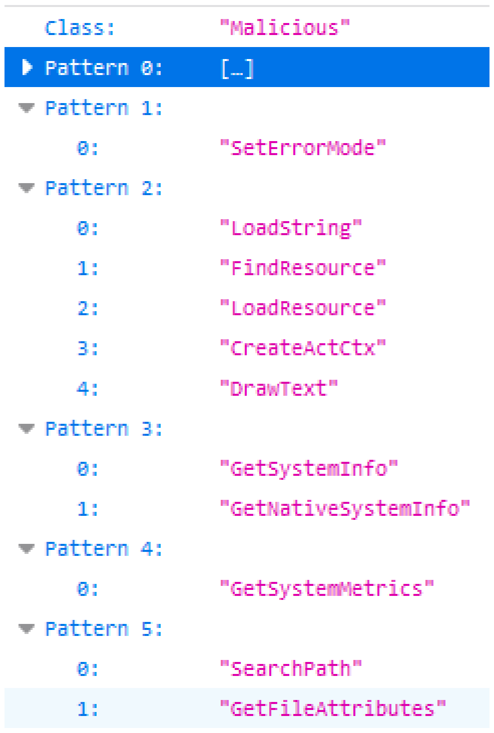
In order to allow for further classification of API call graphs within the overarching behavioural graph, different commands which serve the same purpose are treated as the same call. An example of this is seen in samples with multiple calls which all serve the purpose of creating a new key for the registry -

1. RegCreateKeyExW
2. RegCreateKeyExA
3. RegCreateKeyW

For the API call graph, all of these calls have been transformed into RegCreateKey.

Firstly, a dataset of program samples was constructed by analysing both malicious and benign samples within the Cuckoo sandbox. This dataset ΔB_G contained sixteen-hundred malicious examples and five-hundred-and-fifty benign examples. After being analysed, the API calls, their Operating System resource arguments and the time they were called are all extracted from the Cuckoo reports and held in a JSON file. At this point, exact duplicate calls which happen in sequence are removed as during dynamic analysis it is possible for calls to be logged which have not actually occurred and are as such, simply copies of the previous call.

From here, API calls are linked together through the dependencies present in both their Operating System arguments using both the addresses in memory and the names of the processes, unlike in relevant studies using behavioural graphs which only use the memory addresses. Whilst it should be noted that the malware may use a different name for a process than the actual process' name, it was decided to still be an important factor in correctly linking API calls within this study. These graphs are then ordered based on the time at which the call was made in order to appropriately structure the order they happen in. This is crucial for identifying the correct patterns because the transference of data between different calls within the API graphs is important for their underlying functionality.



```
Class:          "Malicious"
▶ Pattern 0:    […]
▼ Pattern 1:
    0:          "SetErrorMode"
▼ Pattern 2:
    0:          "LoadString"
    1:          "FindResource"
    2:          "LoadResource"
    3:          "CreateActCtx"
    4:          "DrawText"
▼ Pattern 3:
    0:          "GetSystemInfo"
    1:          "GetNativeSystemInfo"
▼ Pattern 4:
    0:          "GetSystemMetrics"
▼ Pattern 5:
    0:          "SearchPath"
    1:          "GetFileAttributes"
```

Fig. 2: Excerpt from a Behavioural Graph in JSON format

Once all of the patterns for a program have been successfully identified, duplicates within the same program are removed. This is because, whereas in MalGAN the objective is the generation of adversarial examples for specific malware, the objective here is training a generator which can successfully obfuscate these patterns to generate adversarial examples. At this point the dataset of behavioural graphs $\varDelta B_G$ is created by iterating through each program's behavioural graph and creating a list of every unique occurrence.

By utilising one hot encoding, each program's behavioural graph is compared to the dataset and at each pattern, given a one or zero if the pattern exists within the program.

$$\begin{gathered} \mathrm{T}_e = (V, C) \\ \mathrm{T}_{BG} \in \varDelta B_G \\ \mathrm{T}_{eVn} = \{0|1\} \\ \mathrm{T}_{eV1} = 1 \iff B_{G1} \in T_{BG} \end{gathered}$$

Where T_e is equal to an encoded program and contains a binary vector V and a class definition C which defines if T is benign or malicious when training the discriminator. T_{BG} refers to the behavioural graph for program T, $\varDelta B_G$ is the dataset of all behavioural graphs used in this work. As with one hot encoding T_{eVn} equals a one if the n^{th} value of the greater dataset exists within the behavioural graph for program T. This results in a binary vector which can be used as the inputs for a neural network.

3.3 Dataset

The dataset used for this work was the MC-binary-dataset [2]. Sixteen-hundred malicious samples and five-hundred-and-fifty benign samples were used to create the behavioural graph dataset and used for training the black box detector and generator system. The malicious dataset was split with sixty percent going to the generator system and forty percent going to the black box detector. This results in the generator having nine-hundred-and-sixty malicious examples to utilise and the black box detector having six-hundred-and-forty to train on. This split is for two reasons:-

1. It better simulates a real world scenario where an attacker is unlikely to have the exact malicious examples used to train a detection system.
2. It forces the generator to obfuscate the malicious patterns, rather than obfuscating specific malware as in both the original and improved MalGAN papers.

3.4 Network Architecture

Generator The generator is a dense, multi-layer, feed-forward network with weights classified as θ_g. The output layer is the same shape as the number of patterns within the dataset. The weighting of the graph is informed from the feedback of the discriminator and effects the distribution of the noise throughout the vector in an attempt to favour patterns which allow the generated example to appear benign. The final layer uses a sigmoid activation function for the express purpose of limiting the outputs to the range [O, 1], and in order to ensure that the malware remains functionally malicious, the generated adversarial example is combined with the original sample using the OR function, allowing the non-zero values in the adversarial example to act as pattern obfuscation. This resulting vector is transformed into a binary by affixing a one in every position where the value is greater than zero-point-five and a zero when below this threshold.

This obfuscated behavioural pattern vector is then able to be used as the input for the black box detector to be labelled, and then based on this labelling receives gradient information from the discriminator in order to improve its weighting and results. The goal of the generator is to have its adversarial examples mislabelled as benign by the detector.

Discriminator The discriminator in this instance is a substitute detector. It acts as a stand-in for the black box detector system to enforce the real world

Layer	**Activation**	**Size**
Input		
Dense	ReLU	256
Batch Normalization		
Dense	ReLu	128
Dense	ReLU	64
Batch Normalization		
Dropout(0.5)		
Output Layer	Sigmoid	

Table 1: The model used as the generator in graph based MalGAN

scenario where attackers would be unable to access the underlying system within the detector. It is fed classifications from the black box detector based on the adversarial examples and benign code given to the black box which can then be used to train the generator. It does this by learning the classification rules involved in the black box detector. Like the generator it is a dense, multi-layer, feed forward network.

As input it takes in either a benign example or an adversarial one that has been classified and labelled by the black box detector. As its purpose is as a stand-in for the black box detector it is only able to see the samples that the detector itself has labelled as malicious, rather than the data set of actual malicious examples. This ensures that it trains the generator to successfully fool the detector. As with the generator, the goal and associated loss function of the discriminator relates to lowering the number of examples correctly identified as malware by the black box detector.

Layer	**Activation**	**Size**
Input		
Dense	ReLU	256
Dense	ReLu	128
Dropout(0.5)		
Dense	ReLU	64
Output Layer	Sigmoid	

Table 2: The model used as the discriminator in graph based MalGAN

3.5 Machine Learning Models used for Comparison

Multilayer Perceptron (MLP) MLP is a deep Artificial Neural Network (ANN) which has different layers consisting of at least an input layer, a hidden layer and an output layer. Input reception is handled by the input layer, the hidden layer is termed the computation engine and decision making or predictive analysis is carried out by the output layer [19].

Random Forest (RF) RF is also referred to as random decision forest. This is a machine learning ensemble that combines several algorithms to derive better learning results. RF as the name suggests, comprises of several separate decision trees that form an ensemble with every of these trees spitting out a prediction for a class and the class that has the highest vote then becomes the prediction of the model [6].

Logistic Regression (LR) LR is commonly employed for binary data and for categorical target variables. An example would be in the prediction of an email as being either benign (0) or spam (1). A logit transformation is used to force the Y value to take on differing values between 0 and 1. The probability $P = 1/(1 + e - (c + bX))$ is initially calculated after that X is then linearly correlated to $log_n P/(1 - P)$ [12].

4 Results Discussion and Evaluation

Here, in answering the two research questions, (1) To what extent can a Graph based MalGAN be used in creating adversarial samples? and (2) How does the Graph based MalGAN compare with the original MalGAN?, we present an overview of the performance of the Graph based MalGAN model which is provided in Table 3 which compares MalGAN's True Positive Rate (TPR) against different machine learning algorithms. The TPR is the percentage of the samples that the black box has correctly classified and the lower the value is, the better the model has performed. We present a graph for each learning algorithm, visually showing the change in the TPR throughout the training and testing phases of each epoch.

4.1 Overall Performance

Table 3 presents the TPR of malware detection for the blackbox detector showing how many of the original malware samples the different algorithms were able to classify, how many of the original MalGAN samples the algorithms were able to classify and finally, how many of the Graph based MalGAN samples that they were able to classify. It is important to also include the TPR figures for the malware examples before they are obfuscated as high rates of correct classification at this stage add validity to the behavioural graph method for presenting malware. If the detector was incapable of classifying malware correctly even before the adversarial examples were generated then it could be determined that the behavioural graph method was flawed and as such, results for the classification of their adversarial examples would be invalid. The results for the Graph based MalGAN are also compared to the original MalGAN in order to determine if any improvements to the detection rate has been observed. In this instance, as the objective is successfully generating examples which fool the detector, a lower true positive rate is desired.

It can be observed that against the unaltered malware examples, the blackbox detector was able to successfully classify the behavioural graph's patterns as malicious. This adds validity that the behavioural graph method for malware detection is a valid and good method of detection and therefore a system which can successfully obfuscate these patterns is useful and results gathered from such obfuscation are valid. The Graph based MalGAN was able to successfully reduce the TPR for the blackbox detector consistently against multiple different algorithms across both the training and testing set. From this, it can be observed that neither the discriminator nor the generator became over fitted against the training set. This infers that the Graph based MalGAN would consistently work against new datasets and against multiple algorithms implying robustness against change in the model.

TPR for Blackbox Detector						
	Original Malware		**Original MalGAN**		**Graph Based MalGAN**	
	Training Set	Testing Set	Training Set	Testing Set	Training Set	Testing Set
MLP	93.00%	94.00%	0.00%	0.00%	1.90%	2.70%
RF	98.00%	100.00%	0.20%	0.19%	0.00%	1.00%
LR	97.00%	96.00%	0.00%	0.00%	0.80%	2.00%

Table 3: TPR for each Algorithm

MLP The progress for how the TPR changed over training was mapped to a graph for easy visualisation. It can be observed that for the algorithm MLP,

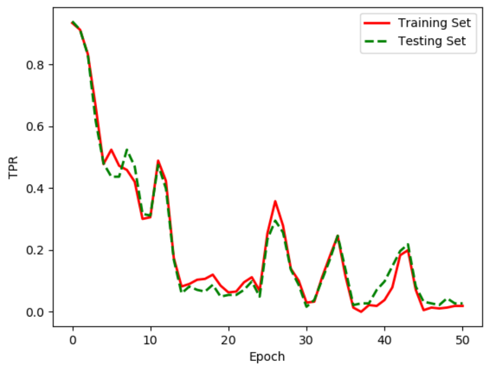


Fig. 3: TPR progress Graph for MLP algorithm

the TPR had some spikes where it increased, but overall trended towards zero.

The spiked increases may be caused by low variance within the dataset, and the randomness at which the noise is applied.

RF The progress for how the TPR changed over training was mapped to a graph for easy visualisation. For algorithm RF, the TPR trended quickly towards zero

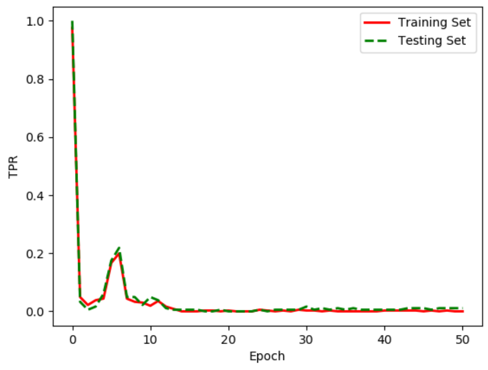


Fig. 4: TPR progress Graph for RF algorithm

before spiking and then had a gradual trend towards zero.

LR The progress for how the TPR changed over training was mapped to a graph for easy visualisation. For algorithm LR, the TPR had a spiky descent

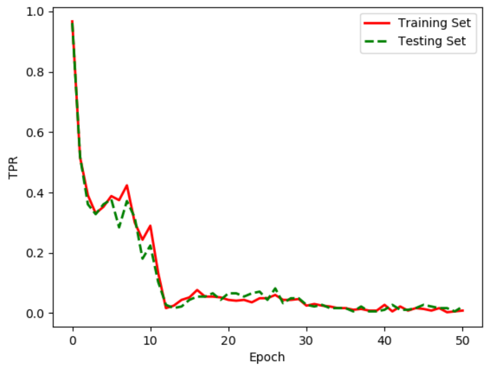


Fig. 5: TPR progress Graph for LR algorithm

before evening out and trending towards zero.

4.2 Evaluation

Whilst the model performed well, the input range given by behavioural graphs is very large and makes the generation of adversarial examples challenging as a balance must be struck between generating enough noise to create adversarial examples and not generating so much as to create patterns which belong in neither dataset.

Graph based MalGAN versus Original MalGAN in terms of TPR Whilst the presented model was able to achieve results of less than two percent detection, this is not as low as the original MalGAN model which produced results all of which were less than one percent as seen in Table 3. It should be noted, however, that behavioural graph based interpretations of malware capture more semantic information than simple feature vectors as they rely on specific patterns of API features existing within code rather than simply the existence of a specific call. Being able to obfuscate adversarial examples against detectors which work at the semantics level of the code is a progressive step forward in the field and as this is one of the first attempt as such obfuscation, the low detection rates of under two percent shows its validity moving forward and that it should be further researched and improved upon. Graph based MalGAN also continues the same robustness as its predecessor as demonstrated by their comparable differences in between algorithms.

Graph based MalGAN versus Original MalGAN in terms of Graphs mapping TPR over Epochs Whilst the original MalGAN paper does not show all of their graphs mapping the change in TPR over the number of epochs, those that they do show demonstrate a similar gradient as those presented by the Graph based MalGAN. One of the few differences between them is the rate at which the TPR lowers with the original MalGAN presenting smoother graphs with a gradual decrease over the number of epochs versus the spiky graphs presented in this paper. A possible reason for this is the size of the patterns being obfuscated. Graph based MalGAN uses an input layer size of one-thousand-eight-hundred-and-ten whereas the original MalGAN uses an input layer of one-hundred-and-twenty-eight. This not only reduces the likelihood that the Graph based MalGAN will randomly assign the correct vector to obfuscate a given piece of malware but increases the time it will take to be able to obfuscate the example. It also makes it more likely for the detector to classify an example as malicious unless the generator is correctly learning benign features to add to its examples. Another possible explanation could be a difference in the size of the datasets. The original MalGAN uses a larger data set for both their benign and malicious examples as one of the limitations for this work was finding an equally sized dataset to use. This could result in the detector simply learning the entire data set and classifying any example that does not exist in its original benign set as malicious.

5 Conclusion and Future Work

The work presented in this paper sought to build on the existing work of MalGAN and present a modern take on adversarial malware generation based on similar research found in the field of malware detection. It presented code not only for the feature extraction, ordering, graphing and encoding of malware into a behavioural pattern based model but also a neural network capable of generating similar but deceiving models. A wider range of both malware and benign samples could have been collected for the training of this model and it is impossible to determine if that has impacted the results gathered here.

Moving forward the work outlined in this paper could be utilised to generate completely new malware based on associated behaviours. In theory, if the behavioural graph approach were to be combined with the original MalGAN's feature vector approach and applied to a semi-random walking function it would be possible to define new malicious patterns for code. Undertaking such a task would potentially involve the combination of both a GAN as presented here and some form of auto-encoder system to handle the wide array of potential variables involved in creating such a pattern generator. Future work could also be undertaken to combine sparse auto encoders within the generator model as this could also potentially lead to a generator capable of creating malware with undocumented behaviour and patterns.

References

1. Anderson, B., Quist, D., Neil, J., Storlie, C., Lane, T.: Graph-based malware detection using dynamic analysis. Journal in computer Virology **7**(4), 247–258 (2011)
2. Andrade, E.d.O.: Mc-dataset-binary (2018), `https://figshare.com/articles/MC-dataset-binary/5995408/1`
3. Babaagba, K.O., Tan, Z., Hart, E.: Nowhere metamorphic malware can hide - a biological evolution inspired detection scheme. In: Wang, G., Bhuiyan, M.Z.A., De Capitani di Vimercati, S., Ren, Y. (eds.) Dependability in Sensor, Cloud, and Big Data Systems and Applications. pp. 369–382. Springer Singapore, Singapore (2019)
4. Babaagba, K.O., Tan, Z., Hart, E.: Automatic Generation of Adversarial Metamorphic Malware Using MAP-Elites. In: P.A. Castillo et al (ed.) 23rd European Conference on the Applications of Evolutionary and bio-inspired Computation. pp. 1–16. Springer-Verlag New York, Inc., Seville (2020)
5. Bonfante, G., Kaczmarek, M., Marion, J.Y.: Control flow graphs as malware signatures (2007)
6. Breiman, L.: Random forests. Machine Learning **45**(1), 5–32 (2001)
7. Cesare, S., Xiang, Y.: Malware variant detection using similarity search over sets of control flow graphs. In: 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications. pp. 181–189. IEEE (2011)
8. Cesare, S., Xiang, Y., Zhou, W.: Control flow-based malware variantdetection. IEEE Transactions on Dependable and Secure Computing **11**(4), 307–317 (2013)
9. Chuman, T., Sirichotedumrong, W., Kiya, H.: Encryption-then-compression systems using grayscale-based image encryption for jpeg images. IEEE Transactions on Information Forensics and security **14**(6), 1515–1525 (2018)

10. Guo, C., Sablayrolles, A., Jégou, H., Kiela, D.: Gradient-based adversarial attacks against text transformers. arXiv preprint arXiv:2104.13733 (2021)
11. He, R., Li, Y., Wu, X., Song, L., Chai, Z., Wei, X.: Coupled adversarial learning for semi-supervised heterogeneous face recognition. Pattern Recognition **110**, 107618 (2021)
12. Hoffman, J.I.: Chapter 33 - logistic regression. In: Hoffman, J.I. (ed.) Basic Biostatistics for Medical and Biomedical Practitioners (Second Edition), pp. 581 – 589. Academic Press, 2 edn. (2019)
13. Hu, W., Tan, Y.: Generating adversarial malware examples for black-box attacks based on gan. arXiv preprint arXiv:1702.05983 (2017)
14. Kawai, M., Ota, K., Dong, M.: Improved malgan: Avoiding malware detector by leaning cleanware features. In: 2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC). pp. 040–045 (Feb 2019)
15. Maeda, H., Kashiyama, T., Sekimoto, Y., Seto, T., Omata, H.: Generative adversarial network for road damage detection. Computer-Aided Civil and Infrastructure Engineering **36**(1), 47–60 (2021)
16. Popli, N.K., Girdhar, A.: Behavioural analysis of recent ransomwares and prediction of future attacks by polymorphic and metamorphic ransomware. In: Computational Intelligence: Theories, Applications and Future Directions-Volume II, pp. 65–80. Springer (2019)
17. Saxe, J., Berlin, K.: Deep neural network based malware detection using two dimensional binary program features. In: 2015 10th International Conference on Malicious and Unwanted Software (MALWARE). pp. 11–20 (2015)
18. Singh, J., Singh, J.: A survey on machine learning-based malware detection in executable files. Journal of Systems Architecture p. 101861 (2020)
19. Taud, H., Mas, J.: Multilayer Perceptron (MLP), pp. 451–455. Springer International Publishing, Cham (2018)