MANIS: Evading Malware Detection System on Graph Structure

Peng Xu
Technical University of Munich
peng@sec.in.tum.de

Bojan Kolosnjaji
Technical University of Munich
kolosnjaji@sec.in.tum.de

Claudia Eckert
Technical University of Munich
eckert@sec.in.tum.de

Apostolis Zarras
Maastricht University
apostolis.zarras@maastrichtuniversity.nl

ABSTRACT

Adversarial machine learning has attracted attention because it makes classifiers vulnerable to attacks. Meanwhile, machine learning on graph-structured data makes great achievements in many fields like social networks, recommendation systems, molecular structure prediction, and malware detection. Unfortunately, although the malware graph structure enables effective detection of malicious code and activity, it is still vulnerable to adversarial data manipulation. However, adversarial example crafting for machine learning systems that utilize the graph structure, especially taking the entire graph as an input, is very little noticed. In this paper, we advance the field of adversarial machine learning by designing an approach to evade machine learning-based classification systems, which takes the whole graph structure as input through adversarial example crafting. We derive such an attack and demonstrate it by constructing MANIS, a system that can evade graph-based malware detection with two attacking approaches: the n-strongest nodes and the gradient sign method. We evaluate our adversarial crafting techniques utilizing the Drebin malicious dataset. Under the white-box attack, we get a 72.2% misclassification rate only by injecting 22.7% nodes with the n-strongest node. For the gradient sign method, we obtain a 33.4% misclassification rate with 36.34% node injection. Under the gray-box attack, the performance of our adversarial examples is evenly significant, although attackers may not have the complete knowledge of the classifiers' mechanisms.

ACM Reference Format:
Peng Xu, Bojan Kolosnjaji, Claudia Eckert, and Apostolis Zarras. 2020. MANIS: Evading Malware Detection System on Graph Structure. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3341105.3373859

1 INTRODUCTION

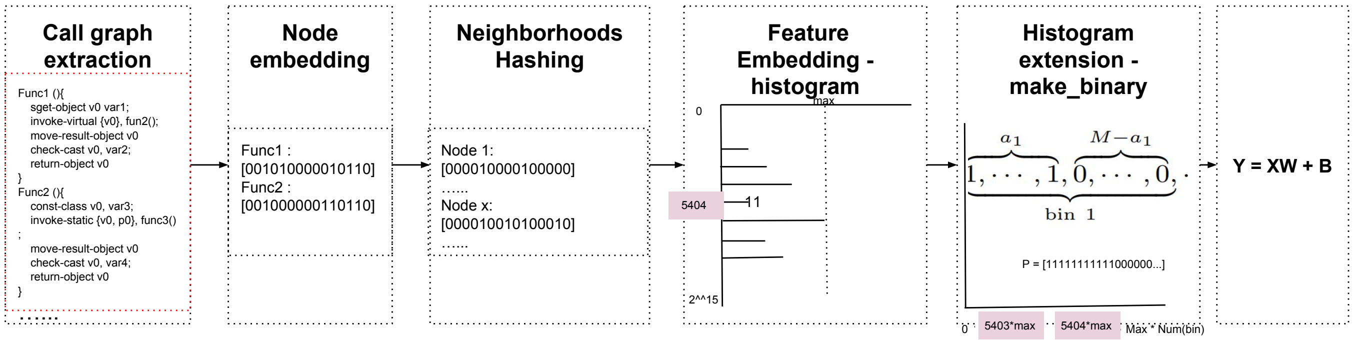
Graph structure-based machine learning networks have become popular in many fields because they are discrete and universal. More precisely, classification on graph structure can assist, especially in complicated and informative networks such as social networks, molecular chemistry, and biology, as well as malware detection [19]. At the same time, adversarial machine learning gains ground day by day [8, 16, 17]. Adversarial examples, crafted by small perturbation of inputs, can influence most of the classifiers based on machine learning and deep neural networks. Though, one question that is yet to be answered is how to craft adversarial examples for the underlying classification systems on graph structure without removing nodes and edges from the original graph. This question has attracted little attention from the scientific community so far.

All existing approaches try to tackle the problem mentioned above in two ways. On the one hand, from the standpoint of crafting adversarial examples on the graph structure, almost all works primarily focus on adding/removing nodes and edges in the graph to craft adversarial examples [5, 24]. However, when, for instance, crafting malicious mobile apps, we cannot remove nodes and edges from the function call graph if we want to keep the app's functionality. Therefore, we cannot use the existing approaches to classify malware under adversarial examples automatically [5, 24].

On the other hand, in the area of adversarial example crafting for classification, many works do not consider the classification on the graph structure. For example, researchers have proposed a gradient-based attack to evade deep networks by only changing a few specific bytes at the end of each malware sample while preserving its functionality [11]. This attack, though, considers the bytes of binary, rather than the graph structure, as the input. Others have suggested attacking recurrent neural networks with the discrete characteristics of the text provided as input [3]. Last but not least, there exist several works that discuss how to craft adversarial examples for images [14, 15, 21].

In this paper, we propose MANIS, an adversarial example crafting framework capable of evading machine learning-based malware detection systems using the whole graph data classification and maintaining all nodes in the original graph in order to preserve the malware's functionality. We leverage two core methods, which can impose classifiers to produce erroneous results. The first method uses the n-strongest nodes; these nodes have the most influence on their neighbors. In brief, we insert the strongest nodes iteratively until we can evade the classification boundary in order to misclassify the malicious sample to a benign one. The second method uses the gradient sign approach to craft the adversarial examples. Due to the discrete feature of the graph representation, we cannot directly calculate the gradient of the objective function like differential functions. Thus, we aim at finding the gradient direction and use it to guide the insertion of nodes to the graph. We also implement a projection process, mapping the updated value depending on the gradient direction to graph's histogram representation.

SAC '20, March 30-April 3, 2020, Brno, Czech Republic
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3373859


Call graph extraction
Node embedding
Neighborhoods Hashing
Feature Embedding - histogram
Histogram extension - make_binary
Func1 : [001010000010110]
Func2 : [001000000110110]
Node 1: [000010000100000]
......
Node x: [000010010100010]
......
0
max
5404
11
2^^15
$\underbrace{\overbrace{1,\cdots,1}^{a_1},\overbrace{0,\cdots,0}^{M-a_1}}_{\text{bin } 1},\cdot$
P = [11111111111000000...]
0
5403*max
5404*max
Max * Num(bin)
Y = XW + B

```
Func1 (){
    sget-object v0 var1;
    invoke-virtual {v0}, fun2();
    move-result-object v0
    check-cast v0, var2;
    return-object v0
}
Func2 (){
    const-class v0, var3;
    invoke-static {v0, p0}, func3()
;
    move-result-object v0
    check-cast v0, var4;
    return-object v0
}
......
```

14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
binop	unop	invoke	staticop	instanceop	arrayop	branch	jump	throw	new	test	monitor	return	move	nop

Figure 1: Adagio malware detection. Top: Detection system includes six steps. Botton: 15-Dalvik instruction categories.

To evaluate MANIS, we use the Android app ecosystem. More precisely, we train our framework with 49,947 benign applications from the AndroZoo project [1] and with 5,560 malicious applications retrieved from the Drebin malware samples [4]. After that, we apply our attacking methods on the malware samples we have using n-strongest nodes, gradient sign method, and random nodes injection. The results reveal that, with the gradient sign method, the classifier misclassified 33.4% malware to benign only with 36.34% node injection, whereas, with the n-strongest nodes method, it misclassified 72.2% malware to benign only with 22.7% node injection.

In summary, we make the following main contributions:

- We develop the first framework to attack a classification system using the whole graph structure without removing any node in the original graph.
- We implement our attack against a widely-used Android malware detection framework on the graph structure.
- We evaluate our adversarial methods using real malware and test them for both white-box and gray-box attacks.

2 BACKGROUND

2.1 Adversarial Machine Learning

Adversarial machine learning spans both the analysis of vulnerabilities in machine learning algorithms and algorithmic techniques that generate more robust machine learning networks. On the one hand, we consider that the attacks against machine learning algorithms [12, 13] have two directions: (i) attacks against the classifier which means attackers can change their behaviors to mislead the classifier at testing time [5, 8, 9, 11, 15, 21, 24] and (ii) attacks that modify the training dataset in order to construct a corrupted classifier, also known as poisoning attack [10, 22]. On the other hand, adversarial examples crafting (i.e., a process to utilize the small perturbations to legitimate inputs with the intent of misleading machine learning models) is one of the most popular approaches. The perturbations are designed to be as small as possible to mislead classifiers rather than a human being as a user to classify the resulting input.

2.2 Android Apps and Code Analysis

Android applications are primarily written in the Java programming language, compiled into DEX bytecode, and executed in either the Dalvik virtual machine or the Android Runtime (ART).[1] Each Android application is composed of *DEX* files, *AndroidManifest.xml*, Android resources files as well as other configurations.

Static analysis of Android apps provides an understanding of the code structure without executing the program. It can help to ensure that the code adheres to industry standards. A typical static analysis process starts by expressing the code of the analyzed app to some abstract models. Call graph analysis [23], or API call graph analysis [18], is a technique to show the functions' transfer from callers to callees. It represents every possible run path of the program with a static method. In real code analysis, the call graph is generally over-approximating because of the points-to problems and undecidable targets issues.

2.3 Malware Detection on Graph

One of the most popular machine learning networks for malware detection on a graph is the Adagio network proposed by Hugu et al. [7] and is illustrated in Figure 1. The extracted call graph is a directed graph containing nodes for each application's functions and edges from callers to callees. Let us present this graph as a 4-tuple $G = (V, E, L, \ell)$, where V is a finite set of nodes. $E \subseteq V \times V$ is shown as the set of directed edges. L is the multiset of labels in the graph, and $l : V \longrightarrow \ell$ is the labeling function. Adagio defines 15 distinct categories of instructions (i.e., bottom part of Figure 1) based on their functionalities.[2] The labeling function ℓ is defined by One Hot Encoding with 15 distinct instruction categories. Consequently, the set of labels L is presented by a subset of all possible 15-bit sequences.

Overall, the detection system on the graph can be summarized as the following expressions:[3]

[1]https://source.android.com/devices/tech/dalvik
[2]Dalvik's specification defines 256 instruction in total
[3]More detailed information can be found in the Adagio research paper [7]

$$f(G_h(V,E)) = W * X + B$$
$$X = S\_Bin(S\_Hist(S\_Hash(embed(V,E)))), \tag{1}$$

where $embed()$ stands for the node embedding, S_Hash means the neighborhood hashing, and S_Hist states the feature embedding. Lastly, S_Bin represents a function which maps the histogram H to a P-dimensional vector with bin; $P = N * M$, M is the maximum value of all bins and N is the number of bins in each histogram. Each bin of the histogram is associated with a M dimension. These dimensions are filled with several "1"s according to the value of a_i, whereas the remaining $M - a_i$ dimensions are set to 0. This means, for example, node 5404 (N_{5404}) in bin_i appears 11 times, then bin_i represents 11 as $[1,1,1,1,1,1,1,1,1,1,1,0,0,\dots]_{5404}$, like the P vector in Figure 1.

3 DESIGN

3.1 Overview

We associate each graph $g_i \in G$ with a label $y \in Y = \{-1, 1\}$, where $Y = 1$ means that the graph is classified as malicious in the binary classification and $Y = -1$ that the graph is benign. The dataset $D : \{(G, g_i, y_i), i \in N\}$ is represented by pairs of graphs and their labels. Examples of such tasks include, among others, classifying the drug molecule graphs according to their functionality as well as a social community characteristic by features. In this type of graph representation, the classifier $f \in F : G \rightarrow Y$ is optimized to minimize the following loss function:

$$Loss = 1/N \sum_{i=1}^{N} Loss(f(G_i), y_i), \tag{2}$$

where $Loss(*)$ is the cross entropy by default.

However, for the adversarial examples crafting task, our purpose is to maximize the loss function. In other words, we need to increase the possibility of misclassification with small perturbations of the original graph. Given a trained classifier f and an instance from the dataset $D : \{(G, g_i, y_i), i \in N\}$, the attacker aims to misclassify graph G_i, which can be explained as follows:

$$\max_{G^*} Loss^*$$
$$Loss^* = 1/N \sum_{i=1}^{N} Loss(f(G_i^*), y_i) \tag{3}$$
$$G_i^* = G_i + \alpha * \xi^{adv}(G_i),$$

where ξ is a function responsible for creating a small perturbation in order to misclassify the graph. Therefore, we need to construct G^* in order to maximize the loss function.

Because this theoretical model is generative, it targets not only the node embedding based system but also those systems that take the whole graph/subgraph as feature embedding. On the other hand, for other graph-based detection systems [2], which could not allow deleting nodes and edges from graph/subgraph, the above model is also suitable. The methods above demonstrate how we can maximize the loss function on the graph. Although this paper focuses on Adagio, our attacking approaches can easily be extended to other models as well.

3.2 Adversarial samples crafting on graph

N-strongest Nodes. Hugo et al. [7] present a method to construct a relevance map by shading each node in the graph with the sum of the weights of the neighborhood to which it belongs. As a consequence, they find the node with maximum weight values, which has the most significant influence on the neighborhood nodes. However, in our work in order to successfully misclassify a malicious application to a benign one (changing Y from a positive to a negative value), we need to find the node(s) with the minimal weights and insert these nodes multiple times in the graph until $Y < 0$. We define this method as the n-strongest nodes. The strongest node in our work means the node, the weight value of which is minimal.

The initialization process is a step to prepare the weight and find the node(s) which have the minimum weight value. We get the weight value from the machine learning system through training and testing. With these results, we obtain n different nodes directly through n minimal weight values. We treat these nodes with minimal weight values as the n-strongest nodes. For instance, after malware detection, we get the five strongest nodes, which are $[4116, 0, 4110, 12755, 13314]$) with histogram representation.

After the initialization step, we need to pick up each malware sample to craft its benign variant in order to be misclassified by the detection system. To do so, we calculate the number of nodes in the original graph and set the number of nodes that can be injected into the original graph. This injected number is changed by the percentage of total nodes in the graph with α. The misclassification success rate is different due to the variety of this value.

The injection operation is a crucial step in the n-strongest nodes. As described in the detection system in Section 2.3, there are several steps ($X = S\_Bin(S\_Hist(S\_Hash(embed(V,E))))$) from original graph $G(V,E)$ to X as the input of machine learning network, including call graph extraction, node embedding, neighborhoods hashing, feature embedding histogram, and histogram extension. We construct our attacking model after neighborhood hashing.

The process of constructing adversarial examples can be summarized as follows. First, we transform the function representation of nodes in the graph to 15-bit (15 reduced instructions in function embedding) boolean representation. Then, we select nodes from the n-strongest nodes list and map these nodes to the boolean representation of the labeling function (the 15-bit representation of a node is labeling) to attach our strongest nodes. Following the design of Adagio, we do not consider the graph's edge embedding in our design. We only inject the corresponding nodes into hashing values, which consider the current node and neighborhoods' node embedding. After this step, we carry out the process of feature embeddings and transform the histogram H to a $P-$dimensional vector, which is further fed into the machine learning system to classify the apps as malware or benign. In our work, we demonstrate n strongest nodes injection with different numbers of n strongest nodes (from 1 to 5) in order to decrease the total injected nodes' ratio. This could be extended to scenarios with more than five injected nodes.

In the last step, especially for the n-strongest node injection, we design different attacking strategies for detection models with histogram and non-histogram extension. As mentioned in the Section 2.3, histogram extension distributes a node contribution in a graph to thousands of subcomponents. As a consequence, it is more

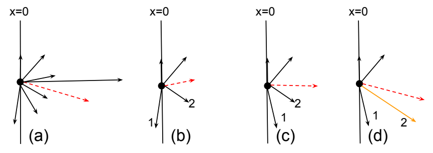


Figure 2: Gradient Mapping

challenging to craft adversarial examples on thousands of subcomponents than on one component. Also, the histogram extension has a significant influence on the misclassification rate. This is why we need comparable designs for the same attacking method.

Gradient-Based Approach. We use the Android apps ecosystem to evaluate our framework. To do so, we manipulate the graph of Android apps G to change malicious apps' graph G into benign ones G' with gradient sign method. To successfully obtain the graph of the malicious apps, we have the following requirements:

R1: In a histogram extension scenario, we cannot include any value less than zero in the P-dimensional vector because these values stand for the occurrence of the corresponding nodes in a graph. We cannot express some nodes appearing less than zero time. For example, a node N_{250} occurring 5 times is represented as $[1, 1, 1, 1, 1]_{250}$. We cannot represent it as $[1, 1, 1, 1, -1]_{250}$.

R2: With histogram extension representation, all "1"s—the number of "1" stands for the occurrence times of node—should align at the beginning of the P-dimensional vector. For example, a node N_{250} occurring 5 times is represented as $[1, 1, 1, 1, 1]_{250}$. We cannot represent it as $[1, 1, 1, 1, 0, 0, 1]_{250}$.

R3: Both for the histogram extension and the non-histogram extension scenarios, we cannot reduce the occurrences of the original nodes number from the original graph. For example, for the node N_{250} occurring 5 times, we cannot represent it with 4 (non-histogram extension) and $[1, 1, 1, 1, 0]_{250}$ (histogram extension).

In the following, we state how these requirements have influenced our design decisions.

Gradient Computation: Our attack aims to increase the number of the misclassified malicious samples (i.e., it decreases the malware classification confidence or increases the misclassification rate). This process can be defined as one constrained optimization problem. Figure 1 illustrates the aforementioned statement. Our work includes gradient calculation and node mapping (projection).

First, we present the gradient calculation for scenarios of histogram extension. To begin with, we use the gradient sign method to craft the manipulated graph roughly. We present the Equation 4 as a constrained optimization problem.

$$\begin{gathered}\min_{n} f(G_h)\\ s.t. d(G_h, G'_h) \leq m,\end{gathered} \tag{4}$$

where G_h is a P-dimensional vector and stands for the whole graph embedding, presented with Equation 5 in Adagio [7]; while G'_h is the manipulated graph embedding. We solve this optimization problem with the gradient sign method through the following squared loss expression as our objective function:

$$f(X) = (X * W + B - Y)^2, \tag{5}$$

where X is the node representation with its number of occurrences (non-histogram extension) or P-dimensional vector (histogram extension), B presents the offset, and Y is the label. Due to the non-differentiability of the objective function, we define a direct vector that has the same direction with the gradient in order to get the gradient of error function $\nabla f(X)/X$. Similarly, since each graphical representation does not include all node components in the histogram representation, we cannot directly utilize the whole vector representation of gradient direction (getting from the trained weight values) in our work.

Here, we introduce a simplified representation of the gradient direction. Figure 2 (a) shows a gradient vector (red dashed line arrow) with seven components (black arrows). Meanwhile, Figure 1 (b) presents a vector with only four components. As a consequence, the gradient is changed (i.e., red dashed line). On the other hand, this representation also has a negative component (i.e., line 1 in Figure 1 (b)), which is not allowed in our graph embedding mechanism because each component stands for the number of occurrences of the node. Therefore, in order to keep the same gradient direction, we need to delete or transform this negative component (i.e., line 1 on the left side in Figure 1 (b)) to positive forms. In our work, we change the negative component to the inserted positive component (change line 1 in Figure 1 (b) to line 1 Figure 1 (c)).

In the last step, we extend the length of each component—the length of line 2 in Figure 1 (d) is two times that of line 2 in Figure 1 (c)—in order to get the same direction with the gradient (i.e., the red dashed line in Figure 1 (a) and red dashed line Figure 1 (d) are parallel). After we get the direction of the simplified gradient, we can use the gradient sign to craft the manipulated graph, which has no negative components due to the gradient mapping scheme.

With this step, we get the manipulated P-dimensional vector by the gradient sign method. Again, taking N_{250} as an example, the original representation is: $[1, 1, 1, 1, 1]_{250}$ and after manipulating by gradient sign method, it is changed to:

$$[1, 2, 1, 1, -1, 0, 0, -1, 2, -4, \ldots, 1, -4, 2, \ldots]_{250}$$

This representation cannot meet the requirements $R1$ and $R2$. Thus, we need an additional operation to get the legitimate P-dimensional vector of function call graph. We will discuss this later in the paper.

Gradient calculation under the non-histogram extension is a straight-forward issue compared to gradient calculation under the histogram extension. We can get the gradient in the same manner as in the histogram extension. However, we do not need to construct the reduced gradient direction as in the histogram extension. We manipulate the node value (the number of occurrences) directly on the graph with the histogram representation like the pixels in an image. For example, we have nodes in the graph with histogram representation $[\ldots, 52, 34, 98, \ldots]$. After gradient-based manipulation, this representation changes to:

$$[\ldots, 53.5, 32.5, 99.5, \ldots, \ldots, \ldots]$$

with $\alpha = 0.5$, and the sign value of the first and third elements are 1 and the second is -1. So far, we acquire the manipulated histogram representation, but we cannot directly craft an adversarial example on it because it cannot meet $R3$. Therefore, we need to perform node projection, which is the same as the histogram extension.

Node Projection: With the gradient sign method, we get the manipulated graph with the form of feature embedding. Because of the gradient sign operation ($\mathbf{X'} = \mathbf{X} + \text{sign}(\mathbf{r\_direct})$), in the histogram extension, we get the values of some nodes of more than 1; meanwhile, others are less than 0, which indicates the number of occurrence of the node.

Therefore, we need to operate the graph generated by the gradient sign method in order to meet these requirements ($R1$ and $R2$). First, we iterate all nodes in the graph and discover all nodes that appear more than one time, which means that the P-dimensional vector of the node has at least one sub-components ($V = [1, \ldots]$). This way, we narrow the injected node as much as possible by eliminating the never appeared nodes in the graph. In other words, we do not introduce new nodes into the original graph. Next, for the nodes appearing more than one time, we divide manipulated sub-components into two groups: the negative and the positive group. For elements in the negative group, we reassign "1" replacing it with a negative value, which means this sub-component appears one time. By this operation, we meet the $R1$ requirement. For the elements in the positive group, we reduce these manipulated values (by gradient sign method) to "1"s as well. This means that this sub-component can only appear one time. With these steps we can change the manipulated P-dimensional representation:

$$[1, 2, 1, 1, -1, 0, 0, -1, 2, -4, \ldots, 1, -4, 2, \ldots]_{250}$$

to a new one:

$$[1, 1, 1, 1, 1, 0, 0, 1, 1, 1, \ldots, 1, 1, 1, \ldots]_{250}$$

So far, the new representation cannot meet the $R2$ requirement because these "1"s appear not only at the beginning of the vector but also in the middle and at the end. In order to meet $R2$, we need to reduce the number of these "1"s, which indicates the appearance of sub-components. In other words, this means that these "1"s should be aggregated at the beginning of the P-dimensional vector.

The last issue is how many sub-components should be kept in the P-dimensional vector, which is heavily influenced by the number of injected nodes and also the misclassification rate. We adjust the injected sub-components number from 1 to 5 to measure our results. In theory, it could be extended as much as possible until the largest number of sub-components. Finally, we can get the final representation of vector P as: $[\mathbf{1,1,1,1,1}, \mathit{1,1,1,1,1}, 0, 0, 0, \ldots]_{250}$. In this representation, the first five "**1**"s stand for the original number of occurrences. The second five "1"s stand for the adjusted occurrence times (vary from 1 to 5) by projection operation. These rest sub-components will be "0"s.

For the issue of node projection in the non-histogram extension, we need to operate the aforementioned manipulated graph in the previous section in order to meet $R3$. Therefore, we design a two-step node projection scheme. First, we need to keep the number of nodes in the original graph. For example, we change the manipulated vector (graph's vector representation) from:

$$[\ldots, 53.5, 32.5, 99.5, \ldots, \ldots]$$

to:

$$[\ldots, 53.5, 34, 99.5, \ldots]$$

by comparing the corresponding values in the original graph. Second, we use the ceiling operation to get the number of injected nodes because we cannot inject nodes with half. Then, we get the new vector: $[\ldots, 54, 34, 100, \ldots]$. We also set a *threshold* value to limit the number of injected nodes. The value of *threshold* will influence the value of the misclassified rate and injected node ratio, demonstrated in Section 4. Finally, we compare all manipulated input values with the *threshold* and accomplish the node projection.

Random Node Injection. This is the simplest attacking model which randomly selects the nodes and inject them into the graph. In our work, we only select nodes and not edges in the graph since we modify our graph structure after neighbors' nodes hashing operation, which has already considered the node and edge connections. Although this attacking method is not expensive, the misclassification is not very significant. We will compare these results with the approaches above in Section 4.

3.3 Adversarial Android Apk examples

So far, we have discussed how to get the adversarial examples based on a graph structure. However, if we want to evade the malware detection system automatically, at the end of our processing, we need to construct adversarial Apk examples from the modified graph. In our work, we consider the dead-area to store the modified graph nodes, which means the injected nodes, which stand for the functions in Apk files, cannot be invoked forever. However, for the targeting malware detection system, the modified malware samples cannot be detected by it. Meanwhile, the functionality of the application should remain the same. In this work, we use the Android repackaging method for accomplishing our purposes.

4 EVALUATION

Our attacking methods are crafted in a framework, called MANIS, and evaluated on large datasets of real Android applications. First, we show how do we prepare the dataset we used in our work to evaluate our attacking methods. Then, we proceed with the evaluation of the n-strongest nodes attacking approach. For the evaluation of the other attacks, we state our results as follows. We measure our attacking approaches not only from the success rate of the misclassification but also from the number of inserted nodes and the relationship between the number of inserted nodes and the successfully misclassified rate. Finally, to illustrate the efficiency of our crafting adversarial examples, we testify with the randomly selected nodes from the function call graph in order to evade the detection system.

4.1 Dataset

The dataset consists of 49,947 benign and 5,560 malicious Android applications obtained from the AndroZoo [1] and the Drebin project [4] respectively. To decide whether an application from AndroZoo is benign or malicious, we leverage VirusTotal [20]. Overall, we got 49,947 benign from the 100,288 samples we tested.

To extract the function call graph from Android's APK/DEX files and feed these graphs to the machine learning detection system after several graph operation steps, we utilized the *Androguard* framework [6]. Finally, we obtained the modified function call graph as the target of adversarial examples crafting.

We tested our work with a small and a big dataset. For the small, we selected 1000 out of the 49,947 benign samples and 415

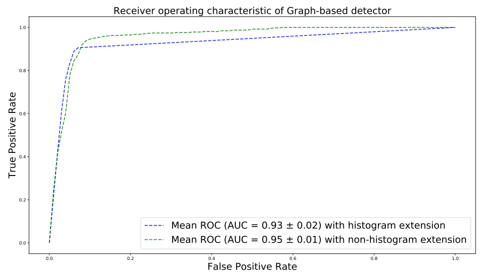


Figure 3: Histogram and non-histogram extension accuracy

malicious ones. For the big dataset, we utilized all the 49,947 benign and 5560 malicious samples. The final results came from both of them. Additionally, we split our dataset into five folds. We used the four folds to train and the last one to test our model.

4.2 White-box and Gray-box Attacks

In white-box attacks, we assume the attackers are allowed to access any information of the target malware classifier, including the weights, offsets, and labels. Meanwhile, we define gray-box attacks as the ones that the attackers can access limited information about the target classifier. However, attackers can access other classifiers (weights, offsets as well as labels) that are trained by other folds. For the training and testing data, in both white-box and gray-box environments, attackers can retrieve all of them.

4.3 Histogram and Non-Histogram Extension

The histogram extension that has presented in Section 2.3 (S_Bin in $X = S\_Bin(S\_Hist(S\_Hash(embed(V, E))))$, maps each histogram representation H to a $P-$dimensional ($P = M * N$, *M is the maximum value of all bins in the dataset, N is the number of bins in each histogram*) vector $\phi(H)$—Formula 5 in [7]) as the last step of data preprocessing. In our work, we delved into this step and find that it affects the misclassified rate drastically; however, the accuracy of the machine learning detection system does not change too much. Therefore, we designed a non-histogram extension and compared the accuracy rate of the detection system with histogram and non-histogram extension. Figure 3 displays the results with histogram and non-histogram extension. It illustrates the efficiency and drawback with and without the non-histogram extension. If we take out histogram extension step from the original Adagio (without our modifications), we obtain 0.95 accuracy on average. Meanwhile, under the same setting and dataset, with the histogram extension, we get an accuracy of 0.93 on average. However, other characteristics (misclassified rate and node injection ratio) of the two models are extremely different. We will discuss these differences below.

4.4 N-Strongest Nodes

We now present the results of the n-strongest nodes attacking models from both the non-histogram and histogram extension models. Table 1 shows that only injecting the 22.7% nodes (one strongest node) into a function call graph will cause the trained classifier to misclassify 72.2% of malicious apps. Meanwhile, if we utilize the five strongest nodes and inject these nodes into the original graphs, we get the 40.8% misclassified rate with the 25.7% node injection.

Table 1: N-strongest nodes (non-histogram extension)

	Non-histogram extension(white-box)					Non-histogram extension(gray-box)				
Strongest nodes	1	2	3	4	5	1	2	3	4	5
Injected nodes ratio ($\overline{x}$)	22.7%	23.7%	17.6%	15.6%	25.7%	20.69%	21.25%	32.26%	17.76%	18.98%
Injected nodes ratio (σ)	5.8%	22.3%	22.7%	22.1%	22.4%	5.19%	20.67%	27.93%	30.76%	32.87%
Misclassified Rate ($\overline{x}$)	72.2%	26.7%	26.6%	32.8%	40.8%	80.79%	49.43%	35.62%	21.14%	19.8%
Misclassified Rate (σ)	15.4%	28.4%	34.1%	43.7%	42.5%	4.9%	42.9%	25.9%	25.6%	26.8%

Table 2: N-strongest nodes (histogram extension)

	histogram extension (white-box)					histogram extension(gray-box)				
Strongest nodes	1	2	3	4	5	1	2	3	4	5
Injected nodes ratio ($\overline{x}$)	24.30%	37.27%	18.51%	46.95%	40.24%	23.68%	22.38%	14.41%	25.88%	23.35%
Injected nodes ratio (σ)	8.6%	8.2%	10.4%	11.3%	19.1%	24.94%	17.04%	13.17%	15.13%	14.52%
Misclassified Rate ($\overline{x}$)	6.01%	29.77%	6.97%	35.81%	17.65%	5.46%	21.33%	4.39%	4.70%	5.03%
Misclassified Rate (σ)	1.89 %	22%	3.3%	27%	14%	3.71%	29.31%	1.21%	0.82%	0.77%

With one strongest node, we get the highest misclassified rate with the lowest injected ratio because that node has the strongest (minimal weights) influence for moving Y from $Y > 0$ to $Y < 0$. However, for the remaining situations, even if we inject more nodes into graph than the former one, we acquire lower misclassified rates. The reason is that the one first strongest node gives the highest contribution to move Y from $Y > 0$ to $Y < 0$, but the second (third, fourth, fifth) nodes will lower the misclassified rate because they have positive values to affect Y value oppositely. Even worse, more injected nodes mean a higher injected ratio. Additionally, from the standpoint of standard deviation, we also get the smallest values (0.154 for misclassified rate and 0.058 for injected node ratio) with one strongest node injection.

In contrast to the n-strongest nodes without the histogram extension, we obtain a minimal misclassification rate compared with the n-strongest nodes method with histogram extension. For one strongest node, we only get a 6.01% misclassified rate with 24.40% node injection. Although for two strongest nodes and four strongest nodes, we get a higher misclassified rate, the standard derivations of them are also very high, which means that these data are not stable. The original reason for this phenomenon is that the process of the histogram extension introduces many subcomponents replacing with one component. This means one node contribution can be divided into multiple small contributions from many subcomponents. Therefore, even if there are several ways to create a small perturbation for each node, the probability of misclassifying malware is very little. The reason is that if we distribute this small perturbation to thousands of subcomponents (M in vector P, in our test, we get the value of M with 8,910 and 17,652 for two datasets), and inject the n-strongest node multiple times, the contribution of perturbation is $n * 1/8910$ (or $n * 1/17652$), n is the number of injected nodes, and the final value of this perturbation is also very tiny. However, from the adversarial example crafting point of view, we can craft malicious samples successfully even if the detection system has robust characteristics with histogram extension.

When we examine the Table 1 and Table 2, we discover that both of them get high standard deviation values. Strictly speaking, the n-strongest node method does not work very well on graph-structured data. In order to compare results with another method, we constrain our result in Figure 4 only with the values less than the average standard deviation from the tables above. By contrasting Table 1 with Table 2, we find that histogram extension can improve the robustness under the influence of adversarial examples (except

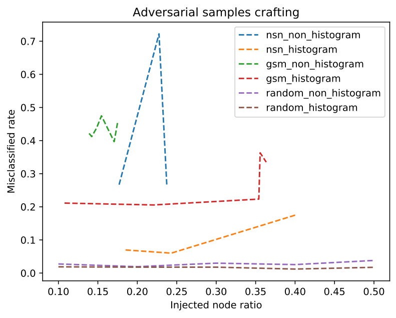


Figure 4: Misclassified rate with n-strongest node, gradient sign method and random selection (white-box)

the two and four strongest nodes injection since these results have high standard deviation values).

Under the gray-box attack, we also retrieve positive results, both with the histogram and the non-histogram extension.

4.5 Gradient Sign Method

As described in Section 3, to adjust the direction of the reduced direct vector, we need to prolong the length (occurrence of the same node) of existing nodes. We accomplish this step with the process of node insertion. We show the misclassified rate and inserted nodes ratio with the gradient sign method in Table 3 and Table 4.

For the non-histogram extension, we use different α and *threshold* values to evaluate our gradient sign method. We set three different *threshold* values: $threshold < \alpha$, $threshold <= \alpha$, and $threshold <= 2 * \alpha$. We illustrate the efficiency of our method in Table 3. With $threshold < \alpha$, we can get very a high misclassified rate with the cost of 50 times that of injected nodes. Besides that, we get 47.46% as the highest misclassified rate with only 15.46% node injection in $threshold <= \alpha$ and 39.68% as lowest misclassified rate with 17.06% node injection with $threshold <= 2 * \alpha$ configuration.

For the histogram extension, we also test the different number of adjusted occurrence of a node described in Section 3.2. We present our histogram extension results in the Table 4. The misclassified rates change with the different ratios of node injection ($\alpha = 1$). We acquire the highest misclassified rate of 33.49% with the 36.3% node injection when we keep five nodes at the node projection step. Meanwhile, we only get a 20.56% misclassified rate with the 22.05% node injection keeping two nodes during node projection.

One phenomenon from Table 4 is that the results follow the rule of the more node injection, the higher the misclassified rate, except for the first column. For the first column, we only inject 10.47% of the nodes by keeping one node at the node projection, which means we keep the number of nodes with a positive weight value as small as possible. For the rest of the cases, we are not only injecting more nodes with negative weight values (positive influence for moving Y from $Y > 0$ to $Y < 0$), but also nodes with positive values.

Under the gray-box attack, with the gradient sign method, we get a significant misclassified rate with a similar injection ratio of the white-box attack. Just one strange aspect is that with histogram extension, even if we inject more nodes into the graph, the misclassified rate does not change. The reason is that we take a reduced

Table 3: Gradient sign method (non-histogram extension)

α	0.1			0.2			0.3		
threshold	< 0.1	<= 0.1	<= 0.2	< 0.2	<= 0.2	<= 0.4	< 0.3	<= 0.3	<= 0.6
	non-histogram extension (white-box)								
Injected nodes ratio ($\overline{x}$)	56.5x	17.48%	17.06%	51.8x	15.46%	14.82%	46.9x	14.22%	13.89%
Injected nodes ratio (σ)	1.98	1.1%	1.1 %	2.74	1%	0.6%	59.74 %	1.4%	0.5%
Misclassified rate ($\overline{x}$)	79.6%	45.3%	39.68%	94.21%	47.46%	43.69%	98.07%	41.20%	42.17%
Misclassified rate (σ)	5.2%	5.6%	1.9%	0.6%	3.7%	6.7%	2.6%	7%	0.2%
	non-histogram extension (gray-box)								
Injected nodes ratio ($\overline{x}$)	59.62x	15.61%	15.35%	48.57x	14.72%	14.7%	50.09x	15.07%	15.26%
Injected nodes ratio (σ)	4.71	1.18%	0.78 %	3.48	0.99%	0.98%	1.89	1.55%	1.86%
Misclassified rate ($\overline{x}$)	79.76%	37.59%	38.95%	96.06%	43.69%	44.18%	97.43%	44.65%	43.05%
Misclassified rate (σ)	3.78%	7.95%	8.29%	2.08%	8.11%	8.23%	1.33%	8.43%	8.07%

Table 4: Gradient sign method (histogram extension)

	histogram extension (white-box)					histogram extension(gray-box)				
N	1	2	3	4	5	1	2	3	4	5
Injected nodes ratio ($\overline{x}$)	10.47%	22.05%	35.40%	35.42%	36.3%	9.52%	21.01%	33.11%	33.11%	33.11%
Injected nodes ratio (σ)	0.04%	2.23%	0.22%	1.49%	2.04%	0.4%	0.8%	1.05%	1.05%	1.05%
Misclassified rate ($\overline{x}$)	21.12%	20.56%	22.3%	36.30%	33.49%	18.79%	18.79%	18.79%	18.79%	18.79%
Misclassified rate (σ)	5.8%	6.52%	6.63%	8.21%	11.93%	2.04%	2.04%	2.04%	2.04%	2.04%

Table 5: Randomization method.

	non-histogram extension (white-box)					histogram extension(gray-box)				
Injected nodes ratio ($\overline{x}$)	10%	20%	30%	40%	50%	10%	20%	30%	40%	50%
Misclassified rate ($\overline{x}$)	2.74%	1.92%	2.98%	2.55%	3.81%	1.87%	1.78%	1.78%	1.2%	1.73%
Misclassified rate (σ)	0.93%	0.85%	1.5%	1.09%	1.83%	1.31%	0.99%	1.25%	0.73%	1.13%

gradient direction, obtained from the trained model. We conclude that the gray-box attack influences the misclassified rate significantly with histogram extension. For non-histogram extension, the gray-box attack does not influence too much.

The misclassification rate with our gradient sign with histogram extension is not changed consistently by increasing the node injection due to the robust characteristic on graph structure and the usage of histogram extension, which is different from most of the other adversarial example crafting with the gradient sign method. As shown in Figure 3, the success rate of our method with the histogram extension is only changed linearly until 35% node injection but stayed almost stable despite injecting more nodes. While in most other gradient sign methods, the misclassification rate grows significantly with the increase of injected nodes, which is the same as our method with the non-histogram. For instance, Kolosnjaji et al. [11] demonstrate that the misclassification rate of a real Windows binary (PE file) is changing fast with the increase of injected nodes. With our gradient sign method under the histogram extension scenarios, we only get the misclassification rate of 33.49%. However, for the non-histogram extension method, we also get a significant misclassification rate with the increase of injected nodes. In summary, the histogram extension is an excellent way to improve the robustness under the influence of adversarial examples.

4.6 Randomization Method

Finally, we considered the random node insertion to demonstrate the efficiency of our n-strongest node approach and gradient sign method. In contrast to the aforementioned two schemes, we selected randomly the nodes from function call graphs after label hashing.

To compare the efficiency of our crafting method, we injected randomly selected nodes with 10%, 20%, 30%, and 40% ratio. We evaluated those nodes both for the non-histogram and histogram extension. We show our result in Table 5. More precisely, it demonstrates that the misclassified rate is meager with the randomization method; even if we inject 50% new nodes into the graph, we only get a 3.81% misclassification rate. On the other hand, we discovered that the histogram extension has better robustness compared to the non-histogram one under the influence of adversarial examples.

We summarize our attacking models in Figure 4. We illustrate the significance of our n-strongest nodes method (NSN) and gradient sign method (GSM). To compare all results with a $0-1$ node injection ratio, we eliminate the injected node ratio larger than one from Table 3. Figure 4 shows the functionality of histogram extension and its robustness with adversarial examples. With the non-histogram extension, both NSN and GSM get a high misclassification rate with a small node injection ratio. With histogram, the GSM gets a stable misclassification rate, and it peaks when injected 35% nodes. For NSN, we get very small misclassification. GSM works more stable than NSN with both histogram and non-histogram. For the random node selection method, we get minimal misclassification rates.

5 RELATED WORK

Many works in various fields have successfully crafted adversarial examples. Cleverhans [16] summarizes many approaches to craft the adversarial examples. Among the methods for evading the existing machine learning models, the fast gradient sign method (FGSM) [8], a gradient-based method, is a simple approach to misclassify the images. Meanwhile, the Jacobian Saliency Map Approach (JSMA) [17] presents forward derivative and adversarial saliency maps to attack the deep neural networks.

Successfully attacking a malware detection system is not trivial. Kolosnjaji et al. [11] propose a gradient-based attacking method, which can evade deep neural networks by changing only a few specific bytes at the end of each malware sample while the original functionality remains intact. Anderson et al. [3] provide a solution to attack a recurrent neural network with the discrete characteristics of the text from IMDB movie reviews as input samples. However, none of the previous methods do consider the graph structure as input; thus, we cannot directly use them in our evading method. Dai et al. [5] focus on the adversarial attacks that mislead deep learning models by modifying the combined structure of data, while Zügner et al. [24] introduce a study of adversarial attacks on attributed graphs, focusing on models exploiting graph convolutions.

6 CONCLUSION

In this paper, we present two methods to craft adversarial examples to evade machine learning detection systems: n-strongest nodes and gradient sign method. Both solutions demonstrate that graph-based machine-learning malware detection approaches are vulnerable to adversarial samples. Especially with the n-strongest nodes, we show how to craft the adversarial examples with multiple insertions of nodes with minimal weight values. The gradient sign method (GSM) is an effective way to craft adversarial examples on the graph structure. We have also proved the effectiveness of GSM by solving multiple problems in practice, such as non-differentiable objective function, indirection mapping from a graph to feature, and node embedding with graph hashing. Overall, this paper shows that it is possible to evade machine learning systems through adversarial example crafting, which takes whole graph structures as inputs.

ACKNOWLEDGMENTS

This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 833115 (PREVISION)

REFERENCES

[1] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting Millions of Android Apps for the Research Community. In *IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR).*

[2] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. 2011. Graph-Based Malware Detection Using Dynamic Analysis. *Journal in computer Virology* 7, 4 (2011), 247–258.

[3] Mark Anderson, Andrew Bartolo, and Pulkit Tandon. 2017. Crafting Adversarial Attacks on Recurrent Neural Networks. (2017).

[4] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Network and Distributed System Security Symposium (NDSS).*

[5] Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. 2018. Adversarial Attack on Graph Structured Data. In *International Conference on Machine Learning (ICML).*

[6] Anthony Desnos et al. [n.d.]. Androguard: Reverse Engineering, Malware and Goodware Analysis of Android Applications. https://github.com/androguard.

[7] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2013. Structural Detection of Android Malware Using Embedded Call Graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security.*

[8] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. [n.d.]. Explaining and Harnessing Adversarial Examples (2014). *arXiv preprint arXiv:1412.6572* ([n. d.]).

[9] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems.*

[10] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. 2018. Manipulating Machine Learning: Poisoning Attacks and Countermeasures for Regression Learning. *arXiv preprint arXiv:1804.00308* (2018).

[11] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. 2018. Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables. *arXiv preprint arXiv:1803.04173* (2018).

[12] Bojan Kolosnjaji, Apostolis Zarras, Tamas Lengyel, George Webster, and Claudia Eckert. 2016. Adaptive Semantics-Aware Malware Classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.*

[13] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. 2016. Deep Learning for Classification of Malware System Call Sequences. In *Australasian Joint Conference on Artificial Intelligence (AI).*

[14] Pei-Hsuan Lu, Pin-Yu Chen, Kang-Cheng Chen, and Chia-Mu Yu. 2018. On the Limitation of Magnet Defense Against L1-Based Adversarial Examples. In *International Conference on Dependable Systems and Networks Workshops.*

[15] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. Deepfool: A Simple and Accurate Method to Fool Deep Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.*

[16] Nicolas Papernot, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Fartash Faghri, Alexander Matyasko, Karen Hambardzumyan, Yi-Lin Juang, Alexey Kurakin, Ryan Sheatsley, et al. 2016. Cleverhans V2. 0.0: An Adversarial Machine Learning Library. *arXiv preprint arXiv:1610.00768* (2016).

[17] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. In *IEEE European Symposium on Security and Privacy (EuroS&P).*

[18] Naser Peiravian and Xingquan Zhu. 2013. Machine Learning for Android Malware Detection Using Permission and Api Calls. In *IEEE International Conference on Tools with Artificial Intelligence (ICTAI).*

[19] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. 2017. Convolutional Neural Networks Over Control Flow Graphs for Software Defect Prediction. In *IEEE International Conference on Tools with Artificial Intelligence (ICTAI).*

[20] Virus Total. 2019. VirusTotal-Free Online Virus, Malware and URL Scanner. *Online: https://www.virustotal.com/* (2019).

[21] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. 2017. Ensemble Adversarial Training: Attacks and Defenses. *arXiv preprint arXiv:1705.07204* (2017).

[22] Huang Xiao, Battista Biggio, Blaine Nelson, Han Xiao, Claudia Eckert, and Fabio Roli. 2015. Support Vector Machines Under Adversarial Label Contamination. *Neurocomputing* 160 (2015), 53–62.

[23] Yujie Yuan, Lihua Xu, Xusheng Xiao, Andy Podgurski, and Huibiao Zhu. 2017. RunDroid: Recovering Execution Call Graphs for Android Applications. In *Joint*

Meeting on Foundations of Software Engineering.

[24] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. 2018. Adversarial Attacks on Neural Networks for Graph Data. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.