Semantics-Preserving Node Injection Attacks Against GNN-based ACFG Malware Classifiers

Dylan Zapzalka, Saeed Salem, and David Mohaisen

Abstract—To increase security for devices connected to the internet, research has gone into using Graph Neural Networks (GNNs) to inhibit the spread of malware through detection. GNN classifiers that use Attributed Control Flow Graphs (ACFGs) have demonstrated favorable results in classifying software binaries as malicious or benign. In this work, we show that such classifiers are vulnerable to Adversarial Examples (AEs) by proposing several grey-box adversarial attacks that perform node injection and preserve the semantics of a program. We demonstrate that adversaries can take advantage of the aggregation properties of GNNs to apply effective perturbation outside of the original ACFG nodes of a software binary through node injection. We conducted experiments on our methods and compared them against two similar semantics-preserving adversarial attacks. Our results have shown that our methods of applying perturbation through node injection can result in higher evasion rates while decreasing the amount of perturbation needed to fool detectors. Namely, we deliver an evasion rate of up to 94.83% with only 2.49% of total perturbation, in comparison with a maximum evasion of 79.50% at 2.78% perturbation by a state-of-the-art approach and only 27.44% at 2.95% perturbation by the baseline attack. Our results highlight the need for creating more robust GNN malware detectors.

1 Introduction

Malicious software, also known as malware, is one of the most significant threats to users and data security today [1], [2]. As users increasingly move online for social media, shopping, and work, the already high costs associated with malware threats have been rising [3]. For instance, ransomware—a type of malware that blocks access to a computer system by encryption until a ransom is paid—was expected to have cost 20 billion dollars globally in 2021 [4]. To combat malware, researchers have been developing effective deep neural network-based models to inhibit the spread of malware through detection [5]–[14].

One class of neural networks that has recently gained researchers' attention for malware detection is Graph Neural Networks (GNNs). Through the use of static graph features obtained from software binaries, GNNs can adequately capture traditionally used information, such as opcodes [5] and n-grams [6] while capturing the structural information that makes up the program through the graph's topology. GNN models that use Attributed Control Flow Graphs (ACFGs) for malware classification, such as MAGIC [10] and HawkEye [11], produce favorable results with a variety of GNN architectures, including DGCNN [15] and GCN [16]. Moreover, they have demonstrated other advantages, including supporting cross-platform malware detection on multiple systems and architectures [11].

Despite success in classifying malware, GNNs are vulnerable to Adversarial Examples (AEs) [17], samples that are intentionally created to misclassify a model by adding small worst-case perturbations. In the case of malware classification, there is an enormous incentive for malicious actors to generate AEs: one AE that bypasses a malware detection system could lead to stolen sensitive information, loss of access to systems, or damage to critical infrastructure [18].

To better understand the vulnerabilities of deep learning models for malware detection, researchers have proposed numerous algorithms for generating AEs. For instance, Abusnaina *et al.* [19] proposed a method of fooling deep-learning-based malware detectors called GEA, which works by embedding the Control Flow Graph (CFG) of a benign software binary into a malicious software binary. Building upon this idea, SGEA [20] was proposed to reduce perturbations by only injecting a small subgraph into the original sample. Both works, however, are limited to CNN-based malware detectors. Other methods have focused on generating AEs for GNN-based malware detectors. Zhang *et al.* [21] proposed a reinforcement learning algorithm called the Semantics-preserving Reinforcement Learning (SRL) attack that injects semantic `nop` instructions into the original ACFG nodes of a software binary. The semantic `nop` instructions ensure that the functionality of the original software binary is preserved as the injected instructions are executable. Research has also been done in developing GNN AEs that work by changing the graph structure. One such algorithm, called VGAE-MalGAN [22], changes the graph structure by injecting nodes and edges into an existing API graph via a Generative Adversarial Network (GAN) [23]. Although each of these algorithms can produce effective AEs, there currently exists no algorithm that exploits ACFG node injections to create semantics-preserving AEs for GNN-based malware detectors.

In this work, we address this problem by developing realistic, semantics-preserving AEs to fool GNN-based malware classifiers by injecting artificial nodes into an ACFG. To create a semantics-preserving AE for software binaries, we are limited in the type of perturbation we can add directly to the executable section of the AE. For instance, suppose that to generate worst-case perturbation via a gradient-based

- *D. Zapzalka is with the Department of Computer Science at North Dakota State University (e-mail:* ***dylan.zapzalka@ndsu.edu****).*
S. Salem is with the Department of Computer Science and Engineering at Qatar University (e-mail: ***saeed.salem@qu.edu.qa****).*
D. Mohaisen is with the Department of Computer Science at the University of Central Florida (e-mail: ***mohaisen@ucf.edu****).*

method, e.g., the Fast Gradient Sign Method (FGSM) [24], we needed to add an arithmetic instruction to the executable section of the software binary. If we were to naively inject a line of assembly code such as `ADD EAX 7`, we would be changing the semantics of the program, which could lead to the failure of the AE. Thus, when adding perturbation to a part of the program that is executable, we are limited to injecting semantic `nop` instructions, e.g., `ADD EAX 0`. A more serious problem arises when we must take away an instruction to generate an AE. It is clear that removing any executable instruction without an equivalent replacement would change the functionality of the AE. One way of trying to get around this issue would be to remove and replace the instruction with new code that satisfies the requirements of functionality preservation and adding effective perturbation. However, this scenario is difficult at best for an adversary to carry out. Therefore, given the restrictions of preserving the semantics of a program, a method like FGSM can only apply perturbation uni-directionally to the executable part of the software binary.

To solve the problem of adding perturbation, we propose new and efficient grey-box gradient-based methods for generating AEs over the ACFG of a program. Our methods of generating AEs, called Semantics-preserving Node Injection Attack (SNIA), Semantics-preserving Multiple Node Injection Attack (SMNIA), and Semantics-preserving Node Injection Clustering Attack (SNICA), work by exploiting the aggregation steps of GNNs to indirectly apply perturbation to the executable portion of the AE without changing its functionality. To do this, we generate and connect inexecutable artificial nodes to a subset of the executable nodes of the original ACFG. Such a method has a multitude of benefits in contrast to adding perturbation directly to the executable portion of the ACFG. First, since the artificial nodes are not executable, we can add any perturbation to them without changing the functionality of the program. Through this step, the adversary can mimic applying perturbation bi-directionally to the executable portion of the software binary by aggregating perturbation from an inexecutable artificial node. Another benefit of this approach is the reduction of the total amount of perturbation needed to fool a model. Through our approach, one artificial node can connect to multiple different original ACFG nodes to multiply the effect of its perturbation throughout the graph instead of having a similar perturbation added multiple times.

Contributions. In this paper, we make the following contributions. 1) We investigate ways to exploit the aggregation property of GNNs to generate semantics-preserving AEs that follow several defined constraints. 2) We propose novel gradient-based methods of AE generation for ACFG malware classifiers by injecting inexecutable nodes. 3) We evaluate the proposed methods on a real-world dataset and show their effectiveness. To the best of our knowledge, this is the first work that examines semantics-preserving AEs in the context of GNNs exploiting the aggregation step.

Organization. The organization of this paper is as follows. The related work is presented in Section 2. In Section 3, we introduce the background required for understanding the rest of this paper. In Section 4, we present the generation of AEs, including our problem formulation, gradient-based methods for adding perturbations, and our new methods of creating AEs. Our experimental evaluation is presented in Section 5. Discussion and future work are presented in Section 6. Concluding remarks are drawn in Section 7.

2 Related Work

Much research has been conducted on generating AEs through gradient-based techniques. Goodfellow *et al.* [24] proposed the Fast Gradient Sign Method (FGSM) that generates AEs by adding perturbations in the same direction of the gradient of the cost function with respect to the input data. Although applicable with some care to the software domain, their original work was intended in the image domain. Kurakin *et al.* [25] later proposed an iterative version of FGSM that adds perturbations in the direction of the gradient multiple times. Such methods work very well when the target model's architecture and parameters are known, however, an adversary cannot use them if they have no access to the internal details of the model. A method introduced by Papernot *et al.* [26] solves this problem by introducing a substitute model that trains over a synthetic dataset. FGSM can then be applied to the substitute model to generate AEs for the target model.

The utilization of a surrogate model has previously been employed to create AEs GNN malware detection systems [27]. Zhang *et al.* [21] used a substitute model for the Semantics-preserving Gradient based Insertion (SGI) adversarial attack that works by injecting semantic `nop` instructions into ACFGs. Semantic `nop` instructions have also been added to the original nodes of an ACFG through a reinforcement learning algorithm called the Semantics-preserving Reinforcement Learning (SRL) attack [21]. Another gradient-based approach developed by Yumlembam *et al.* [22] uses a Generative Adversarial Network (GAN) algorithm called VGAE-MalGAN to create AEs for GNN malware classifiers that use Android API graphs. VGAE-MalGAN creates AEs by injecting new nodes and edges into an existing API graph using a generator and a substitute detector.

Other methods can be used to create AEs for malware detectors without using the gradient. Abusnaina *et al.* [19] proposed an adversarial attack for CNN-based models that use CFG features called GEA. The attack works by embedding a benign software binary inside of a malicious target software binary such that the benign software binary code is never executed. Absunaina *et al.* [20] later built upon GEA by introducing sub-graph embedding and augmentation (SGEA) to reduce the amount of perturbation required to cause misclassification. As noted earlier, their work is only limited to CNN-based detectors, and it is unclear if their performance will generalize to other architectures.

Previous research has also focused on node injection adversarial attacks for GNNs, which is the most closely related line of work to our work. Tao *et al.* [28] proposed two different node injection attacks that are limited to injecting a single node. The first is an optimization-based approach called OPTI whereas the second attack called Generalizable Node Injection Attack (G-NIA) speeds up the slow optimization process through the use of a parametric model. Another node injection attack called Topological Defective

Graph Injection Attack (TDGIA) by Zou *et al.* [29] works by combining a topological defective edge selection strategy with a smooth feature optimization objective to generate the node-feature vector for the injected node. The tests on TDGIA were limited to leveraging the topology of the first-level neighborhood of a graph, and it is unclear how similar algorithms would perform when considering greater levels of neighborhood information. Recently, Fang *et al.* [30] proposed a node injection strategy called Global Attacks via Node Injections (GANI), which injects nodes while still maintaining structural similarity through a degree sampling operation. Similarly, Chen *et al.* [31] introduced the Harmonious Adversarial Objective (HAO), a constraint that allows AEs to become more unnoticeable to defenses by making sure that maliciously injected nodes preserve homophily.

3 Background

In this section, we review the preliminaries required for understanding the rest of this work. Namely, we review background information on graphs (section 3.1), graph neural networks (section 3.2), control flow graphs generated through software binaries (section 3.3).

3.1 Graphs

A graph, denoted as $G = (V, E)$, is a mathematical object consisting of two finite sets: the vertex set and the edge set. The vertex set $V = \{v_1, v_2, ..., v_n\}$ contains the graphs vertices, also called nodes, and the edge set $E \subseteq \{(v_i, v_j) \in V^2 \text{ and } v_i \neq v_j\}$ contains pairs of distinct vertices referred to as edges. Graphs can also be represented by a matrix called the adjacency matrix. An adjacency matrix A that represents a graph of order n is an $n \times n$ matrix where each entry a_{ji} is 1 if there exists an edge connecting v_i to v_j; otherwise, the entry is equal to 0. Graphs in some circumstances may have node-feature vectors X_v for $v \in V$. These node-feature vectors are stacked to create a node-feature matrix X where the i^{th} row of the matrix represents the feature vector of the i^{th} vertex of the graph.

3.2 Graph Neural Networks

Graph Neural Networks (GNNs) are a class of neural networks used to perform predictions on graph-based data. Given a graph $G = (V, E)$ with a node-feature matrix X, the goal of a GNN is to learn a representation vector h_v for every $v \in V$ to perform either node, edge, or graph classification. We denote the representation vector at layer l for node v as h_v^l and we initialize $h_v^0 = X_v$. Through one or more GNN layers, h_v^l is updated by aggregating its neighbors' node representation vectors as well as h_v^{l-1} itself. After l iterations, the nodes representation vector will have captured the structural information within a neighborhood of radius l [32]. We can define the node representation vector at the l-th layer as

$$h_v^l = \mathsf{AGGREGATE}^l\{h_u^{l-1} : u \in N(v) \cup \{v\}\} \tag{1}$$

where $N(v)$ is the set of neighbors for some node v.

With the final learned representation vectors, we can perform graph classification via a graph representation vector h_G, which can be obtained by pooling every node representation vector of the graph given some pooling function. In other words, $h_G = pool(H)$ where H is the node representation vectors stacked such that row v of H is h_v and $pool$ is some permutation-invariant function. Once h_G is obtained, the goal for graph classification is to pass h_G through some function f such that $f(h_G) = y_G$ where y_G is the label of graph G. The architecture of a typical model that uses GNN layers for graph classification is shown in Fig. 1. We now proceed to define a GNN layer called Graph Convolutional Networks (GCN) and a common GNN architecture for graph classification called Deep Graph Convolutional Neural Networks (DGCNN) that will be used throughout the paper.

3.2.1 Graph Convolutional Networks (GCN)

One of the most popular GNN layers is the GCN layer [16], which is defined by the following aggregation rule:

$$H^{l+1} = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^lW^l).$$

We define $\hat{A} = A + I$ as the adjacency matrix A of the directed graph G with added self-connections represented as the identity matrix I. $\hat{D}$ is the diagonal matrix where $\hat{D}_{ii} = \sum_j \hat{A}_{ij}$. W^l is a layer-specific trainable weight matrix. H^l is defined as the matrix of activation's in the lth layer where $H^0 = X$. Lastly, σ is the activation function.

To make a model for graph classification, the GCN layers would need to be followed by some global pooling layer. The pooling layer used in this paper for the GCN models will be the global mean pooling layer defined as:

$$h_G = \mathsf{Mean}(\{h_v, \forall v \in V\}).$$

For the purpose of this paper, the pooling layer of any GCN model will be followed by a deep neural network (DNN) to perform the final graph label prediction.

3.2.2 Deep Graph Convolutional Neural Network (DGCNN)

DGCNN is a GNN for graph classification [15]. The model in DGCNN is made up of three successive parts: the graph aggregation layers, the SortPooling layer, and a traditional CNN. Using the same notation used to define the GCN layer, we define the DGCNN layer as such:

$$H^{l+1} = \sigma(\hat{D}^{-1}\hat{A}H^lW^l).$$

After all of the graph aggregation layers are finished, they are concatenated such that $H^{1:l} = [H^1, H^2, ..., H^l]$.

After the graph convolutional layers, DGCNN has the SortPooling layer with the purpose of sorting the feature descriptors and transforming the input to unify the graph sizes. SortPooling starts by sorting $H^{1:l}$ in descending order with respect to the value of the last column of $H^{1:l}$. The sorting step is followed by unifying the size of $H^{1:l}$ by appending $k - n$ zeros if $n < k$ or trimming the last $n - k$ rows, where n is the initial amount of rows and k is a user-defined integer. Lastly, once SortPooling has been performed, H is fed through a traditional CNN to produce the final graph prediction.

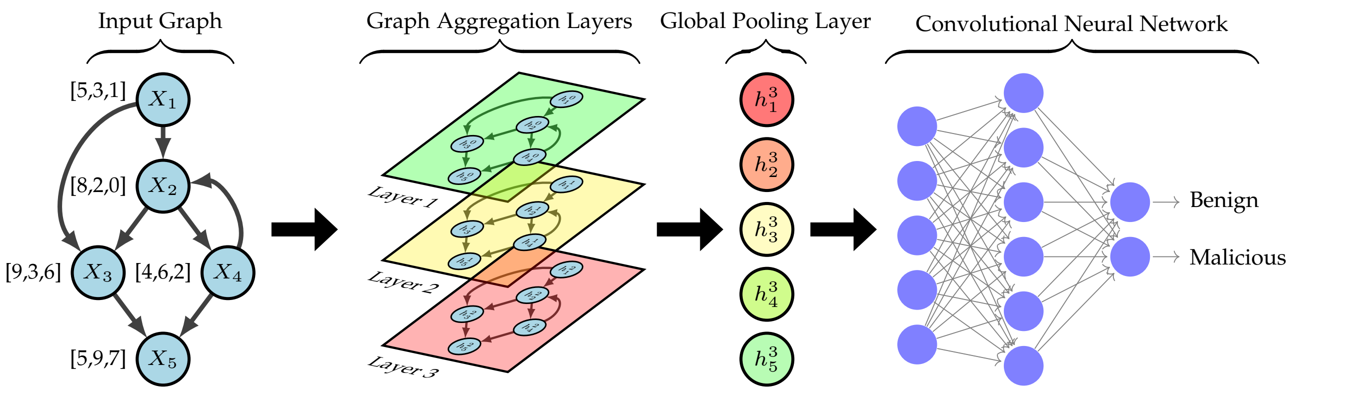

Fig. 1: An example of a typical GNN model used for malware detection (two-class classification), starting with an input graph (e.g., ACFG) that is aggregated, pooled, and convoluted to produce a final graph label.

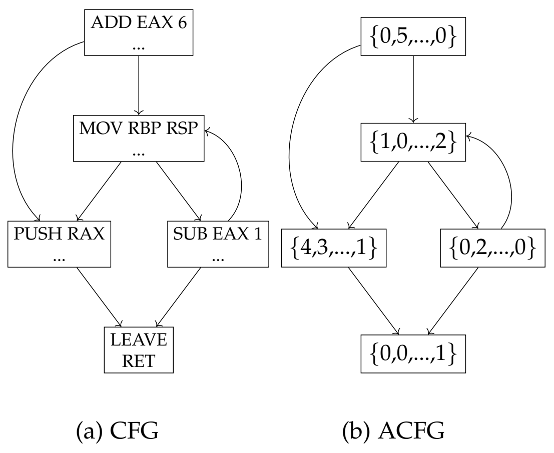

Fig. 2: CFG and ACFG developed from a software binary.

3.3 Control Flow Graphs

We utilize two types of graphs, the Control Flow Graph (CFG) and the Attributed Control Flow Graph (ACFG).

Control Flow Graph (CFG). An illustration of a CFG is shown in Fig. 2a. A CFG is a directed graph $G = (V, E)$ that represents every possible path of execution that can occur in a running program. Each vertex of the graph, $v_i \in V = \{v_1, v_2, \cdots, v_n\}$, represents a contiguous block of code of maximal length, i.e., the first lines of code is the only entry, and the last lines of code represent the possible exits. The set of edges E represents the possible paths of execution of the program such that $(v_i, v_j) = e_k \in E = \{e_1, e_2, \cdots, e_m\}$ if e_k is a jump or branch from v_i to v_j.

Attributed Control Flow Graph (ACFG). An illustration of an ACFG extracted from the CFG in Fig. 2a is shown alongside it in Fig. 2b. An ACFG is a control flow graph with an associated node-feature matrix X of size $n \times m$ where the i^{th} row of the matrix represents the features of the i^{th} vertex, v_i. In this paper, the node features are made up of the opcodes of the program as defined in Section 5. Thus, we represent any ACFG as $G = (A, X)$ where A is the adjacency matrix of the control flow graph and X is its corresponding node-feature matrix.

4 Generating Adversarial Examples

In this section, we formulate generating semantics-preserving AEs and propose effective adversarial attacks that work by injecting inexecutable artificial nodes.

4.1 Problem Formulation

We start by discussing the goals and constraints we must abide by when generating a semantics-preserving AE. Our first objective is to force the target model f to misclassify some sample x with class y by adding perturbation to x. Namely, our first goal is to force $f(x') = y'$ where x' is the AE and y' is some class other than y. Next, our goal is to minimize the amount of perturbation added to x to create x'. That is, $|x - x'| \leq \Delta$ where Δ is the maximum amount of perturbation we allow ourselves to add, and $|x - x'|$ denotes the difference between x and x'. Lastly, we aim to add perturbation only in such a way that x and x' are semantically equivalent. We denote two semantically equivalent samples as $x \equiv x'$. Thus, we can combine the following three objectives to generate our final constraint rule as such: for the original sample x with an associated class label y, we wish to produce an adversarial example x' for some target model f such that:

$$(f(x') \neq y) \wedge (|x - x'| \leq \Delta) \wedge (x \equiv x'). \tag{2}$$

4.2 Threat Model

Background. Adversarial attacks are broadly grouped into three categories based on the amount of knowledge the adversary may have about the target model and associated details: white-box, black-box, and grey-box attacks.

White-box. In a white-box attack, the adversary has complete knowledge of the model's information, such as its parameters and dataset. An example of a white-box attack is FGSM which uses the gradient of the target model to produce AEs [24].

Black-box. The adversary does not know the target model's parameters for a black-box attack. The only information they can derive is through queries of the target model. One way of creating black-box attacks is to use a substitute model [26], where instead of taking the gradient of the target model to generate AEs, the adversary uses the gradient from a substitute model that was trained using a synthetic dataset.

Grey-box. In a grey-box attack, adversaries have levels of information between those provided by black-box and white-box attacks. Like a black-box attack, the adversary still cannot access the target model's parameters. However, the adversary may have other information about the model and its development, e.g., model type, training data, and the features the model uses to make predictions.

Our Model. The adversarial attacks we propose are closest to the grey-box threat model: that is, we assume that the adversary has very limited access and knowledge of the inner workings of the target model. For our constraints, we assume that the adversary does not know the target model's parameters or its training data. The adversary only knows the model's features and that it uses GNN layers. In this case, the target model is a DGCNN that predicts the labels of software binaries given their corresponding ACFG where the node-feature vectors represent the number of different types of opcodes that each node contains. Our target model is based on the MAGIC model developed by Yan *et al.* [10]. Although we only focus on the MAGIC model for this paper, our approach can easily be extended to other ACFG GNN malware detectors [33]. For our experiments, we used the features in Table 1.

4.3 ACFG Perturbations with Gradient-Based Methods

According to the Goodfellow *et al.* [24], we can linearize the cost function J for some model f around the current value of the models parameters θ to obtain an optimal max-norm constrained perturbation of

$$\eta = \varepsilon \cdot \text{sign}(\nabla_X J_f(\theta, X, y)) \tag{3}$$

where X is the models input and y is the label associated with X. For both accuracy and simplification, we set $\varepsilon = 1$. To create an AE with respect to the gradient, we would add η to the current input, i.e.,

$$X := X + \eta. \tag{4}$$

If we wish to update the perturbation over multiple iterations, we can set the initial AE to the original sample such that $X^i = X$ for iteration $i = 0$ and the update step for $i > 0$ can be defined as

$$X^i := X^{i-1} + \eta. \tag{5}$$

Depending on how the adversary wishes to add the perturbation to an ACFG, we will have to change the update step to fit any possible limitations that may arise due to the constraints in equation (2). We will look at two different types of perturbation an adversary can apply: executable perturbation and inexecutable perturbation.

If we are applying a perturbation to the part of the ACFG that is executable, we cannot take away any of the opcodes associated with the node-feature vector. Thus, in this scenario, we would add perturbation with regard to the following equation:

$$X^i := X^{i-1} + \sigma(\text{sign}(\nabla_{X^{i-1}} J_f(\theta, X^{i-1}, A, y))). \tag{6}$$

where σ is the ReLU function.

In contrast, if we're adding perturbation to the part of the ACFG that isn't executable, there isn't a worry about taking away perturbation. However, there is no way to represent a negative amount of assembly instructions, so we must, therefore, make sure that we do not try to take away any instructions that do not exist. Therefore, we would follow the following update rule:

$$X^i := \sigma(X^{i-1} + \text{sign}(\nabla_{X^{i-1}} J_f(\theta, X^{i-1}, A, y))) \tag{7}$$

Next, we must solve the issue of actually obtaining the gradient to update the AEs as is used in equations (6) and (7). Since we do not have access to the model's parameters or training data from our grey-box constraints, we do not have any way of getting the gradient directly from the target model itself. Instead, we must develop a substitute model f' that is used to simulate the target model f using the techniques developed by Papernot *et al.* [26].

To create the substitute model, we assume the adversary has basic knowledge of the target model's architecture, i.e., they know it is some type of GNN that uses ACFGs for malware detection. Therefore, f' will be made as some arbitrary generic GNN that uses ACFGs to perform two-class classification. We also assume that the adversary has basic querying capabilities to generate a synthetic dataset that will be used to train f'. To generate a synthetic dataset, an adversary must first collect a normal dataset comprised of benign and malicious software binaries. The synthetic dataset is then created by querying the target model to generate the labels for each sample in the dataset. The labels provided by the target model are used instead of the original labels so that f' is a close approximation of f [26]. Once the substitute model is trained with the synthetic dataset, the adversary can use it to craft AEs by using the parameters of f' instead of f.

4.4 Semantics-preserving Gradient Insertion (SGI)

Semantics-preserving Gradient Insertion (SGI) is a black-box gradient-based AE generation method developed by Zhang *et al.* [21] that works by injecting semantic `nop` instructions directly into the original nodes of the ACFG. For our slightly modified version, we will classify it as a grey-box attack as we assume the adversary knows the additional information described in the threat model.

To develop an AE using SGI, we use a substitute model to inject `nop` instructions as is shown in Algorithm 1. For each iteration for a maximum number of iterations, we use the gradient to determine which additional `nop` instruction to inject. Given that the perturbation is added to executable portions of the software binary, the perturbation should be updated with respect to equation (6) since we are unable to take away any of the original software binary instructions. However, we modified the equation for Algorithm 1 for purposes of accuracy, with the additional constraint of only allowing a maximum of one instruction to be added for each node-feature vector per iteration. The iterations only stop if the maximum amount of perturbation is reached, if there are no more optimal instructions to add, or if the maximum amount of iterations have been executed.

Algorithm 1 Algorithm of the SGI Attack

```
Input: $G = (A, X)$, $y$, $f'$, $\theta'$, $n_i$, $\Delta$ /* $n_i$: iterations */
Output: $G' = (A, X')$
1: $i \leftarrow 1$
2: $X' \leftarrow X$
3: while $i \leq n_i$ do
4:     $g_i \leftarrow \nabla_{X'} J_{f'}(\theta', X', A, y)$
5: /* Break if no optimal instructions to add */
6:     if $\max(g_i) \leq 0$ then
7:         break
8:     end if
9:     $X^i \leftarrow$ zeros(size($X$))
10:    $X^i_{\arg\max(g_i)} \leftarrow 1$ /* Add optimal instruction */
11:    $X' \leftarrow X' + X^i$
12:    if $|X - X'| \geq \Delta$ then /* Cap the perturbation */
13:        break
14:    end if
15:    $i \leftarrow i + 1$
16: end while
17: $G' \leftarrow (A, X')$
```

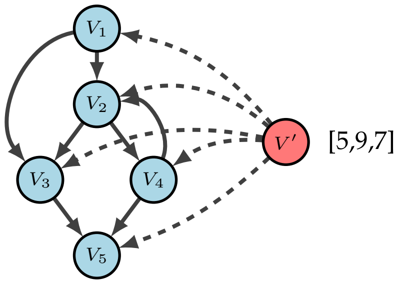

Fig. 3: SNIA artificial node connections.

4.5 Attack I: Semantics-preserving Node Injection Attack (SNIA)

In contrast to applying perturbation into the original executable nodes of the ACFG, Semantics-preserving Node Injection Attack (SNIA), only seeks to apply perturbation to the inexecutable parts of the software binary while still affecting how the classifier views the original portion of the program, as is shown in Fig. 3. To do this, SNIA takes advantage of how the aggregation step (1) of a GNN defines the node representation vector of each node in one layer based on its neighbor's values of the previous layer. For instance, if we wanted to change the value of some targeted node representation vector h_v without applying perturbation directly, all we need to do is add an edge from some node containing perturbation v' to the target node v. Since $v' \in N(v)$, the new node representation vector h_v^{l+1} should be different than h_v^l.

Building upon this idea, to develop an AE given some ACFG, we start by injecting a single artificial node so the code represented by the artificial node may never be executed, as is shown in Fig. 3. That is, there should be no directed edge from any of the original nodes to the artificial node. As seen in Algorithm 2, each element of the artificial nodes feature vector is initially set to zero. To determine which connections to make from the artificial node to the original nodes, we choose the set of original

Algorithm 2 Algorithm of the SNIA Attack

```
Input: $G = (A, X)$, $y$, $f'$, $\theta'$, $n_i$, $\Delta$, $n_f$
/* $n_i$: iterations, $n_f$: features */
Output: $G' = (A', X')$
1: $i \leftarrow 1$
2: $X^0 \leftarrow$ zeros$(1, n_f)$ /* Initial injected node feature */
3: $X' \leftarrow X \| X^0$ /* $\|$ is a concatenation operator */
4: $A' \leftarrow$ edgeGenerator$(A)$ /* Create new edges */
5: while $i \leq n_i$ do
6:     $X^i \leftarrow \sigma(X^{i-1} + \text{sign}(\nabla_{X^{i-1}} J_{f'}(\theta', X', A', y)))$
7:     $X' \leftarrow X \| X^i$ /* Update perturbation */
8:     if $|X - X'| \geq \Delta$ then /* Cap the perturbation */
9:         break
10:    end if
11:    $i \leftarrow i + 1$
12: end while
13: $G' \leftarrow (A', X')$
```

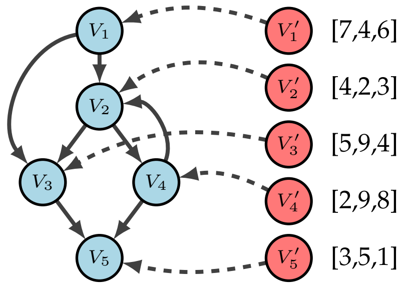

Fig. 4: SMNIA artificial node connections.

nodes that have the highest centrality value, e.g., degree, eigenvector, closeness, or betweenness centrality. The idea behind using centrality to make connections is that the nodes with the highest centrality value will be able to propagate the perturbation more effectively to other nodes with each aggregation step. Once the initial vector is set and the new edges are added, we iteratively update the node-feature vector of the artificial node by using a substitute model f'. For each iteration, we use the signed gradient of the cost function J for the substitute model f' to determine which instructions to inject. Since the perturbation being added is in an inexecutable part of the ACFG, the perturbation should be updated with respect to equation (7). We continue to update the AE iteratively until the maximum number of iterations has been executed or the perturbation limit has been reached.

4.6 Attack II: Semantics-preserving Multiple Node Injection Attack (SMNIA)

Similar to SNIA, the Semantics-preserving Multiple Node Injection Attack (SMNIA) only applies perturbation to artificial inexecutable nodes of an ACFG. The only difference is that SMNIA has multiple artificial nodes where each artificial node only connects to a single original node, as shown in Fig. 4. Again, each element of every artificial node-feature vector is set to zero, connections to the original nodes are determined by centrality values, and the artificial nodes are iteratively updated by using the signed gradient as is shown in Algorithm 3.

Algorithm 3 Algorithm of the SMNIA Attack

```
Input: $G = (A, X)$, $y$, $f'$, $\theta'$, $n_i$, $\Delta$, $n_e$, $n_f$
/* $n_i$: iterations, $n_f$: features, $n_e$: edges */
Output: $G' = (A', X')$
1: $i \leftarrow 1$
2: $X^0 \leftarrow$ zeros$(n_e, n_f)$ /* New node for each new edge */
3: $X' \leftarrow X \parallel X^0$
4: $A' \leftarrow$ edgeGenerator$(A)$ /* Create new edges */
5: while $i \leq n_i$ do
6:     $X^i \leftarrow \sigma(X^{i-1} + \text{sign}(\nabla_{X^{i-1}} J_{f'}(\theta', X', A', y)))$
7:     $X' \leftarrow X \parallel X^i$ /* Update perturbation */
8:     if $|X - X'| \geq \Delta$ then /* Cap the perturbation */
9:         $break$
10:    end if
11:    $i \leftarrow i + 1$
12: end while
13: $G' \leftarrow (A', X')$
```

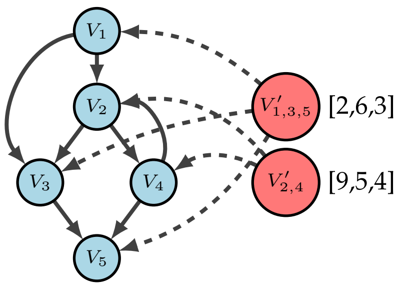

Fig. 5: SNICA artificial node connections.

The main benefit of using a single artificial node per connection is that each artificial node does not have to overgeneralize the perturbation associated with it. For instance, if an artificial node is connected to multiple nodes, it must generalize what perturbation is best for all of them, even if the optimal perturbation for one connected node is different than another. Although the extra artificial nodes can lead to additional accuracy in evasion, a drawback is an increase in the amount of perturbation required to create an AE per connection. Since each artificial edge must have its own perturbation, much more perturbation will be added to the sample as the number of edges increases. To obtain the accuracy of SMNIA and the low perturbation of SNIA, we designed the Semantics-preserving Node Injection Clustering Attack (SNICA).

4.7 Attack III: Semantics-preserving Node Injection Clustering Attack (SNICA).

Although the SMNIA attack is sufficient for producing effective AEs, much of the perturbation is superfluous and can be reduced significantly. For instance, if the node-feature vector of two artificial nodes v_1' and v_2' are equal or very similar to one another, there is no need to add perturbation to both of the nodes. Instead, we could remove node v_2' and its associated perturbation and have v_1' connect to the original node v_2' previously connected to, i.e., $(v_1', v_2) \in E$. Therefore, to reduce the repetitious perturbation produced by SMNIA, SNICA combines similar nodes through hierarchical agglomerative clustering on the set of artificial node-feature vectors X' with respect to the Euclidean distance.

Algorithm 4 Algorithm of the SNICA Attack

```
Input: $G = (A, X)$, $y$, $f'$, $\theta'$, $n_i$, $m_i$, $\Delta$, $n_e$, $n_f$, $n_c$
/* $n_i, m_i$: iterations, $n_e$: edges, $n_f$: features, $n_c$: clusters */
Output: $G' = (A', X')$
1: $i \leftarrow 1$
2: $X^0 \leftarrow$ zeros$(n_e, n_f)$ /* New node for each new edge */
3: $X' \leftarrow X \| X^0$
4: $A' \leftarrow$ edgeGenerator$(A)$ /* Create new edges */
5: while $i \leq n_i$ do
6:     $X^i \leftarrow \sigma(X^{i-1} + \text{sign}(\nabla_{X^{i-1}} J_{f'}(\theta', X', A', y)))$
7:     $X' \leftarrow X \| X^i$ /* Update temporary perturbation */
8:     if $|X - X'| \geq \Delta$ then /* Cap the perturbation */
9:         break
10:    end if
11:    $i \leftarrow i + 1$
12: end while
13: $A' \leftarrow$ clusterNodes$(A, X^i)$ /* Cluster similar nodes */
    /* Develop new perturbation for clustered nodes */
14: $X^0 \leftarrow$ zeros$(n_c, n_f)$
15: $X' \leftarrow X \parallel X^0$
16: $i \leftarrow 1$
17: while $i \leq m_i$ do
18:    $X^i \leftarrow \sigma(X^{i-1} + \text{sign}(\nabla_{X^{i-1}} J_{f'}(\theta', X', A', y)))$
19:    $X' \leftarrow X \parallel X^i$ /* Update perturbation */
20:    if $|X - X'| \geq \Delta$ then /* Cap the perturbation */
21:        break
22:    end if
23:    $i \leftarrow i + 1$
24: end while
25: $G' \leftarrow (A', X')$
```

As shown in Algorithm 4, SNICA starts by performing the SMNIA algorithm. After the artificial nodes are generated from SMNIA, the process of clustering the nodes begins. To combine the nodes, we group each node into one of n clusters using a hierarchical agglomerative clustering algorithm, e.g., ward, average, complete, or single. All nodes for each individual cluster are combined into a single node that keeps all of the original connections, as is shown in Fig. 5. The artificial node-feature vectors are then reset to zero vectors and regenerated through iterative updates with the signed gradient, as is done in both SNIA and SMNIA.

5 Experimental Evaluation

In this section, we perform experiments to determine the effectiveness of each of the semantics-preserving adversarial attacks.

5.1 Experimental Setup

To generate AEs for SNIA, SMNIA, SNICA, and SGI, we used a DGCNN for the target model and a simple GCN for the substitute model. The DGCNN model starts with four graph convolution layers, each with the ReLU activation function and 256, 128, 64, and 1 output channels. The outputs are concatenated and passed through a convolution layer with 16 output channels, a max pool layer of size

TABLE 1: Node-feature vector attributes.

Feature	Description
F1	# Numeric Constants
F2	# Transfer Instructions
F3	# Call Instructions
F4	# Arithmetic Instructions
F5	# Compare Instructions
F6	# Mov Instructions
F7	# Termination Instructions
F8	# Data Declaration Instructions

2, and another convolution layer with 32 output channels. The output is then flattened and passed through two dense layers each with 2048 output channels. It is finally passed through a final dense layer with an output of 2 to obtain the predicted label of benign or malicious. The GCN model has two graph convolution layers with the ReLU activation function and 64 output channels each. There is then a global mean pooling layer followed by a dense layer of size 1024 with a ReLU activation function and a dense layer of size 2 to obtain the final output.

All of the models used in this work were developed and trained using TensorFlow 2.11.0 on Windows 11 WSL2 using an RTX 3060ti Graphics Processing Unit (GPU). We also developed the models using a custom branch of Spektral—a graph deep learning library for Python and TensorFlow 2 [34]. To help create the AEs, we used an AE generation library called Clever Hans [35].

5.1.1 Dataset

To train the models, we used a dataset of 3000 malicious and 3000 benign x86 Linux software binaries. All of the software binaries were collected and processed on an Ubuntu 20.04 virtual machine using scripts written in Python 3.8. The malicious binaries are made up of no particular family of malware and were collected from VirusShare.com [36] during January 2022. All of the benign binaries were taken from across GitHub repositories and are used for a variety of different applications.

Data splits. The dataset is split into two training sets of 2400 samples to be used separately to train the target and substitute model. The training set for the substitute model was transformed so that each samples label corresponded with the output of the target DGCNN model. For testing and validation, the dataset is split into a validation and testing set comprised of 600 samples each for the target and substitute model to share.

Feature extraction. To extract the ACFG from each software binary, we used a binary analysis framework for Python called Angr. Every node in each ACFG is comprised of a size 8 node-feature vector that contains 8 attributes derived from each node's x86 assembly code sequence [10] as is shown in Table 1. The classification results for the models over the training, validation, and testing sets are shown in Table 2.

5.1.2 Evaluation Metrics

In this paper, we use the following evaluation metrics. 1) *Training Accuracy (TrA):* the percentage of all correctly classified samples by a model over the training set. The training set was used to train each of the models. 2) *Validation Accuracy (VA):* the percentage of all correctly classified samples by a model over the validation set. The validation set was used to tune the hyperparameters of the models. 3) *Test Accuracy (TeA):* the percentage of all correctly classified samples by a model over the testing set. The testing set was only used once per model to evaluate how well it performed on unseen data. 4) *False Positive (FP):* the percentage of benign binaries that were classified as malware by a model. 5) *False negative (FN):* the percentage of malware binaries that were classified as benign by a model. 6) *Max Perturbation Cap (MPC):* a constraint used for limiting the amount of perturbation that can make up an AE while it is being generated. It is defined as the maximum ratio of perturbed instructions over the number of total instructions of an AE. 7) *Total Perturbation (ToP):* the ratio of perturbed instructions over the total number of instructions of a generated AE. 8) *Evasion Rate (EvR):* the percentage of all misclassified AEs by a model.

TABLE 2: Classification performance. All evaluation metrics are defined in section 5.1.2, and are shown as percentages.

Model	TrA (%)	VA (%)	TeA (%)	FP (%)	FN (%)
Substitute GCN	95.88	97.83	95.83	2.50	1.67
Target DGCNN	99.88	97.67	96.67	2.33	1.00

5.2 Baselines

For our baselines, we conducted tests on both SGI and Graph Embedding and Augmentation (GEA) [19]. GEA is a simple semantics-preserving node-injection adversarial attack that does not require the use of gradients. Instead, it works by combining the ACFGs from two different software binaries. The attack embeds a target sample x_{tar} into some other sample x_{org} with the goal of preserving the functionality of x_{org} and having a model misclassify it. In our case, x_{tar} and x_{org} will be different classes, with one being a benign sample and the other being a malware sample.

5.3 Results

To determine how effective applying perturbations via node injection is, we conducted tests on GEA, SGI, SNIA, SMNIA, and SNICA. For our baseline, we used GEA, which embeds the ACFG from one sample into another, whereas SGI applies perturbations directly to the original executable nodes of the ACFG. In contrast to the aforementioned attacks, SNIA, SMNIA, and SNICA work by adding perturbations through inexecutable artificial nodes with edges connected to the original nodes of the ACFG. For the gradient-based AE generation methods, we tested several different max perturbation values to see how much of an influence they had on the evasion rate. We also tested how well node injection attacks work at evading a target model that has undergone adversarial retraining, which is a simple and effective way of mitigating AEs for GNN-based malware classifiers [37].

Comparison with Baseline (GEA). Our tests for GEA consisted of embedding a target sample ACFG into the ACFG of a sample with a different class. We selected benign and

TABLE 3: SGI evasion performance: maximum perturbation cap (MPC), evasion rate (EvR), and total perturbation (ToP). The lower part of the table shows a comparison with the baseline (GEA) for two perturbation

MPC (%)	EvR (%)	ToP (%)
1.00	64.17	0.95
2.00	71.50	1.45
3.00	78.33	1.92
4.00	78.67	2.36
5.00	79.50	2.78
Unlimited	77.17	7.80
GEA	27.44	2.95
GEA	60.37	63.21

malicious target samples based on their size. The small target samples produced AEs with an evasion rate of 27.44% and an average total perturbation of 2.95%. In comparison, the AEs with larger target samples had an evasion rate of 60.37% but had an average total perturbation of 63.21%. The low evasion rate of GEA is most likely due to there being no edges from the target ACFG nodes to the original ACFG nodes. Without edges from the target ACFG to the original ACFG nodes, the perturbation will have no impact on the original ACFG node feature vectors through the aggregation steps of the model.

Performance of SGI. As shown in Table 3, SGI was able to successfully attack the target model with an evasion rate of 79.50% with a max perturbation cap of 5.00%. At lower max perturbation caps such as 1.00%, SGI was only able to generate AEs with a 64.17% evasion rate. Even with no limit as to how much perturbation could be added to the sample, SGI was only able to achieve an evasion rate of 77.17%. With the SGI results, there seems to be a limitation on how effective adding executable perturbation can be. To see if applying perturbation through artificial nodes is more effective, we tested SNIA, SMNIA, and SNICA.

Performance of SNIA. To determine the best way of connecting the artificial node to the original nodes for SNIA, we tested four measurements of node centrality: degree, betweenness, closeness, and eigenvector centrality. Using these measurements, we added n edges from the artificial node to the n nodes of the sample that had the highest centrality values. Table 4 shows the results of SNIA using each centrality measurement with a max perturbation cap of 5.00% and 5 connected edges. Our results show that the best node centrality measurement for adding edges from the artificial node to the original nodes was degree centrality. It outperformed all other measurements in both evasion performance and in the amount of perturbation needed to create an effective AE. Because of this, we used degree centrality to add edges from the artificial nodes to the original ACFG nodes for SNIA, SMNIA, and SNICA.

Comparison and SMNIA. As illustrated in Figure 6, SNIA is more effective in generating adversarial examples than SGI as we increase the number of edges connecting the artificial node with the original nodes. With a max perturbation cap of only 1.00%, SNIA was able to generate AEs with an evasion rate greater than 80% once the number of connected edges was greater than 3. Moreover, as we increased the max

TABLE 4: SNIA's performance for different centrality measurements: evasion rate (EvR) and total perturbation (ToP).

Node Centrality	EvR (%)	ToP (%)
Degree Centrality	94.00	2.81
Betweenness Centrality	93.67	3.17
Closeness Centrality	92.67	4.35
Eigenvector Centrality	90.83	3.41

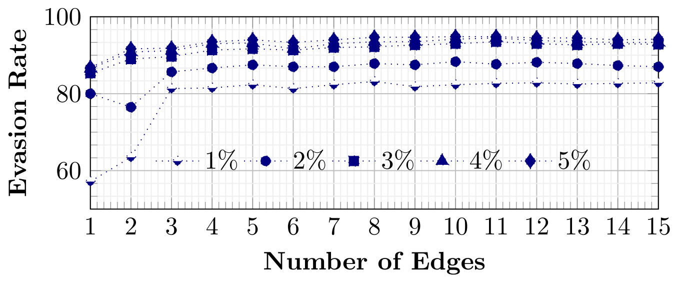

Fig. 6: SNIA evasion rate with max perturbation caps (varying, up to 5%, as shown in the legend with the different percentages) and a varying number of edges.

perturbation cap, the evasion rates continued to increase. With a max perturbation cap of 5.00% and 10 edges, SNIA was able to generate AEs with an evasion rate of 94.83% and an average total perturbation of 2.49%. However, the evasion rate starts to plateau with respect to an increasing amount of added edges. We believe this is due to an over-generalization of the artificial node perturbation being added as the perturbation must be a good fit for each connected original node. To see if this was the case, we tested AE generation with SMNIA, where each artificial node only connects to a single original ACFG node.

SMNIA was able to have a higher evasion rate than SNIA as shown in Figure 7. However, this is only when the amount of perturbation that can be added is unrestricted. With no max perturbation cap and 8 edges, SMNIA was able to generate AE's with an evasion rate of 98.67% with an average total perturbation of 20.08%. When the total perturbation is restricted, SMNIA is unable to produce AEs with a greater evasion rate. Since there is only a single edge connection per artificial node, the perturbation that is added is unable to propagate as effectively during the aggregation steps in contrast to a technique that has multiple edge connections per artificial node, such as SNIA. To try and find a middle ground between the two adversarial attacks, we tested SNICA, which has multiple artificial nodes and multiple connections per node.

Performance of SNICA. For SNICA, we performed the tests with a max perturbation cap of 5.00% and with different amounts of injected artificial nodes. As shown in Figure 8, SNICA is able to produce much more effective AEs than SMNIA when restricted with a perturbation cap. With 2 artificial nodes and 10 edges, SNICA is able to produce AEs with a 94.00% evasion rate and 3.53% average total perturbation. Without a max perturbation cap, 2 nodes, and 8 edges, SNICA produced AEs with a 97.50% evasion rate and only 6.30% average total perturbation. In contrast to SMNIA, this is only down from an evasion rate of

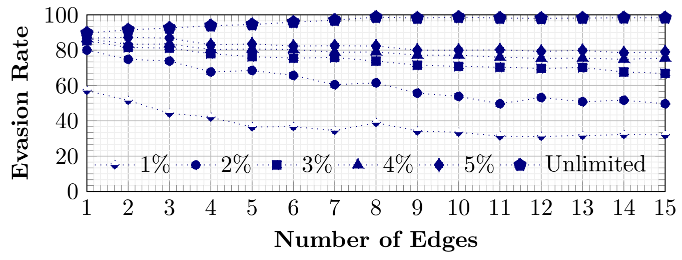

Fig. 7: SMNIA evasion rate with max perturbation caps (varying, up to unlimited, as shown in the legend with the different percentages) and a varying number of edges.

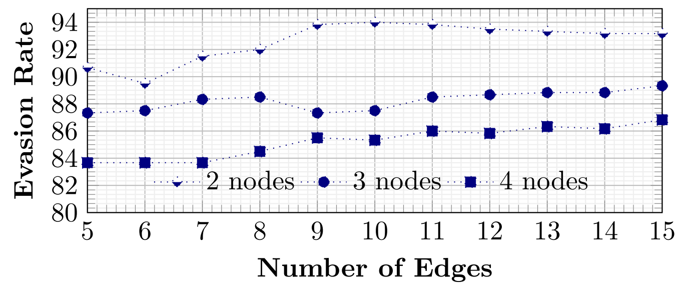

Fig. 8: SNICA evasion rate with a max perturbation cap of 5.00% under a varying number of nodes and edges.

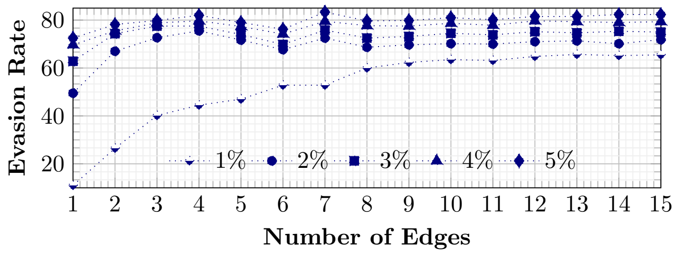

Fig. 9: SNIA evasion rate for a target model that has undergone adversarial retraining, with max perturbation caps (varying, up to 5%, as shown in the legend with the different percentages) and a varying number of edges.

TABLE 5: Adversarial attack evasion rates with a max perturbation cap of 5.00% for SGI and our approach, and design-compliant (small, large) perturbation in GEA (for comparison to a baseline).

Attack	EvR (%)	ToP (%)
GEA (small)	27.44	2.95
GEA (large)	60.37	63.21
SGI	79.50	2.78
SNIA (ours)	**94.83**	**2.49**
SMNIA (ours)	87.17	4.99
SNICA (ours)	94.00	3.53

98.67%, whereas the perturbation decreased down from 20.08%. However, when we inject more than 2 artificial nodes, the evasion rate starts to decrease once again due to the restrictions on the amount of perturbation that can be added. Also, SNICA requires additional time for generating each AE, averaging 0.94 seconds per sample, in contrast to 0.76 seconds for SNIA and 0.40 seconds for SMNIA. These times are observed for AEs with 5 edges and a maximum perturbation cap of 5.00%.

Resistance to Adversarial Retraining. To determine the robustness of node-injection attacks to AE defenses, we performed tests with a target model that has undergone adversarial retraining. The target model was trained with half normal samples and half AE's that were generated using SNIA with a max perturbation cap of 3.00% and 5 edges. As shown in Figure 9, adversarial retraining does have an impact on the evasion rate when compared to a target model that has not received adversarial retraining. However, the evasion rate is still higher than both of the baselines while still requiring only a small amount of perturbation. With a max perturbation cap of 5.00% and 7 connected edges, SNIA could still generate AEs at an 83.33% evasion rate while requiring an average total perturbation of 1.41%.

Throughout these experiments, we observed that each of the node injection adversarial attacks was able to outperform SGI and GEA. As shown in Table 5, each node injection adversarial attack was able to achieve a higher evasion rate than the baselines. Moreover, SNIA, in particular, was able to achieve much higher evasion rates than SGI and GEA while also being robust to adversarial retraining and requiring less perturbation on average to create an effective AE. Our experiments also showed that by injecting more artificial nodes, we were able to achieve a higher evasion rate. With 10 nodes and edges, SMNIA was able to obtain the highest evasion rate at 98.67%. Furthermore, it was shown with SNICA that the total perturbation of an AE can be reduced by clustering injected nodes while still obtaining high evasion rates. Therefore, we argue that our experiments show that creating AEs through node injection is more effective and flexible than applying perturbation directly to the original ACFG nodes.

6 Discussion and Future Work

In this work, we looked at several different types of semantics-preserving adversarial attacks. As a baseline, we looked at a graph embedding attack called GEA, which injected the ACFG of a target sample into another sample. Another type applied perturbations directly to the original executable nodes of the ACFG by injecting `nop` semantic instructions, which was demonstrated with SGI. We presented several node injection attacks that applied perturbations through inexecutable artificial nodes that had edges connected to the original nodes of the ACFG, which was demonstrated by SNIA, SMNIA, and SNICA. The node injection attacks were all able to outperform SGI and GEA in both evasion rate and the amount of perturbation needed to craft an effective AE. Finally, we showed that node injection attacks are still able to evade target models that have undergone adversarial retraining.

Although SNIA was able to produce excellent results, the evasion rate started to plateau with an increasing number of added edges. We believe that this is due to an overgeneralization of the perturbation for the artificial node, as it must contain optimal perturbation for every original node it is connected to. To test if this was the case, we performed tests on SMNIA, where each artificial node was only connected to a single original node. When the amount of perturbation that could be added was not limited, the evasion rate was able to get much higher than SNIA. However, SMNIA failed to perform well under any significant perturbation constraints. We believe this is due to the lack of connections per artificial node, which prevented perturbation from efficiently propagating to the original nodes during the aggregation steps. With the SNICA tests, we showed that we could combine artificial nodes with similar node-feature vectors to reduce the amount of perturbation needed to create an AE.

To connect the inexecutable artificial nodes to the original nodes of the ACFG, we tested several different node centrality measurements such as degree, betweenness, closeness, and eigenvector centrality. For future work, we would like to explore ideas of applying more sophisticated node connections similar to other techniques introduced by various researchers [28]–[31]. By using more sophisticated edge generation techniques, we believe that we could obtain higher evasion rates and lower perturbation for all three of the presented adversarial attacks.

Moreover, future work should include how to detect and prevent node injection attacks such as SNIA, SMNIA, and SNICA. One possible avenue of detection could be a pruning stage during the extraction of the ACFG that removes all nodes that do not have a path to execution. If this is successfully done, all of the perturbations generated by the adversary should be removed before the ACFG is fed into a model for classification. However, such a pruning step could potentially eliminate useful information that is not perturbation depending on the accuracy of the ACFG generation technique [38].

Finally, future research should look into how node injection attacks similar to SNI, SMNIA, and SNICA can be extended to other GNN-based malware detectors that use graph data other than ACFGs. For instance, Gao *et al.* [39] proposed GDroid, a GCN model that uses API graphs to detect Android malware. Another work by Cai *et al.* [40] has shown that function call graphs can detect malware with GNNs.

7 Conclusion

This paper proposed several semantics-preserving grey-box adversarial attacks for GNNs that use ACFGs for malware classification. Our results showed that adversaries can craft AEs that take advantage of the aggregation steps of GNNs by propagating perturbations from injected inexecutable nodes to the original executable nodes of an ACFG. In contrast to SGI — a similar attack that applies perturbation directly to the executable portion of an ACFG — SNIA, SMNIA, and SNICA all produced AEs with higher evasion rates and lower amounts of perturbation. Moreover, the node injection attacks were able to greatly outperform our baseline attack GEA. Each node injection method had a two-step process of generating AEs. First, each of the artificial nodes was connected to the original nodes through a degree centrality ranking step. Next, the perturbation associated with each artificial node was generated by iteratively applying the signed gradient. Overall, our results show a need for more robust GNN-based malware detectors that use ACFGs for classification.

References

[1] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 83–97, 2016.

[2] A. Mohaisen, O. Alrawi, and M. Mohaisen, "AMAL: high-fidelity, behavior-based automated malware analysis and classification," *Comput. Secur.*, vol. 52, pp. 251–266, 2015.

[3] Y. Xiao, Y. Jia, X. Cheng, S. Wang, J. Mao, and Z. Liang, "I know your social network accounts: A novel attack architecture for device-identity association," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1017–1030, 2023.

[4] A. A. Elkhail, N. Lachtar, D. Ibdah, R. Aslam, H. Khan, A. Bacha, and H. Malik, "Seamlessly safeguarding data against ransomware attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 1, pp. 1–16, 2023.

[5] C. Wang, Z. Qin, J. Zhang, and H. Yin, "A malware variants detection methodology with an opcode based feature method and a fast density based clustering algorithm," in *2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, pp. 481–487, 2016.

[6] E. Raff, R. Zak, R. Cox, J. Sylvester, P. Yacci, R. Ward, A. Tracy, M. Mclean, and C. Nicholas, "An investigation of byte n-gram features for malware classification," *Journal of Computer Virology and Hacking Techniques*, vol. 14, 02 2018.

[7] Z. Markel and M. Bilzor, "Building a machine learning classifier for malware detection," in *2014 Second Workshop on Anti-malware Testing Research (WATeR)*, pp. 1–4, 2014.

[8] H. Alasmary, A. Khormali, A. Anwar, J. Park, J. Choi, A. Abusnaina, A. Awad, D. Nyang, and A. Mohaisen, "Analyzing and detecting emerging internet of things malware: A graph-based approach," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8977–8988, 2019.

[9] M. Hassen and P. K. Chan, "Scalable function call graph-based malware classification," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pp. 239–248, 2017.

[10] J. Yan, G. Yan, and D. Jin, "Classifying malware represented as control flow graphs using deep graph convolutional neural network," in *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pp. 52–63, IEEE, 2019.

[11] P. Xu, Y. Zhang, C. Eckert, and A. Zarras, "Hawkeye: cross-platform malware detection with representation learning on graphs," in *International Conference on Artificial Neural Networks*, pp. 127–138, Springer, 2021.

[12] S. Jiang, Y. Hong, C. Fu, Y. Qian, and L. Han, "Function-level obfuscation detection method based on graph convolutional networks," *Journal of Information Security and Applications*, vol. 61, p. 102953, 2021.

[13] P. Feng, J. Ma, T. Li, X. Ma, N. Xi, and D. Lu, "Android malware detection based on call graph via graph neural network," in *2020 International Conference on Networking and Network Applications (NaNA)*, pp. 368–374, 2020.

[14] C. Li, G. Shen, and W. Sun, "Cross-architecture intemet-of-things malware detection based on graph neural network," in *2021 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–7, 2021.

[15] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *Thirty-second AAAI conference on artificial intelligence*, 2018.

[16] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[17] H. Wu, C. Wang, Y. Tyshetskiy, A. Docherty, K. Lu, and L. Zhu, "Adversarial examples on graph data: Deep insights into attack and defense," 2019.

[18] A. Abusnaina, M. Abuhamad, H. Alasmary, A. Anwar, R. Jang, S. Salem, D. Nyang, and D. Mohaisen, "DL-FHMC: deep learning-based fine-grained hierarchical learning approach for robust malware classification," *IEEE Trans. Dependable Secur. Comput.*, vol. 19, no. 5, pp. 3432–3447, 2022.

[19] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, and A. Mohaisen, "Adversarial learning attacks on graph-based iot malware detection systems," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1296–1305, 2019.

[20] A. Abusnaina, H. Alasmary, M. Abuhamad, S. Salem, D. Nyang, and A. Mohaisen, "Subgraph-based adversarial examples against graph-based iot malware detection systems," in *International Conference on Computational Data and Social Networks*, pp. 268–281, Springer, 2019.

[21] L. Zhang, P. Liu, Y.-H. Choi, and P. Chen, "Semantic-preserving reinforcement learning attack against graph neural networks for malware detection," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[22] R. Yumlembam, B. Issac, S. M. Jacob, and L. Yang, "Iot-based android malware detection using graph neural network with adversarial defense," *IEEE Internet of Things Journal*, pp. 1–1, 2022.

[23] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[24] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations*, 2015.

[25] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *Artificial intelligence safety and security*, pp. 99–112, Chapman and Hall/CRC, 2018.

[26] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, (New York, NY, USA), p. 506–519, Association for Computing Machinery, 2017.

[27] H. Li, Z. Cheng, B. Wu, L. Yuan, C. Gao, W. Yuan, and X. Luo, "Black-box adversarial example attack towards fcg based android malware detection under incomplete feature information," *arXiv preprint arXiv:2303.08509*, 2023.

[28] S. Tao, Q. Cao, H. Shen, J. Huang, Y. Wu, and X. Cheng, "Single node injection attack against graph neural networks," in *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, ACM, oct 2021.

[29] X. Zou, Q. Zheng, Y. Dong, X. Guan, E. Kharlamov, J. Lu, and J. Tang, "TDGIA: Effective injection attacks on graph neural networks," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ACM, aug 2021.

[30] J. Fang, H. Wen, J. Wu, Q. Xuan, Z. Zheng, and C. K. Tse, "Gani: Global attacks on graph neural networks via imperceptible node injections," 2022.

[31] Y. Chen, H. Yang, Y. Zhang, K. Ma, T. Liu, B. Han, and J. Cheng, "Understanding and improving graph injection attack by promoting unnoticeability," 2022.

[32] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," *arXiv preprint arXiv:1810.00826*, 2018.

[33] X. Ling, L. Wu, W. Deng, Z. Qu, J. Zhang, S. Zhang, T. Ma, B. Wang, C. Wu, and S. Ji, "Malgraph: Hierarchical graph neural networks for robust windows malware detection," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pp. 1998–2007, IEEE, 2022.

[34] D. Grattarola and C. Alippi, "Graph neural networks in tensorflow and keras with spektral," 2020.

[35] N. Papernot, F. Faghri, N. Carlini, I. Goodfellow, R. Feinman, A. Kurakin, C. Xie, Y. Sharma, T. Brown, A. Roy, A. Matyasko, V. Behzadan, K. Hambardzumyan, Z. Zhang, Y.-L. Juang, Z. Li, R. Sheatsley, A. Garg, J. Uesato, W. Gierke, Y. Dong, D. Berthelot, P. Hendricks, J. Rauber, and R. Long, "Technical report on the cleverhans v2.1.0 adversarial examples library," *arXiv preprint arXiv:1610.00768*, 2018.

[36] "https://virusshare.com/," 2022.

[37] H. Li, S. Zhou, W. Yuan, X. Luo, C. Gao, and S. Chen, "Robust android malware detection against adversarial example attacks," in *Proceedings of the Web Conference 2021*, pp. 3603–3612, 2021.

[38] K. Liu, H. B. K. Tan, and X. Chen, "Binary code analysis," *Computer*, vol. 46, no. 8, pp. 60–68, 2013.

[39] H. Gao, S. Cheng, and W. Zhang, "Gdroid: Android malware detection and classification with graph convolutional network," *Computers & Security*, vol. 106, p. 102264, 2021.

[40] M. Cai, Y. Jiang, C. Gao, H. Li, and W. Yuan, "Learning features from enhanced function call graphs for android malware detection," *Neurocomputing*, vol. 423, pp. 301–307, 2021.

Dylan Zapzalka obtained his B.S. in Computer Science and Mathematics from North Dakota State University in 2022. He is currently an Associate Software Engineer at Veritas Technologies. Dylan Zapzalka will be attending the University of Michigan as Ph.D. student studying Computer Science starting in 2023. His interests span the areas of machine learning, cybersecurity, and causal inference.

Saeed Salem received his Ph.D. in computer science from Rensselaer Polytechnic Institute, New York. He is currently a full professor of Computer Science and Engineering at Qatar University. Before then, he was a Full Professor at North Dakota State University. Dr. Salem's research is in the broad areas of graph mining and machine learning with a focus on developing algorithms for learning from graphs with applications in the domain of security, and biological networks. Dr. Salem's group developed enumeration algorithms for mining all frequent subgraphs, cross-graph dense graphs, and approximate frequent subgraphs from heterogeneous graphs.

David Mohaisen obtained his Ph.D. in Computer Science from the University of Minnesota in 2012. He is currently a professor of computer science at the University of Central Florida, where he leads the Security and Analytics Lab (SEAL), which he has been leading since 2017. Previously, he was an Assistant Professor at SUNY Buffalo (2015-2017) and a Senior Scientist at Verisign Labs (2012-2015). His research interests are in applied security and privacy, covering aspects of computer and networked systems, software systems, IoT and AR/VR, and machine learning. His research has been published in top conferences and journals, with multiple best paper awards. His work was also featured in the New Scientist, MIT Technology Review, ACM Tech News, Science Daily, etc. Among other services, he has been an Associate Editor of IEEE TMC, TDSC, TCC, and TPDS. He is a senior member of ACM (2018) and IEEE (2015), a Distinguished Speaker of the ACM, and a Distinguished Visitor of the IEEE Computer Society.