# Adversarial Learning Attacks on Graph-based IoT Malware Detection Systems

Ahmed Abusnaina Aminollah Khormali Hisham Alasmary
Jeman Park Afsah Anwar Aziz Mohaisen
University of Central Florida

***Abstract*—IoT malware detection using control flow graph (CFG)-based features and deep learning networks are widely explored. The main goal of this study is to investigate the robustness of such models against adversarial learning. We designed two approaches to craft adversarial IoT software: off-the-shelf methods and Graph Embedding and Augmentation (GEA) method. In the off-the-shelf adversarial learning attack methods, we examine eight different adversarial learning methods to force the model to misclassification. The GEA approach aims to preserve the functionality and practicality of the generated adversarial sample through a careful embedding of a benign sample to a malicious one. Intensive experiments are conducted to evaluate the performance of the proposed method, showing that off-the-shelf adversarial attack methods are able to achieve a misclassification rate of 100%. In addition, we observed that the GEA approach is able to misclassify all IoT malware samples as benign. The findings of this work highlight the essential need for more robust detection tools against adversarial learning, including features that are not easy to manipulate, unlike CFG-based features. The implications of the study are quite broad, since the approach challenged in this work is widely used for other applications using graphs.**

***Keywords*-Adversarial Learning, Deep Learning, Graph Analysis, Internet of Things, Malware Detection**

## I. Introduction

The Internet of Things (IoT) is a novel networking prototype that interconnects a large number of devices, with many different applications, such as sensors, voice assistants, automation tools, *etc.* [1]. Multiple pieces of software or applications are installed in each of those devices to function. At the same time, those applications can be exploited through vulnerabilities, leading to a wide variety of security threats and impacts, such as Distributed Denial of Service (DDoS) attacks launched by Mirai botnet [2]. Thus, it is essential to understand IoT software to address those security issues through analysis, abstraction, and classification [1], [3]. To do so, there has been a large body of research work on the problem of software analysis in general, and a few attempts on analyzing IoT software in particular. It should be noted that the research work on IoT software analysis has been very limited not only in the size of the analyzed samples, but also the utilized approaches [4], [5]. A promising direction leverages a graph-theoretic approach in security [6], particularly, to analyze IoT malware. Representative static characteristics can be extracted from a graph [7], [8], such as, Control Flow Graph (CFG), which is abstracted from IoT malware samples. As IoT software can be represented using graph-based features from CFG, those features can be utilized to build an automatic detection system to identify whether a given software is malicious or benign [9]. Moreover, the type of the malicious software can be identified through malware family-level classification and label extrapolation, a concept widely applied [10].

Machine learning algorithms, specifically deep learning networks, are actively used in the process of detecting/classifying malicious software from benign ones [10], [11]. Generally, machine/deep learning networks, thanks to their high performance, are widely used in a wide range of applications, such as health-care [12], finance [13], industry [14], [15], computer-vision [16], and cyber-security [17], [18]. For instance, machine learning theory is leveraged into the process of software graph analysis to build more powerful analysis tools [19]. One such application is exploring IoT malware using both graph analysis and machine learning [9]. These models not only can learn the representative characteristics of the graph, but also can be utilized to build automatic detection system to predict the label of the unseen software. However, the rise in the utilization of deep learning models in security-related domains creates incentives for adversaries to manipulate the underlying model to produce their desired outputs. It has been shown that the machine/deep learning networks are prone to vulnerabilities. For example, an adversary can force the model to produce his desired output, *e.g.,* misclassification, through crafting the adversarial examples (AEs) [20], [21].

Machine and deep learning models learn the inherent pattern of the input dataset. Therefore, the model output is highly dependent on the input dataset. At the same time, the AEs are being crafted through applying small perturbation to the input dataset. Note that the crafted samples are very similar to the original ones, and are not necessarily outside of the training data manifold. Recently, researchers presented several algorithms for generating adversarial examples, such as the fast gradient sign method [22], DeepFool method [21], the Jacobin-based saliency map method [20], *etc.*. Although it is an active research area in the security and machine learning communities, there is very little research work done on understanding the impact of adversarial learning on deep learning-based IoT malware detection system and practical implications [23], particularly those that utilize CFG features. It is worth noting that labeling a malicious IoT software as benign may lead to disastrous results in practice, particularly in sensitive applications and domains, highlighting the importance of the issue to be explored in more detail.

**Goal of this study.** Motivated by the aforementioned issues, our main goal is generating *adversarial IoT software samples* *that (1) fool the classifier and (2) function as intended.*

**Approach.** To tackle the above objectives, first we conducted an in-depth analysis of malware binaries through constructing abstract structures using CFG, which are analyzed from multiple aspects, such as the number of nodes and edges, as well as graph algorithmic constructs, including average shortest path, betweenness, closeness, density, *etc.* Then we designed two approaches to craft AEs, including off-the-shelf adversarial learning approach and GEA approach, while applying small changes to the graph features. The off-the-shelf adversarial attack approach incorporates eight well-known adversarial learning methods to force the model to misclassification. Whereas, the GEA approach aims to preserve the functionality and practicality of the generated adversarial sample through a careful connection of benign graph to a malicious one.

**Findings.** Extensive experiments are conducted to evaluate the performance of our work. The results demonstrate that off-the-shelf adversarial attack methods are able to achieve a high misclassification rate of 100%, while the GEA approach is able to *misclassify all malware samples as benign*. The findings highlight the need for more robust detection tools against adversarial learning, such as more sophisticated features that are not easy to manipulate, unlike CFG-based features.

**Summary of contributions.** Our contributions are as follows: First, we examined the robustness of CFG-based deep learning IoT malware detection system using two different approaches, including off-the-shelf adversarial learning algorithms and graph embedding and augmentation. The proposed GEA approach that generates adversarial IoT software, through embedding representative target sample to the original software, while maintains the practicality and functionality of the attacked sample. Second, we evaluated the performance of the proposed method via intensive experiments showing the effectiveness of the proposed approach in producing successful AEs. We found that although off-the-shelf adversarial learning algorithms can generate AEs with high misclassification rate of 100%, they do not guarantee the practicality and functionality of the crafted AEs. Whereas, the proposed graph embedding and augmentation approach maintains the practicality of the generated adversarial IoT software examples, which can misclassify all malicious IoT software as benign.

**Organization.** In section II, a brief background is provided. The presented practical approach for generating practical adversarial IoT software is described in section III. The performance of the proposed approach, evaluated through intensive experiments, is in section IV. Related work has been discussed in section V. Finally, limitations and future work are discussed in section VI, followed by the conclusion in section VII.

## II. Preliminaries

We incorporate adversarial learning techniques into CFG-based deep learning IoT malware detection systems in an attempt to understand the robustness of such models against adversarial learning attacks as a result of AEs.

We provide a required preliminary knowledge for understanding those techniques and approaches required for malware analysis to extract graph structures and to automate their labeling using machine learning. In particular, we provide general knowledge about the malware analysis approaches in §II-A. The CFG-based analysis for IoT malware detection is described in §II-B. Finally, we describe background knowledge about the concept of adversarial machine learning and its effects on machine/deep learning models in §II-C.

### *A. Malware Analysis*

Malware analysis is widely used to understand the functionality and the behavior of malware. It helps us to understand the capabilities and the intent of the malware and malware authors. The results of the analyses are often used to build detectors and design defenses to protect against future malware campaigns. There are two approaches utilized for analyzing malicious software: (i) static and (ii) dynamic analysis. Static analysis approaches analyze malware binaries without running them. Given the malicious nature of the malware, static analysis is utilized as a precursor to the dynamic analysis. The malware binary can then be executed in a sandboxed environment with a much reduced focus to observe the patterns, like the behavioral artifacts – in what is called the dynamic analysis.

**Static Analysis.** Static analysis approaches employ various techniques to extract indicators to determine whether the software is malicious or benign [24]. The various analysis points, such as strings, functions, disassembly *etc.* hint at the possible execution pattern of the software. For example, the traces of user name and password list, along with shell based login attempt, hints at possible usage of dictionary attack being utilized by the software. These inferential results are drawn from static analysis leverage the analysts to emphasize and scrutinize on specific patterns. Additionally, traces can also be used by analysts to address issues during dynamic analysis, *e.g.,* virtual machine obfuscation, ptrace obfuscation *etc.*.

Among the first steps towards static analysis of a software is understanding its composition. Reverse engineering can be used to understand a software's composition, architecture, and design. Off-the-shelf tools help analysts achieve the above goal. In addition to the aforementioned goals, reverse engineering can also help disassembling the software to generate its high-level representation, including CFGs and Data Flow Graphs (DFGs). The CFG of a program is the graphical representation of the flow of control during the execution of that program. while the DFG represents the system events to understand the possible execution of the system behaviors. It explains the flow of the data that passes from one node to another. Although static analysis is quite powerful and popular, it sometimes stops short of achieving its end goals due to multiple evasion techniques. For example, malware authors use evasion techniques to prevent their malware from being analyzed. Some of the techniques utilized include packing (UPX [25]), obfuscation (function-, string- obfuscation), *etc.*.

**Dynamic Analysis.** Unlike static analysis, dynamic analysis executes the application in a simulated and monitored environment to observe its behavior and understand its functionality [26]. This approach unearths different behavioral patterns of a software. In particular, it unravels a program's network patterns, such as communication with the Command

and Control (C&C) server. Since the malicious nature of software can affect the status of the machine it is executed on, the following measures are adopted: 1) comparing the system state before and after the execution of the application, or 2) monitoring the application's actions during the execution.

In line with the static analysis, software authors adopt means to prevent their software from getting reversed. To do so, they employ conditions such that the software exits or crashes upon encountering virtual machines, debugging tools and/or network monitoring tools. Additionally, dynamic analysis is time consuming.

### B. Graph Analysis

**Graph Analysis.** The CFG is a graph representation of the program which shows the all paths that can be reached during the execution as in Figure 2. In a CFG, the set of nodes means the basic blocks where each block is a straight-line instructions without any *jump* or *jump target*, while the set of directed edges corresponds to the *jump* which traverses from the block to the other block at the branch (*if*), loop (*while, for*), and the end of the function (*return*). Once the first instruction of the basic block is executed, the rest of the instructions in the same block are necessarily executed in order unless terminated by the external interference. In general, CFG is used for the structural analysis of the program. For example, from the perspective of optimization, the CFG is used to analyze the reachability of each block. By constructing the CFG and evaluating the reachability, the flaws of the program (infinite loop or unreachable codes) can be found and addressed.

**CFG-based Analysis.** In graph theory, there are various concepts that express the characteristics of a graph. Given $G = (V, E)$, for example, the number of vertices ($|V|$) means the order of $G$, while the number of edges ($|E|$) corresponds to the size of $G$. The density of the graph can be defined as $D = |E|/(|V| * (|V| - 1))$ for directed simple graph, which means the ratio of the number of edges in $G$ to the maximal number of edges in the complete graph. The centrality is measured for the each node $v \in V$, which shows how important a specific node is. In detail, there are several different kinds of centrality, such as closeness centrality, betweenness centrality, Eigenvector centrality, *etc.*.

These indicators (and further concepts not described above) can be considered the features of the graph $G$. Moreover, the combination of those metrics can be a more deterministic characteristic of the graph. Considering that a CFG is a kind of graph, it is true that each binary has not only its unique graph representation but also the associated values, such as the order, size, and density of CFG, and centrality for each vertex in CFG. On the other hands, the graph-based analysis can provide the possibility for identifying the malware. Because it is highly likely that the binaries in the same "family" share the structural similarity (even if there is a little difference), the CFG-based features can be combined with the state-of-the-art machine learning technique to determine whether a given binary is malicious or not.

### C. Threat Model

In adversarial deep learning, the goal is to generate AE that forces the classifier $f$ to misclassify the input sample $x$ to the desired output. Such attack applies small perturbation to the original sample to craft the AE [20]. Attacks on deep learning network can be classified from different perspectives, including attack's target type into targeted or untargeted, and the knowledge of the adversary about the model to black-box, or white-box attacks [27], [28]. In this study, we assume that **the adversary has full knowledge regarding the structure, link weights, *etc.* of the model**. We also assume that the adversary tries to conduct **both targeted and untargeted misclassification attacks**.

A brief description of these adversarial attack categories assumed in our threat model is provided below.

**Targeted attacks.** The focus of this attack is to generate AE $x'$ that forces the classifier $f$ to misclassify into a specific target class $t$. For instance, the adversary generates a set of malicious IoT software samples, which are classified as benign. That is: $x' : [f(x') = t] \wedge [\Delta(x, x') \leq \epsilon]$, where $f(.)$ represents the classifiers output, $\Delta(x, x')$ denotes the difference between $x$ and the crafted AE $x'$, whereas $\epsilon$ is a distortion threshold.

**Untargeted attacks.** The focus of untargeted attack is to generate an AE that forces the classifier $f$ to misclassify to any class other than the original class $f(x)$, where $x$ is the original input. That is: $x' : [f(x') \neq f(x)] \wedge [\Delta(x, x') \leq \epsilon]$, where $f(.)$ shows the classifier's output, $\Delta(x, x')$ represents the difference between $x$ and $x'$, and $\epsilon$ is the distortion threshold.

## III. Generating Adversarial Examples

Deep learning-based classifiers have been widely used for IoT malware detection. Recent studies highlighted the vulnerability of deep learning models against adversarial machine learning attacks. Therefore, the key goal of this study is to investigate the robustness of deep learning-based IoT malware detection systems that are trained over CFG-based features. In addition, we try to generate realistic AEs that preserve the functionality and practicality of the original samples. To do so, we design two approaches: generic adversarial machine learning attacks and GEA. The first approach generates AEs using generic adversarial methods to conduct attacks on the deep learning-based IoT malware detection systems, causing misclassification. While the first approach may not generate practical AEs, due to applying changes to the feature space, which can be hard or unrealistic to be reflected to the CFG of the original sample, the GEA approach generates AEs that are realistic. More information regarding the proposed approaches are presented in §III-A and §III-B.

### A. Off-the-Shelf Adversarial Attacks

This approach incorporates well-established adversarial machine learning attack methods into IoT malware detection. These methods apply small perturbation into the feature space to generate AEs that lead to misclassification. The general flow graph of the AE's generation process is demonstrated in Figure 1. We trained a convolutional neural network (CNN) model with four convolutional layers based on the extracted

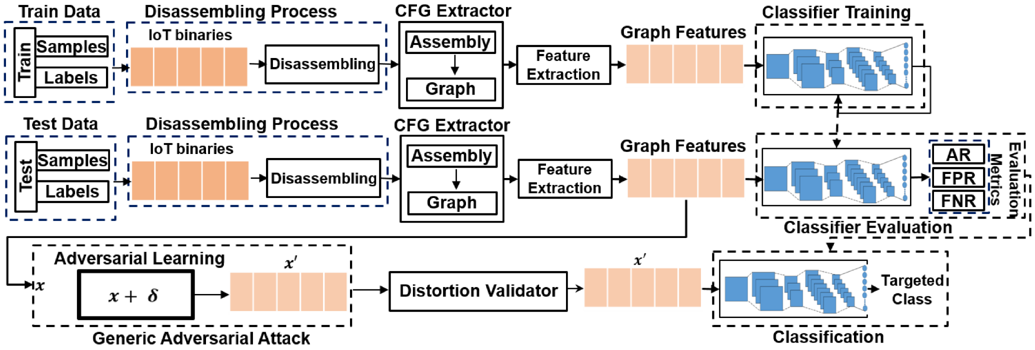


Fig. 1. General flow graph of the generic adversarial machine learning approach

features from the CFGs of the train dataset. Then we evaluated the performance of the trained model based on the test data and using standard metrics, such as accuracy rate, false positive rate, and false negative rate. Then, we used the evaluated model for the rest of our adversarial attack analysis, as our baseline model. Note that the distortion validation task checks whether each feature is within the feature space range. In the following, a brief description of the adversarial machine learning attack methods utilized in this study is provided.

**Carlini & Wagner (C&W) Method.** Carlini and Wagner [29] proposed gradient-based attacks to generate high-confidence AEs based on $L_\infty$, $L_2$, and $L_0$ norms by optimizing the penalty and distance metrics. The process is defined as $min\,||\delta||_p^2\ :\ \ g\,(x+\delta)\,=\,y'\ ,\ x+\delta\in X$, where $\delta$ is the perturbation parameter, $g()$ is the objective function of the classifier, $y'$ denotes the targeted class, and $x$ represents the input image. In this work, we generate AEs based on $L_2$ norm. As $L_2$ distance denotes the distance between the generated AE and the original sample. Therefore, a lower $L_2$ norm value corresponds to a smaller perturbation to the feature space required in generating the AE. The perturbation $\delta$ is defined as $\delta = \frac{1}{2}\,(tanh\,(w)+1) - x$, where $tanh()$ is the hyperbolic tangent function and $w$ is an auxiliary variable optimized by $\min\limits_{w}\,||\frac{1}{2}\,(tanh\,(w)+1)\,||_2 + c\cdot g\left(\frac{1}{2}\,(tanh\,(w)+1)\right)$, where $c$ is a constant. The key goal of C&W method is to minimize the distance between the generated AE and the original sample, increasing the similarity and harden the detection.

**DeepFool Method.** DeepFool is based on iterative linearization of the classifier to generate the AEs [21]. It utilizes $L_2$ distance metric to apply minimal perturbation to change the classification decision. The generating process increases the distance between the input and its associated class iteratively. The class is defined as $\hat{k}(x) = arg_k\ max\ f_k(x)$, where $f_k(x)$ is the output of the objective function corresponding to the class $k$. Assuming that $f(x)$ is an affine classifier represented by $f(x)\ =\ W^Tx\,+\,b$, then the perturbation needed to misclassify input $x$ is described as $arg_r\ min\ ||r||_2\quad s.t.\ \exists k:$ $w_k^T\,(x_0+r)+b_k\ \geq\ w_{\hat{k}(x_0)}^T\,(x_0+r)+b_{\hat{k}(x_0)}$, where $w_k$ is mapped to the $k$-th column of $W$.

**ElasticNet Method.** ElasticNet generates AEs based on $L_1$ distance, providing sparsity in the added perturbation regarding to the feature space [30]. ElasticNet is inspired from C&W, using the same loss function to craft the AEs. The untargeted attack is defined as $f\,(x) = \max\{[Logit\,(x)]_{y_0} - \max\limits_{j\neq y_0}[Logit\,(x)]_j, -k\}$, where $f$ represents the loss function, $x_0$ is the original sample, $y_0$ is the original label, $j$ denotes the current sample label, and $Logit$ is the logit function. ElasticNet generates AEs by manipulating the prediction via the loss function, the formula of the ElasticNet attack is $\min\limits_{\delta}\quad c.f\,(x,y)+\beta||\delta||_1+||\delta||_2^2$, where $\delta\,=\,x-x_0$, $x\in[0,1]^p$, where $c$, $\beta\geq 0$, $c$ and $\beta$ represent the regularization parameters of the loss function $f$.

**Fast Gradient Sign Method (FGSM).** FGSM is a fast method to generate AE based on one-step gradient update [22]. It does not take into account the similarity between the generated AE and the original sample. FGSM can be expressed as $\delta\ =\ \epsilon\cdot sign\,(\nabla_x J_\theta\,(x,y))$ where $\delta$ shows the applied perturbation, $\epsilon$ is a scalar value represents the perturbation threshold. Moreover, $sign(.)$ denotes the sign function, $J(.)$ is the cost function, $x$ is the input image, and $y$ is the label associated with it. $\nabla$ shows the gradient of the cost function $J$ around the current value of $x$. Finally, the output, AE, is $x' = x+\delta$ where $x'$ represents the generated AE.

**Jacobian-based Saliency Map Approach (JSMA).** Papernot *et al.* [20] introduced JSMA, an $L_0$-based iterative adversarial method that perturbs the features based on the adversarial saliency scores. This score reflects the weight of the features in shifting the classifier decision from the original class to the targeted class. Perturbing the features with the highest adversarial saliency scores allows applying minimal perturbation required to generate the AEs. The adversary starts from computing the jacobian of the model and evaluate it against the model's input, the jacobian is defined as $[\frac{\partial f_j}{\partial x_i}\,(\overrightarrow{x})]_{i,j}$, where $i, j$ denotes to the derivative of the input and the class associated with it, respectively. In order to compute the adversarial saliency map, for each feature, the adversary computes $\{0\quad if\ s_t<0\ or\ s_o<0,\quad s_t|s_o|\ otherwise\}$ where $s_t$ represents $\frac{\partial f_t}{\partial x_i}\,(\overrightarrow{x})$ and $s_o$ denotes $\sum_{j\neq t}\frac{\partial f_j}{\partial x_i}\,(\overrightarrow{x})$, where t shows the targeted class of the adversary.

**Momentum Iterative Method (MIM).** MIM generates AEs using momentum-based iterative algorithm, by applying momentum gradient and providing techniques to escape from poor local maximum during the iterations [31]. The main focus of MIM is to generate AE $x'$ that cause misclassification by satisfying $arg\max_{x'} \; J(x', y), \quad s.t. \; ||x' - x||_\infty \le \epsilon$, where $J$ denotes the loss function and $\epsilon$ is a scalar that represents the maximum allowed distortion to be applied to the original sample. The momentum gradient ($M_g$) is then calculated as $M_{g_{t+1}} = \mu M_{g_t} + \frac{\nabla_x J_\theta(x'_t, l)}{||\nabla_x J_\theta(x'_t, l)||}$, where $\nabla$ shows the gradient function and $\mu$ is the decay factor. Initially, $x'_0$ is the original input and $M_{g_0}$ is set to 0. Each iteration, $x'$ is updated as $x'_{t+1} = x'_t + \epsilon \cdot sign(M_{g_t} + 1)$. After $n$ iterations, $x'_{t+1}$ is returned as the AE for input $x$.

**Projected Gradient Descent (PGD) Method.** Madry *et al.* [32] proposed a PGD-based adversarial method to generate AE under minimized empirical risk with the trade-off of high perturbation cost. The model's empirical risk minimization ($ERM$) is defined as $\mathbb{E}_{(x,y)\sim D}[L(x, y, \theta)]$, where $L$ represents the loss function, $x$ shows the original sample and y is the original label. By modifying $ERM$ definition by allowing the adversary to perturb the input $x$ by the scalar value $S$, $ERM$ is represented by $\min_{\theta} \rho(\theta) : \rho(\theta) = \mathbb{E}_{(x,y)\sim D}[\max_{\delta \in S} L(x + \delta, y, \theta)]$, where $\delta$ is the applied perturbation and $\rho(\theta)$ denotes the objective function. Notice that $x'$ is updated in each iteration, after $n$ iterations, $x'$ is returned as the generated AE.

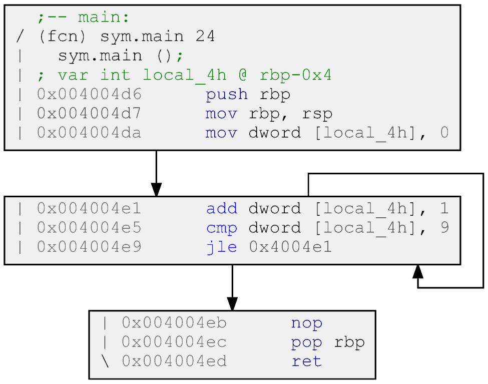
**Virtual Adversarial Method (VAM).** Miyato *et al.* [33] proposed VAM, an adversarial machine learning attack method to generate AEs based on the Local Distributional Smoothness (LDS). Assuming training set $D$, where $D = \{(x^{(n)}, y^{(n)}) \mid x^{(n)} \in R^I, y^{(n)} \in Q, n = 1, \ldots, N\}$, where $R^I$ represents the input space, $Q$ shows the output space, $x$ and $y$ are vectors representing the input samples and output labels. The distributions divergence is defined as $\Delta_{KL}(r, x^{(n)}, \theta) \equiv KL[p(y|x^{(n)}, \theta) \, || p(y|x^{(n)} + r, \theta)]$, where $\epsilon$ is the distortion threshold and $\theta$ represents a constant scalar value. Moreover, the perturbation optimization function $r$ is denoted by $r^{(n)}_{v-adv} \equiv arg\max_{r}\{\Delta_{KL}(r, x^{(n)}, \theta) \, ; \; ||r||_2 \le \epsilon\}$. $r^{(n)}_{v-adv}$ is referring to the virtual perturbation used to generate the AEs.

### B. Graph Embedding and Augmentation (GEA)

Besides validating adversarial learning attacks on graph-based IoT malware detection systems, another contribution in this work is the graph embedding and augmentation (GEA). The key goal of GEA is to generate realistic AEs that maintain the functionality of the original IoT software while also achieving high misclassification rate. The main insight of this approach is to combine the original graph with a selected target graph. We show that such a combination results in misclassification while preserving the functionality and practicality of the original sample. For better understanding of this approach, we elaborate the idea using an example.

Note that the concept of GEA can be used for practically realizing graphs corresponding to modifications introduced by any of the aforementioned eight approaches for AE generation.

**Practical Implementation.** Assume an original sample $x_{org}$ and a selected target sample $x_{sel}$, our main goal is to combine the two samples while preserving the functionality and practicality of $x_{org}$ and achieving misclassification. Note that a condition is set to execute only the functionality related to $x_{org}$ while preventing $x_{sel}$ functionality from being executed. Prior to generating the CFG for these samples, we compile the code using GNU Compiler Collection (GCC) command. Afterwards, Radare2 [34] is used to extract the CFG from the binaries. Figure 2 and Figure 3 show the generated graphs for $x_{org}$ and $x_{sel}$, respectively. As it can be seen, the combined graph in Figure 4 consists of the two aforementioned scripts sharing the same entry and exit nodes.

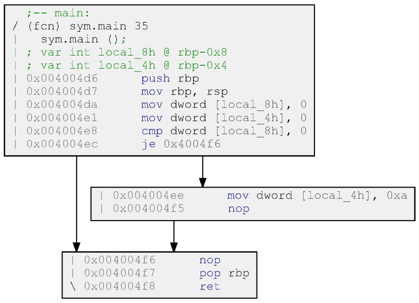


```
  ;-- main:
/ (fcn) sym.main 24
|   sym.main ();
| ; var int local_4h @ rbp-0x4
| 0x004004d6      push rbp
| 0x004004d7      mov rbp, rsp
| 0x004004da      mov dword [local_4h], 0

| 0x004004e1      add dword [local_4h], 1
| 0x004004e5      cmp dword [local_4h], 9
| 0x004004e9      jle 0x4004e1

| 0x004004eb      nop
| 0x004004ec      pop rbp
\ 0x004004ed      ret
```

Fig. 2. The generated CFG for the original sample and used for extracting graph-based features (graph size, centralities, etc.) for graph/program classification and malware detection.



```
  ;-- main:
/ (fcn) sym.main 35
|   sym.main ();
| ; var int local_8h @ rbp-0x8
| ; var int local_4h @ rbp-0x4
| 0x004004d6      push rbp
| 0x004004d7      mov rbp, rsp
| 0x004004da      mov dword [local_8h], 0
| 0x004004e1      mov dword [local_4h], 0
| 0x004004e8      cmp dword [local_8h], 0
| 0x004004ec      je 0x4004f6

| 0x004004ee      mov dword [local_4h], 0xa
| 0x004004f5      nop

| 0x004004f6      nop
| 0x004004f7      pop rbp
\ 0x004004f8      ret
```

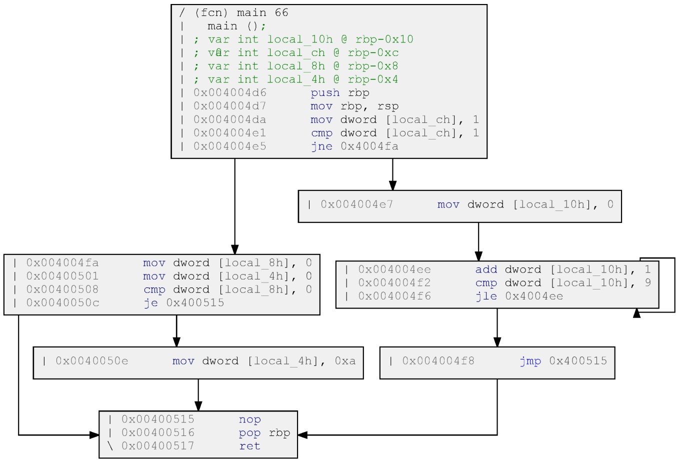
Fig. 3. The graph for the selected target sample generated as in Fig. 2.

**GEA Configuration.** We select six different graphs from the benign and malicious samples based on the graph size. Then, we combine them with the CFGs of the target class samples, through applying the concept described in the basic example subsection III-B, to produce the AEs.

## IV. Evaluation and Discussion

### A. Dataset

Alasmary *et al.* [9] contrast the Android and the IoT malware binaries by comparing the CFGs of the two malware types. We gather the CFG dataset of the IoT malware from them to assess our proposed approach. To facilitate the evaluation, we also gather a dataset of 276 benign IoT samples. Since the IoT devices run Linux distributions, therefore, the Linux kernel files can be executed on the IoT devices and can also be safely considered as benign files. With this on our mind, we visit the OpenWRT's [35] online repository. We then look for router firmware, we download the firmware and filter out



```
/ (fcn) main 66
|   main ();
| ; var int local_10h @ rbp-0x10
| ; var int local_ch @ rbp-0xc
| ; var int local_8h @ rbp-0x8
| ; var int local_4h @ rbp-0x4
| 0x004004d6      push rbp
| 0x004004d7      mov rbp, rsp
| 0x004004da      mov dword [local_ch], 1
| 0x004004e1      cmp dword [local_ch], 1
| 0x004004e5      jne 0x4004fa
```

```
| 0x004004e7      mov dword [local_10h], 0
```

```
| 0x004004fa      mov dword [local_8h], 0
| 0x00400501      mov dword [local_4h], 0
| 0x00400508      cmp dword [local_8h], 0
| 0x0040050c      je 0x400515
```

```
| 0x004004ee      add dword [local_10h], 1
| 0x004004f2      cmp dword [local_10h], 9
| 0x004004f6      jle 0x4004ee
```

```
| 0x0040050e      mov dword [local_4h], 0xa
```

```
| 0x004004f8      jmp 0x400515
```

```
| 0x00400515      nop
| 0x00400516      pop rbp
\ 0x00400517      ret
```

Fig. 4. GEA. This graph is obtained by embedding the graph in Fig. 3 into the graph in Fig. 2, although indirectly done by injecting code directly.

TABLE I
DISTRIBUTION OF IOT SAMPLES ACROSS THE CLASSES.

| Class types | # of Samples | % of Samples |
|---|---|---|
| Benign | 276 | 10.79% |
| Malicious | 2,281 | 89.21% |
| Total | 2,557 | 100% |

Linux executable files in the repositories. Having assembled a benign IoT dataset, we extract CFGs corresponding to each of the samples. To do so, we reverse engineer the binaries to extract the disassembled code. We then generate CFG from the disassembly for further processing. We summarize our dataset in table I. Moving ahead, we find different algorithmic features of the CFGs corresponding to individual binaries. In particular, for each sample, we extract 23 different algorithmic features categorized into seven groups. Table I represents the feature category and the number of features in each category. The five features extracted from each of the four feature categories represent minimum, maximum, median, mean, and standard deviation values for the observed parameters.

### B. Experimental Setup

We conducted all experiments on Ubuntu 18.04, using Python 3.6 with a system comprised of an i5-8500 CPU, 32GB DDR4 RAM, 512GB SSD and 4TB HHD storage and NVIDIA GTX980 Ti Graphics Processing Unit (GPU) used in deep learning processing.

*1) IoT Malware Detection System:* The goal of our detection system is to recognize IoT malicious applications from benign. Therefore, we trained a CNN-based model over the extracted CFG-based features. In this study, the input ($X$) of the model is a one dimensional (1D) vector of size $1\times23$ representing the extracted features. The model architecture consists of three blocks, namely convolutional block 1 (ConvB1), convolutional block 2 (ConvB2), and classification block (CB). In the following a detailed description of each block:

**ConvB1.** Fed by the input features vector ($X$), this block is made up of two convolutional layers, a 1D convolutional layer

TABLE II
DISTRIBUTION OF EXTRACTED FEATURES.

| Feature category | # of features |
|---|---|
| Betweenness centrality | 5 |
| Closeness centrality | 5 |
| Degree centrality | 5 |
| Shortest path | 5 |
| Density | 1 |
| # of Edges | 1 |
| # of Nodes | 1 |
| Total | 23 |

(Conv 1) with padding and 46 filters (${F_{b1}}'$) of size $1 \times 3$, convolving over the input data ($X$) with stride of 1, resulting in a 2D tensor (${C_{b1}}'$) of size $1 \times 23$. Followed by a similar 1D convolutional layer (Conv 2) without padding. The output of this layer is a 2D tensor (${C_{b1}}''$) of size $46 \times 21$. Then, a max pooling operation of size and stride of 2 and dropout with probability of 0.25 are applied. The output of this block is a 2D tensor $S_{b1}$ of size $46 \times 10$.

$$\begin{aligned}
{C_{b1}}_i' &= X \otimes {F_{b1}}_i', && i = 1:46\\
{C_{b1}}_i'' &= {C_{b1}}_i' \otimes {F_{b1}}_i'', && i = 1:46\\
{M_{b1}}_i &= \mathsf{maxpool}({C_{b1}}_i'', 2, 2), && i = 1:46\\
{S_{b1}}_i &= \mathsf{dropout}({M_{b1}}_i, 0.25), && i = 1:46
\end{aligned}$$

**ConvB2.** Fed by the previous block output ($S_{b1}$), this block is similar to ConvB1 with a difference in the number of filters (${F_{b2}}'$) in the convolutional layers. This block consists of 1D convolutional layer (Conv 3) with padding and 92 filters of size $1 \times 3$, convolving with a stride of 1. Followed by a similar 1D convolutional layer (Conv 4) without padding. The output of this layer is a 2D tensor ${C_{b2}}''$ of size $92 \times 8$. Afterwards, a max pooling operation of size and stide of 2 is applied, followed by a dropout with probability of 0.25, resulting into a 2D tensor $S_{b2}$ of size $92 \times 4$.

$$\begin{aligned}
{C_{b2}}_i' &= S_{b1} \otimes {F_{b2}}_i', && i = 1:92\\
{C_{b2}}_i'' &= C_b 1_i' \otimes {F_{b2}}_i'', && i = 1:92\\
{M_{b2}}_i &= \mathsf{maxpool}({C_{b2}}_i'', 2, 2), && i = 1:92\\
{S_{b2}}_i &= \mathsf{dropout}({M_{b2}}_i, 0.25), && i = 1:92
\end{aligned}$$

**CB.** The generated tensor in ConvB2 ($S_{b2}$) is then fed into this block. The input is forwarded to flatten operation, resulting in a 1D tensor ($F_l$) of size $1 \times 368$. Followed by a fully connected (FC) dense layer ($FCL$) of size 512 and a dropout with probability of 0.5 resulting into $S_{FC}$. Finally, $S_{FC}$ is fed to the softmax layer. The outputs of the softmax layer will be evaluated based on the accuracy rate (AR), false negative rate (FNR), and false positive rate (FPR), to measure the performance of the model.

$$\begin{aligned}
F_l &= \mathsf{Flatten}(S_{b2})\\
FCL &= \mathsf{dense}(F_l, 512)\\
S_{FC} &= \mathsf{dropout}(FCL, 0.5)\\
\text{output} &= \mathsf{softmax}(S_{FC})
\end{aligned}$$

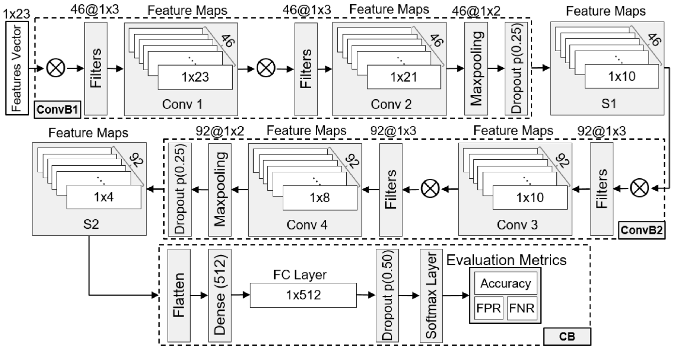


Fig. 5. The internal design of the CNN architecture and used for the detection task in this work. Notice that 46@1x3, for example, stands for applying 46 filters, each of size 1x3 on the input data.

We trained our model using 200 epochs with a batch size of 100. Note that all convolutional and fully connected layers use a Rectified Linear Units (ReLU) activation function. In addition, we used dropout to prevent model over-fitting and max pooling operation to make the features more distinct and to reduce the amount of data. The architecture of the CNN design is shown in Figure 5. We refer the interested reader to [16] for more details on CNN internals.

*2) Generic Adversarial Attacks:* The main focus of the adversary is to craft AEs that fool the IoT malware detector. Generic adversarial machine learning attacks apply small perturbation to the features vector to craft the AEs. In this approach, we check whether the generated AE features are realistic. For instance, if the values of the features can be observed from real samples. All attacks are implemented using Cleverhans [36], a Python library for adversarial machine learning attack methods. Each of these attacks has various variables to be set carefully for best performance. The configuration for each of these attacks is described in the following.

**C&W Attack.** AEs crafted based on C&W $L_2$ distance by setting the learning rate to 0.1 and 200 iterations.

**DeepFool Attack.** DeepFool minimizes the $L_2$ distance between the AE and the original sample, we crafted the AEs using overshooting value of 0.02 with 100 iterations.

**ElasticNet Attack.** ElasticNet generates the AEs based on $L_1$ distance using the same loss function of C&W attack. We generated the AEs by setting the number of iterations to 250 with a learning rate of 0.1.

**FGSM Attack.** As a fast one iteration method, it generates AEs by adding perturbation bounded by the value of $\epsilon$. In the crafting process, we set the distortion threshold ($\epsilon$) to 0.3.

**JSMA Attack.** This method is based on $L_0$ distance, and minimizing $L_0$ distance leads to reducing the number of features changed in the process of generating the AE. To craft the AE using JSMA, we set $\theta$ and $\gamma$ to 0.3 and 0.6, respectively. In the crafting process, we insure that the applied changes can be achieved by manipulating the original graph, which can be done by carefully adding new nodes and edges to the original graph.

**MIM Attack.** This method generates AE by using momentum gradient-based algorithm. To craft the AEs we set the value of epsilon ($\epsilon$) to 0.3 with 10 iterations.

**PGD Attack.** PGD trades the perturbation cost with the empirical risk, by maximizing the perturbation cost while minimizing the empirical risk in the crafting process. The AEs is generated by setting the number of iterations to 40 with epsilon ($\epsilon$) value of 0.3.

**VAM Attack.** It generates the AEs based on LDS, we crafted the AEs by setting epsilon ($\epsilon$) value to 0.3 with 40 iterations.

*3) GEA:* While previous methods manipulate the features vector to generate the AEs, GEA approach modifies the CFG to craft the AEs. Due to the nature of the extracted features, the applied change on the CFG will be reflected to the features, regardless to the effects on the functionality and practicality of the original sample. To evaluate our approach, we selected six different sized graphs from benign and malware samples as $x_{sel}$. The selected graphs vary in the size: minimum, median or max graph size, where the size is referred to as the number of nodes in the graph. To generate AEs, we connected each selected graph with all samples from the opposite class using shared entry and exit nodes as illustrated in the basic example in subsection III-B. Then, to study the effect of changing number of edges on the misclassification rate, we fixed the number of nodes and selected three samples with different number of edges as $x_{sel}$ in the AEs generation process.

### *C. Results & Discussion*

In order to provide a better understanding of our findings, this section is broken down into three sections: deep learning-based IoT malware detection system, generic adversarial machine learning attacks approach, and GEA approach. In the following, the findings are discussed in the associated sections.

*1) Deep Learning-based IoT Malware Detection System:* The goal of this study is to investigate the robustness of deep learning-based model trained over CFG-based features against adversarial attacks. Therefore, we designed a two class classification CNN-based model that distinguishes IoT malware from IoT benign applications. The model is trained over 23 CFG-based features categorized in seven groups. More detailed information regarding the dataset is provided in subsection IV-A. We achieved an accuracy rate of 97.13% with a False Negative Rate (FNR) of 11.26% and False Positive Rate (FPR) of 1.55%. It is worth mentioning that the high value of FNR is due to the imbalanced number of malware and benign samples in the dataset.

*2) Generic Adversarial Learning Attacks:* We implemented eight generic adversarial machine learning attack methods to generate AE by perturbing the feature space. Overall, those approaches have shown, in general, a good performance. In particular, We were able to generate AEs that misclassify malicious IoT applications as benign applications and vice versa. Our experiments demonstrate that we can achieve misclassification rate as high as 100%.

Detailed misclassification rate results of each attack method are in Table III. Note that FGSM and VAM have low misclassification rate, as FGSM search function cannot escape the local minimum, increasing the average number of features changed and lowering the misclassification rate. Moreover, VAM is an iterative method inspired from FGSM, which explains the

TABLE III

EVALUATION USING GENERIC METHODS. MR=MISCLASSIFICATION RATE, AVG.FG=AVERAGE NUMBER OF FEATURES CHANGED TO GENERATE AE, AND CT=CRAFTING TIME IN MILLISECONDS PER SAMPLE.

| Attack Method | MR (%) | Avg.FG | CT (ms) |
|---|---|---|---|
| C&W | 100 | 12.60 | 25.30 |
| DeepFool | 86.39 | 14.90 | 2.56 |
| ElasticNet | 100 | 5.42 | 114.18 |
| FGSM | 25.84 | 23 | 0.37 |
| JSMA | 99.80 | 4.00 | 0.78 |
| MIM | 100 | 20.60 | 0.90 |
| PGD | 100 | 22.56 | 2.40 |
| VAM | 28.80 | 16.64 | 16.58 |

TABLE IV

GEA: MALWARE TO BENIGN MISCLASSIFICATION RATE. MR REFERS TO MISCLASSIFICATION RATE, WHEREAS, CT REFERS TO THE CRAFTING TIME IN MILLISECOND PER SAMPLE.

| Size | # Nodes | MR (%) | CT (ms) |
|---|---|---|---|
| Minimum | 2 | 7.67 | 33.69 |
| Median | 24 | 95.48 | 37.79 |
| Maximum | 455 | 100 | 1,123.12 |

TABLE V

GEA: BENIGN TO MALWARE MISCLASSIFICATION RATE. MR REFERS TO MISCLASSIFICATION RATE, WHEREAS, CT REFERS TO THE CRAFTING TIME IN MILLISECOND PER SAMPLE.

| Size | # Nodes | MR (%) | CT (ms) |
|---|---|---|---|
| Minimum | 1 | 30.65 | 40.65 |
| Median | 64 | 57.60 | 69.23 |
| Maximum | 367 | 88.04 | 473.91 |

TABLE VI

GEA: MALWARE TO BENIGN MISCLASSIFICATION RATE WITH FIXED NUMBER OF NODES. MR REFERS TO MISCLASSIFICATION RATE, WHEREAS, CT REFERS TO THE CRAFTING TIME IN MILLISECOND PER SAMPLE.

| # Nodes | # Edges | MR (%) | CT (ms) |
|---|---|---|---|
| 8 | 7 | 13.72 | 33.84 |
|  | 9 | 13.10 | 34.09 |
|  | 10 | 13.10 | 34.17 |
| 33 | 46 | 94.78 | 40.17 |
|  | 50 | 57.47 | 42.43 |
|  | 53 | 95.74 | 40.79 |
| 63 | 91 | 11.48 | 49.39 |
|  | 93 | 22.84 | 56.31 |
|  | 95 | 8.37 | 63.30 |

improvement in the performance. Whereas, JSMA achieves a misclassification rate of 99.80% with manipulating four features, which include the number of nodes and edges. JSMA requires the least amount of changes to the graph topology to achieve the modifications on the feature space. Nonetheless, the generated new graph does not necessarily preserve the practicality and functionality of the original sample.

*3) GEA:* This approach is designed to generate a practical AE that fools the classifier, while preserving the functionality and practicality of the original sample. Here, we discuss the inherent overhead of the GEA approach. We investigate the impact of the size of the graph, determined by the number of the nodes in a graph, and graph density, determined by the number of edges in a graph while the number of nodes is fixed. Note that all generated samples maintain the practicality and the functionality of the original sample. The obtained results are discussed in more detail in the following.

**Graph Size Impact.** We selected three graphs, as targets, from each of the benign and malicious IoT software, and connected each of these target graphs with a graph of the other class. The target graphs consist of a minimum, median and maximum graph size, and the goal was to understand the impact of size on misclassification with GEA. The results are shown in Table IV and Table V. As it can be seen in Table IV, three benign samples were selected, whereas in Table V three malicious samples were selected. One of the key findings we observed is that the misclassification rate increases when the number of nodes increases, which is perhaps natural. In addition, the time needed to craft the AE is proportional to the size of the selected sample. We achieved a malware to benign misclassification rate of as high as 100%, and a benign to malware misclassification rate of 88.04%, while insuring that the original samples are executed as intended, a property not guaranteed with the eight off-the-shelf approaches above.

**Graph Density Impact.** We fixed the number of nodes and selected graphs with different number of edges. Afterwards, we generated the AEs using GEA approach. Detailed results can be found in Table VI and Table VII. Note that we could not observe any meaningful relationship between the number of edges and the misclassification rate. Rather we observed that the misclassification rate is highly dependent on the confidence of the classifier in classifying the selected sample.

## V. RELATED WORK

Alasmary *et al.* [9] conducted an in-depth CFG-based comparative study for the Android and IoT malware. Pa *et al.* [37] established the first IoT honeypot and sandbox system, called IoTPOT, that run over eight CPU architectures to capture the IoT attacks running over Telnet protocol. Caselden *et al.* [38] built an algorithm that generates an attack from the representation of the hybrid information and CFG applied to the program binaries. Alam *et al.* [39] proposed a metamorphic malware analysis and detection system that uses two different techniques that match the CFGs of small malware and then address the change in the opcodes frequencies. Moreover, Tamersoy *et al.* [40] proposed a malware detection algorithm that identifies the executable files of the malware and then computes the similarities between them to partial dataset files from the Norton Community Watch. Then, they construct graphs based on the measurement of inter-relationship between these files. In addition, Wuchner *et al.* [41] proposed a graph-based detection system that uses a quantitative data flow graphs generated from the system calls, and use the graph node properties, i.e. centrality metric, as a feature vector for the classification between malicious and benign programs.

Some work has been done toward analysis and detection of the Android applications from the lens of CFG. For example, ManXu *et al.* [42] proposed a CNN-based malware detection system for the Android application from the semantic representation of the graph, the CFG and DFG. In addition, Yang *et*

TABLE VII
GEA: BENIGN TO MALWARE MISCLASSIFICATION WITH FIXED NODES. MR REFERS TO MISCLASSIFICATION RATE, WHEREAS, CT REFERS TO THE CRAFTING TIME IN MILLISECOND PER SAMPLE.

| # Nodes | # Edges | MR (%) | CT (ms) |
|---|---|---|---|
| 15 | 16 | 67.02 | 40.03 |
|  | 18 | 41.66 | 40.97 |
|  | 20 | 40.21 | 41.16 |
| 57 | 74 | 86.59 | 47.31 |
|  | 84 | 56.52 | 61.34 |
|  | 91 | 55.79 | 57.31 |
| 71 | 96 | 75.00 | 54.02 |
|  | 100 | 63.04 | 69.71 |
|  | 110 | 49.27 | 65.75 |

*al.* [43] identified and detect Android malicious behaviors throughout generating two level behavioral representations built from the CFG graph and call graphs of the program. Allix *et al.* [44] designed multiple machine learning classifiers to detect the Android malware using different textual representation extracted from the applications' CFGs.

### *A. Adversarial Machine Learning*

Machine/deep learning networks are widely used in security-related tasks, including malware detection [9]–[11], [19]. However, it has been shown that deep learning-based models are vulnerable against adversarial attacks [20], [33]. Given that, it should be noted that such a behavior can be a critical issue in malware detection systems, where misclassifying a malware as benign may result in disastrous consequence [45]. Various adversarial machine learning attack methods have been introduced to generate AEs. For example, Goodfellow *et al.* [22] introduced FGSM, a family of fast method attacks to generate AEs that forces the model to misclassification. In addition, Carlini *et al.* [29] proposed three L-norm-based adversarial attacks, known as C&W adversarial attacks, to investigate the robustness of neural networks and existing adversarial defenses. Similarly, Moosavi *et al.* [21] proposed DeepFool, an $L_2$ distance-based adversarial iterative method to generate AEs with minimal perturbation. Moreover, Madry *et al.* [32] presented PGD-based adversarial method that forces the model to misclassification by increasing the $L_2$ distance between the original and generated sample. Furthermore, Dong *et al.* [31] proposed MIM, a momentum-based algorithm to generate white-box and black-box AEs. Likewise, Chen *et al.* [30] introduced ElasticNet, an $L_1$ distance-based adversarial method to generate AEs.

## VI. LIMITATION AND FUTURE WORK

In this study, we implemented two approaches to generate AEs. Off-the-shelf attack methods apply perturbation directly to the feature space. In order to analyze the practicality and functionality of the generated sample, perturbation should be reflected to the CFG of the original sample. Meanwhile, GEA solves this issue by applying the perturbation to the CFG directly, by carefully connecting the original graph with a selected sample while preserving the practicality and functionality of the original sample. Nonetheless, generating AE using GEA will increase the size of the original sample. Our findings indicate that the accuracy of the GEA approach highly correlates with the size of selected sample.

Malware authors often use different packing techniques, *e.g.,* Ultimate Packer for Executables (UPX), to obfuscate different parts of the malware code base, such as functions and strings. In obfuscated functions, the CFG would differ from the actual unpacked malware. Although, the packed malware samples give an attacker a success rate of 100%, it would be interesting to examine the behavior of unpacked malware samples exposed to similar attacks. For future work, we would investigate this attack scenario. Additionally, we would also investigate more effective methods to minimize the size of the generated AEs, while preserving the main characteristics, such as fooling the classifier and preserving the functionality and practicality as original software, *etc.*

## VII. CONCLUSION

This work studies the robustness of graph-based deep learning models against adversarial machine learning attacks. To set out, first an in-depth analysis of malware binaries is conducted through constructing abstract structures using CFG, which are analyzed from multiple aspects, such as number of nodes and edges, as well as graph algorithmic constructs, such as average shortest path, betweenness, closeness, density, etc. Then two different approaches are designed to generate AEs, including off-the-shelf adversarial attack methods and GEA approach, while applying small perturbation to the graph features. We examined eight different well-established adversarial learning techniques to force the model to misclassification. Although, this approach achieves high misclassification rate, it does not guarantee the practicality and functionality of the crafted AEs. Whereas, GEA approach not only preserves the functionality and practicality of the original sample, but also achieves high misclassification rate. The performance of the proposed method is validated through intensive experiments. We were able to generate AEs that lead to 100% misclassification rate using generic adversarial learning approach. In addition, we observed that GEA is able to misclassify all malware samples as benign ones, highlighting the need for more robust IoT malware detection tools against adversarial learning.

**Acknowledgement.** This work is supported by NSF grant CNS-1809000, NRF grant 2016K1A1A2912757, NVIDIA GPU Grant Program (2018 and 2019), and a Cyber Florida Seed Grant. The authors would like to thank Thang N. Dinh and Saeed Salem for their feedback on an earlier draft.

## REFERENCES

[1] A. Gerber. (Retrieved, 2017) Connecting all the things in the Internet of Things. [Online]. Available: https://ibm.co/2qMx97a

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the Mirai Botnet," in *Proceedings of the 26th USENIX Security Symposium*, 2017, pp. 1093–1110.

[3] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash, "ContexIoT: Towards providing contextual integrity to appified IoT platforms," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium, NDSS*, 2017, pp. 1–15.

[4] A. Azmoodeh, A. Dehghantanha, and K.-K. R. Choo, "Robust malware detection for Internet Of (Battlefield) Things devices using deep eigenspace learning," *IEEE Transactions on Sustainable Computing*, vol. 4, no. 1, pp. 88–95, 2019.

[5] S. Siby, R. R. Maiti, and N. O. Tippenhauer, "IoTScanner: Detecting privacy threats in IoT neighborhoods," in *Proceedings of the 3rd ACM International Workshop on IoT Privacy, Trust, and Security*, 2017, pp. 23–30.

[6] A. Mohaisen, H. Tran, N. Hopper, and Y. Kim, "On the mixing time of directed social graphs and security implications," in *7th ACM Symposium on Information, Computer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012*, 2012, pp. 36–37.

[7] A. Mohaisen, A. Yun, and Y. Kim, "Measuring the mixing time of social graphs," in *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference, IMC 2010, Melbourne, Australia - November 1-3, 2010*, 2010, pp. 383–389.

[8] A. Mohaisen, N. Hopper, and Y. Kim, "Keep your friends close: Incorporating trust into social network-based sybil defenses," in *INFOCOM 2011. 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 10-15 April 2011, Shanghai, China*, 2011, pp. 1943–1951.

[9] H. Alasmary, A. Anwar, J. Park, J. Choi, D. Nyang, and A. Mohaisen, "Graph-based comparison of IoT and android malware," in *Proceedings of the 7th International Conference on Computational Data and Social Networks, CSoNet*, 2018, pp. 259–272.

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

[11] A. Mohaisen and O. Alrawi, "Unveiling zeus: automated classification of malware samples," in *Proceedings of the 22nd International World Wide Web Conference, WWW*, 2013, pp. 829–832.

[12] B. Alipanahi, A. Delong, M. T. Weirauch, and B. J. Frey, "Predicting the sequence specificities of DNA-and RNA-binding proteins by deep learning," *Nature biotechnology*, vol. 33, no. 8, p. 831, 2015.

[13] E. Knorr. (Retrieved, 2015) How PayPal beats the bad guys with machine learning. [Online]. Available: https://tinyurl.com/y8k3hfr7

[14] A. Khormali and J. Addeh, "A novel approach for recognition of control chart patterns: Type-2 fuzzy clustering optimized support vector machine," *ISA transactions*, vol. 63, pp. 256–264, 2016.

[15] A. Addeh, A. Khormali, and N. A. Golilarz, "Control chart pattern recognition using rbf neural network with new training algorithm and practical features," *ISA transactions*, vol. 79, pp. 202–216, 2018.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proceedings of the 26th Annual Conference on Neural Information Processing Systems NIPS*, 2012, pp. 1106–1114.

[17] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proceedings of the 24th USENIX Security Symposium*, 2015, pp. 611–626.

[18] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013, pp. 3422–3426.

[19] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon, "From throw-away traffic to bots: Detecting the rise of DGA-based malware," in *Proceedings of the 21th USENIX Security Symposium*, 2012, pp. 491–506.

[20] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016, pp. 372–387.

[21] S. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "DeepFool: A simple and accurate method to fool deep neural networks," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2574–2582.

[22] C. S. Ian J. Goodfellow, Jonathon Shlens, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations.*, 2015, pp. 1–11.

[23] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel, "Adversarial examples for malware detection," in *22nd European Symposium on Research Computer Security*, 2017, pp. 62–79.

[24] Q. Zhang and D. S. Reeves, "Metaaware: Identifying metamorphic malware," in *Proceedings of the Twenty-Third Annual Computer Security Applications Conference, ACSAC*, 2007, pp. 411–420.

[25] Developers. (Retrieved, 2018) the ultimate packer for executables. [Online]. Available: https://upx.github.io/

[26] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security & Privacy*, vol. 5, no. 2, pp. 32–39, 2007.

[27] N. Papernot, P. D. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the ACM on Asia Conference on Computer and Communications Security, AsiaCCS*, 2017, pp. 506–519.

[28] B. Wang, Y. Yao, B. Viswanath, H. Zheng, and B. Y. Zhao, "With great training comes great vulnerability: Practical attacks against transfer learning," in *Proceedings of the 27th USENIX Security Symposium, USENIX Security 2018*, 2018, pp. 1281–1297.

[29] N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2017, pp. 39–57.

[30] P. Chen, Y. Sharma, H. Zhang, J. Yi, and C. Hsieh, "EAD: elastic-net attacks to deep neural networks via adversarial examples," in *Proceedings of the Conference on Artificial Intelligence*, 2018, pp. 10–17.

[31] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li, "Boosting adversarial attacks with momentum," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2018, pp. 9185–9193.

[32] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations.*, 2018, pp. 1–27.

[33] T. Miyato, S.-i. Maeda, M. Koyama, K. Nakae, and S. Ishii, "Distributional smoothing with virtual adversarial training," in *International Conference on Learning Representations.*, 2016, pp. 1–12.

[34] Developers. (Retrieved, 2019) Radare2. [Online]. Available: http://www.radare.org/r/

[35] Developers. (Retrieved, 2019) OpenWrt project. [Online]. Available: https://openwrt.org

[36] N. Papernot, N. Carlini, I. Goodfellow, R. Feinman, F. Faghri, A. Matyasko, K. Hambardzumyan, Y.-L. Juang, A. Kurakin, R. Sheatsley *et al.*, "cleverhans v2. 0.0: an adversarial machine learning library," *arXiv preprint arXiv:1610.00768*, 2016.

[37] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "IoTPOT: A novel honeypot for revealing current IoT threats," *Journal of Information Processing JIP*, vol. 24, no. 3, pp. 522–533, 2016.

[38] D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, and D. Song, "HI-CFG: construction by binary analysis and application to attack polymorphism," in *Proceedings of the 18th European Symposium on Research in Computer Security*. Springer, 2013, pp. 164–181.

[39] S. Alam, R. N. Horspool, I. Traoré, and I. Sogukpinar, "A framework for metamorphic malware analysis and real-time detection," *Computers & Security*, vol. 48, pp. 212–233, 2015.

[40] A. Tamersoy, K. A. Roundy, and D. H. Chau, "Guilt by association: large scale malware detection by mining file-relation graphs," in *Proceedings of the the 20th ACM International Conference on Knowledge Discovery and Data Mining, KDD*, 2014, pp. 1524–1533.

[41] T. Wüchner, M. Ochoa, and A. Pretschner, "Robust and effective malware detection through quantitative data flow graph metrics," in *Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment Conference, DIMVA*, 2015, pp. 98–118.

[42] Z. Xu, K. Ren, S. Qin, and F. Craciun, "CDGDroid: Android malware detection based on deep learning using CFG and DFG," in *Proceedings of the 20th International Conference on Formal Engineering Methods, ICFEM*, 2018, pp. 177–193.

[43] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. A. Porras, "DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in android applications," in *Proceedings of the 19th European Symposium on Research in Computer Security*, 2014, pp. 163–182.

[44] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. L. Traon, "Empirical assessment of machine learning-based malware detectors for android - measuring the gap between in-the-lab and in-the-wild validation scenarios," *Empirical Software Engineering*, vol. 21, no. 1, pp. 183–211, 2016.

[45] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, U. Meteriz, and A. Mohaisen, "Breaking graph-based IoT malware detection systems using adversarial examples, poster," in *Proceedings of the 12th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec*, 2019.