Soteria: Detecting Adversarial Examples in Control Flow Graph-based Malware Classifiers

Hisham Alasmary[†‡⋆], Ahmed Abusnaina[†⋆], Rhongho Jang[†⋆],
Mohammed Abuhamad[†], Afsah Anwar[†], DaeHun Nyang[¶], and David Mohaisen[†]

[†]University of Central Florida [‡]King Khalid University [¶]Ewha Womans University
hisham, ahmed.abusnaina, r.h.jang, abuhamad, afsahanwar @knights.ucf.edu; nyang@ewha.ac.kr; mohaisen@ucf.edu

***Abstract*—Deep learning algorithms have been widely used for security applications, including malware detection and classification. Recent results have shown that those algorithms are vulnerable to adversarial examples, whereby a small perturbation in the input sample may result in misclassification. In this paper, we systematically tackle the problem of adversarial examples detection in the control flow graph (CFG) based classifiers for malware detection using Soteria. Unique to Soteria, we use both density-based and level-based labels for CFG labeling to yield a consistent representation, a random walk-based traversal approach for feature extraction, and n-gram based module for feature representation. End-to-end, Soteria's representation ensures a simple yet powerful randomization property of the used classification features, making it difficult even for a powerful adversary to launch a successful attack. Soteria also employs a deep learning approach, consisting of an auto-encoder for detecting adversarial examples, and a CNN architecture for detecting and classifying malware samples. We evaluate the performance of Soteria, using a large dataset consisting of 16,814 IoT samples, and demonstrate its superiority in comparison with state-of-the-art approaches. In particular, Soteria yields an accuracy rate of 97.79% for detecting AEs, and 99.91% overall accuracy for classification malware families.**

***Index Terms*—Internet of Things; Adversarial Machine Learning; Malware Detection; Deep Learning**

I. Introduction

The rising acceptance of IoT devices for different industrial and personal applications has been paralleled with a proportionally increase to their susceptibility to attacks. A major reason for their susceptibility to take is their use of vulnerable or insecure functions and services. As such, adversaries exploit these vulnerable services to deliver malware and to launch orchestrated attacks. This makes malware detection an important issue. To address this issue, several prior works have leveraged different machine- and deep-learning algorithms for malware detection atop program analysis [1], [2], [3], [4], [5], [6]. Program analysis approaches utilized for malware include both static and dynamic analyses. Dynamic analysis require executing malware for obtaining behavior features that are fed into machine learning algorithms for detection. Although the dynamic features are comprehensive, dynamic analysis techniques are subject to various shortcomings, and most importantly their complexity and time consumption, resulting in poor scalability. Static analysis, on the other hand, does not require running programs, but relies on programs contents, obtained from the static binary. A popular static analysis technique is using Control Flow Graph (CFG) to build a representative feature modality for malware detection, and is shown to be effective in various studies [7], [8]. Machine learning algorithms are typically implemented atop of the features extracted from the static (and dynamic) analysis techniques. However, these algorithms are susceptible to adversarial attacks, thereby circumventing such detection systems [9]. Therefore, it is essential to detect such attacks. To this end, this work proposes Soteria, a system to defeat the adversarial example attacks on CFG based classifiers for malware detection.

Given that ML models' output depends on the input patterns, ML models can be prone to targeted attacks on their inputs. Particularly, an adversary may fool the models by applying perturbations to the input to generate adversarial examples (AEs) that have similar characteristics with the original sample. As such, recent works have examined the robustness of the machine learning models in general, and have demonstrated the generation of AEs using methods such as the fast gradient sign method [10], generative adversarial networks [11], the DeepFool method [12], and the graph-based adversarial learning [9], among others. Nevertheless, there have been several attempts to defend against the adversarial attacks on the machine learning models by including the adversarial examples in the training process [13]. Although prior works have shown the inefficiency of the malware detection models when subjected to adversarial examples, to the best of our knowledge, there is no work on defending such models from adversarial attacks. Identifying the research gap, with this work, we inch closer towards bridging the gap.

Adversarial attacks on malware detectors have recently been conducted by Abusnaina *et al.* [9], Kolosnjaji *et al.* [14], and Kreuk *et al.* [15]. While Abusnaina *et al.* [9] show the susceptibility of the CFG-based detectors, the other works append bytes to the binary file. Both of these methods change the files while preserving the practicality and functionality of the clean IoT malware. Additionally, the AE creation of malware is limited due to the risk of un-executability. Acknowledging the importance of having effective defense to detect adversarial examples, Soteria utilizes features from

⋆The first three authors contributed equally to this work. ‡ This work was done while the author was at the University of Central Florida.

the CFG to detect them. Particularly, Soteria consists of two major components, the AEs detector and the IoT malware classifier. Soteria starts by labeling the CFG nodes based on two approaches: density-based labeling and level-based labeling. Then, Soteria applies a set of random walks, with a length proportional to the number of nodes in the CFG, on every labeling approach to deeply express and represent the behaviors of the software processes manifested in the CFG. The nodes making up the random walks are then used as the features for the operation of Soteria. In the first phase, the detection system that uses the deep features from the CFG to detect the AEs, thereby stopping their access to the malware classifier with an accuracy of 97.79%. In the next phase, with a flexibility to re-use the feature-set from the detection phase, it classifies the input file as benign or assigns an appropriate family label to the malware with an accuracy of 99.91%.

Contributions. In this paper, we make two contributions:

❶ Motivated by the recent body of work on developing adversarial examples on machine learning-based malware detection algorithms, we propose the design and implementation of Soteria, a system for detecting IoT malware. Similar to other efforts in this space, Soteria utilizes Control Flow Graph (CFG) based feature representations. Unique to Soteria, we use both density-based and level-based labels for CFG labeling, a random walk-based traversal approach for feature extraction, and n-gram based module for feature representation. End-to-end, Soteria's representation ensures a simple yet powerful randomization property of the used classification features, making it difficult even for a powerful adversary to launch a successful attack. Soteria also employs a deep learning approach, consisting of an auto-encoder for detecting AEs, and eliminating them from the classification process, and a CNN architecture for detecting and classifying malware samples.

❷ We evaluate the performance of Soteria, using a large dataset consisting of 16,814 IoT samples, and demonstrate its superiority in comparison with state-of-the-art approaches. Soteria yields an accuracy rate of 97.79% for detecting AEs, and 99.91% overall accuracy for classification malware families.

Organization. The rest of this paper is organized as follows. We introduce our motivation in section II, including practical adversarial example manipulation, limitation of adversarial learning, Graph Embedding, and Augmentation approach, and the threat model. We describe the system design in section III. We analyze and evaluate Soteria in section IV and discuss the results in section V. We review the literature in section VI, and conclude our study in section VII.

II. Background and Motivation

Adversarial examples (AEs) can be generated by slightly manipulating a sample to fool the classifier, and done in the context of malware on either the binary or the code level. ❶ **Binary-level AEs** the generation of such AEs entails manipulating the bytes of the malware sample upon compilation, without any regard to the function and purpose of such bytes, as has been done in several works [16], [17], [18]. Another method for binary-level AEs generation would entail injecting a benign block of bytes into an unreachable part of the malware binary, e.g., by adding a new section or appending the benign bytes to the end of malicious code, thus altering the feature representation introduced by the AE. ❷ **Code-level AEs** the generation of those AEs entails applying perturbation over the original code by either modifying the structure of the code or inserting an external code into it. For instance, augmenting or splitting functions results in a structure modification, thus altering the resulting feature space representation of the sample (e.g., CFG-based).

A. Practical Adversarial Examples

For adversarial attacks against machine learning-based malware detection models to be practical, adversaries must ensure the AEs resulting from the manipulation of a malware sample should still be executable (undamaged), making many of the algorithms proposed in the literature for AEs generation impractical for the malware detection domain. To this end, AEs can be categorized into impractical and practical AEs.

Impractical Adversarial Example. An AE is impractical if the injected code is compiled as an unused function during the compilation process. In the binary level, a sample that manipulated by any form of byte injection (*e.g.*, adding a new section or appending at the end of file) is not considered as the practical adversarial example.

Practical Adversarial Example. A practical AE is a mixture of the benign and/or malicious functions where the manipulated components are reachable (part of the code flow) and executable (do not damage the code).

Both code- and binary-level approaches can be used for generating practical AEs, although binary-level approaches are difficult to apply for fine-grained perturbations. A recent study [9] showed that adding external code to the original one leads to a high misclassification rate of the model's outputs while preserving the functionality of the original code. Such behavior can be critical as it results in changing the source code, execution flow, signature, and binaries, which reduces the performance of state-of-the-art classifiers. In this study, we focus on the injection of external code as a capability for creating AEs, since such an approach affects various representations of the original samples (See section II-D).

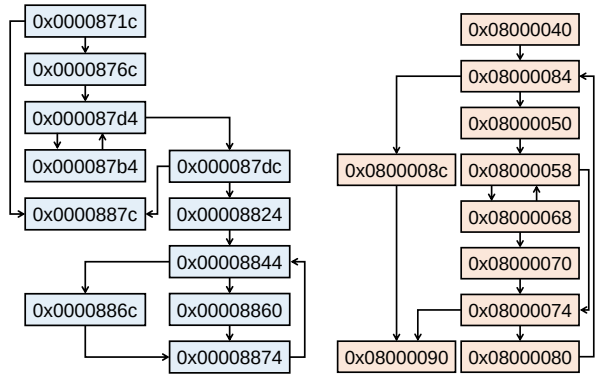
B. Limitation of Adversarial Learning

Adversarial training is a defense to increase the learning model's robustness by training over clean and adversarial datasets. Per Table I, this technique is used for enhancing the robustness of image classifiers by perturbing the training data.

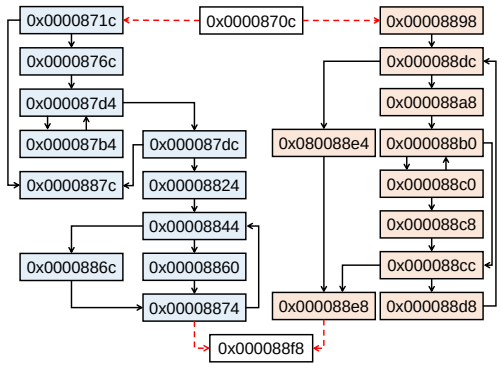
Drawbacks. A large number of studies on adversarial learning were implemented to generate AEs by perturbing the feature space, typically an image. Training a model over AEs generated by one method may not increase its robustness against other methods. This highlights the problem of adversarial learning, training against a set of methods does not guarantee the robustness against different attacks. This problem becomes critical with the existence of code-level manipulation. Where an adversary can change the outcome of the attack by changing

TABLE I: Current adversarial attacks defenses. The focus of the adversarial attacks defenses is on adversarial examples in the context of image classification. Note: MLP is Multilayer Perceptron, DNN is Deep Neural Network, and RNN is Recurrent Neural Network.

Paper	Model	Dataset	Application
Goodfellow *et al.* [19]	MLP	130,000+	Image
Xu *et al.* [20]	DNN	6,000	Image
Meng *et al.* [21]	DNN	121,000	Image
Liao *et al.* [22]	DNN	280,000	Image
Dhillon *et al.* [23]	DNN	60,000	Image
Papernot *et al.* [24]	DNN	130,000	Image
Samangouei *et al.* [25]	DNN	70,000	Image
Miyato *et al.* [26]	RNN	805,753	Text



(a) Original graph (b) Targeted graph



(c) GEA generated graph

Fig. 1: GEA adversarial approach. The CFG in Fig. 1(c) is obtained by embedding the CFG of a selected sample in Fig. 1(b) into the CFG of the original sample in Fig. 1(a). It can be done by injecting code directly.

the embedded code, or slightly changing the attack method *et al.* [9]; e.g., an adversary can decide which portion of the code he wants to execute. Therefore, adversarial benign and malicious samples can co-exist within the same feature space, calling for methods to detect AEs, by only relying on the structure of the clean sample.

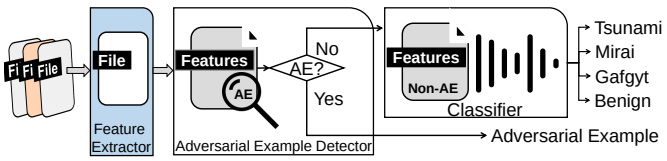


Fig. 2: Architecture of Soteria. IoT samples are fed to the feature extraction process, where each sample is represented by multiple feature vectors. The feature vectors are forwarded to adversarial example detector. All non-AEs are then forwarded to the classifier to be classified into its corresponding family.

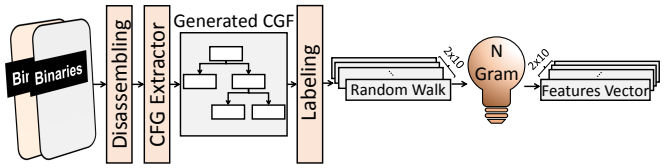


Fig. 3: Soteria feature extraction process. IoT samples binaries are disassembled to extract their corresponding CFGs. Then, two nodes labeling techniques are used (Dense-based and level-based), then, several random walks are done over each labeled graph. The trace of the random walk is then used for feature extraction by using n-grams with TF-IDF.

C. Graph Embedding and Augmentation

Graph Embedding and Augmentation (GEA) is an approach [9] that is shown to produce AEs that are executable, while allowing targeted adversarial attacks, addressing limitations of the literature. GEA inserts benign code into a target malware sample to generate an AE, with different feature representations. GEA applies direct modifications to the CFG of the sample, affecting the extracted features and resulting in a misclassification, both targeted and non-targeted.

Generating AEs with GEA. The process of generating adversarial examples is done by merging the code of the original sample with the code of a targeted sample. The targeted sample belongs to the class which the adversary desires to misclassify to. Fig. 1 shows the generated CFGs of each sample. Fig. 1(a) is the generated CFG of the original sample code, whereas, Fig. 1(b) represents the CFG of the embedded code. Combining the original and external code results in Fig. 1(c). The combination is done by creating a shared entry and exit blocks, where only one branch is executed. In this case, the left branch that belongs to the original sample will be executed. Note that changing in the structure of the code will change the extracted features, resulting in misclassification.

D. Threat Model

The adversary's goal is to fool the classifier by misclassifying malicious samples as benign and vice versa, while preserving the functionality and practicality of the original sample. Our threat model focuses on AEs generated based on code-level manipulation using GEA (in section II-C).

We assume that the adversary has full access to the source code of benign and malicious samples. Moreover, the adversary can edit, compile, and merge samples, and knows the model's design and its internal architecture. The adversary's

goal is to conduct targeted and non-targeted misclassification. The objective of Soteria is to provide a robust and accurate classification in the presence of this model.

III. System Design

A. High-Level Architecture

To address the impracticability of modification-based adversarial examples, we propose Soteria, a malware classification framework that incorporates two modules: adversarial sample detection and malware classification. Soteria manifests the following advantage. It eliminates the cost of extracting new features, meaning that it can re-use the features generated during the detection of the AEs to classify a sample as benign or malicious. Alternatively, the user has the flexibility over the choice of classifier, meaning that the user can make use of different set of features, classifier parameters, or another classifier altogether. Figure 2 represents the high-level architecture of Soteria, comprising of three major components, feature extractor, AEs detector, and malware classifier.

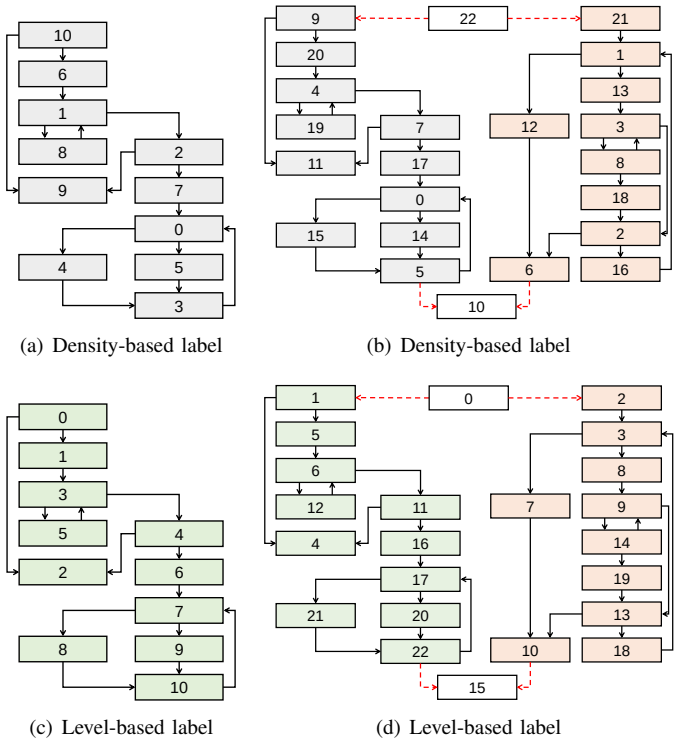
Feature Extractor. Soteria utilizes the features from the graphical representation of a program's flow execution, i.e., CFG. For a graph G, such that, $G = (V, E)$, and nodes (V) and edges (E) represent the basic blocks and the traversed paths, respectively. A critical advantage of the CFG is that it summarizes the control flow by connecting the entry block with reachable blocks directly or indirectly. Particularly, if a block of code is appended to an existing program, with an intention to fool the classifiers, knowing that the appended blocks are unreachable, our feature extraction methodology ignores such blocks, in contrast to binary- and image-base classifiers. The features driven from the CFG ignore the non-executable part of samples, eliminating the effect from noise injection and unused functions in the sample.

AEs Detector. The detector is a standalone component used prior to the classification process to filter out practical AEs. In this way, we eliminate the model's vulnerability to AEs by forwarding only legitimate samples (*i.e.,* benign or malicious) to the classifier that was trained using a non-adversarial dataset. Unlike the commonly used approaches in the literature, the detector is trained using only non-adversarial dataset, while maintaining a distinguishable feature representation that enables detecting potential AEs.

Classifier. Soteria requires a classifier that can accurately classify the samples into malicious and benign with the resistance towards the impractical AEs. For evaluation, we make use of an ensemble of CNN classifiers, however, it can be replaced with another desired method.

B. Adversarial Examples Detector

The purpose of the detector is to distinguish normal samples from adversarial ones, regardless of whether the sample is malicious or not. Fig. 3 shows the flow of the feature extraction, including sample pre-processing with CFG extraction and labeling, followed by feature extraction using n-gram of the obtained random walks on the labeled CFG.



(a) Density-based label (b) Density-based label

(c) Level-based label (d) Level-based label

Fig. 4: Graph labeling using two approaches, density- and level-based. Each node has a label in $[0, |V|-1]$, where $|V|$ is the number of nodes in G. Fig. 4(b) and Fig. 4(d) show the labeling of the GEA generated CFG over the original graphs in Fig. 4(a) and Fig. 4(c), respectively.

1) Sample Pre-processing: The pre-processing phase is concerned with nodes labeling. For a graph $G = (V, E)$, we use two labeling approaches: density-based and level-based.

❶ Density-based Labeling (DBL). The density of a node is defined as the summation of in- and out-edges over the total number of edges in the graph. DBL sorts all nodes according to their density, where the most dense node is labeled as 0 and the least dense node is labeled as $|V|-1$, and the centrality factor of a node is used to rank nodes with tied density CF_{v_i}[1]. If two or more nodes still have the same centrality factor, we assign labels based on their levels, considering the main or entry block function as the entry node. We notice some cases where two nodes with equal values are at the same level (symmetric nodes), and label them in ascending order since switching their labels will not affect the consistency of labeling. Fig. 4(a) shows the result of the density-based labeling. As shown, node 0 and 1 are the most dense nodes because they are connected to four blocks, and node 0 has a higher centrality factor value. The labeling ends by assigning label 10 to the entry block as

[1]Centrality factor of a node is the sum of node's betweenness and closeness centrality values, $CF_{v_i} = B_{v_i} + C_{v_i}$. The betweenness centrality (B_{v_i}) of a node v_i is defined as $\Delta(v_i)/\Delta(m)$, where $\Delta(v_i)$ is the count of shortest paths travel through v_i and connecting nodes v_j and v_t, for all j and t where $i \neq j \neq t$, and $\Delta(m)$ is the total number of shortest paths between such nodes. The closeness centrality (C_{v_i}) of a node is defined as the average shortest path between node v_i and all other nodes in the graph G.

the least dense node with the lowest centrality factor.

Level-based Labeling (LBL). The level of a node v_i is defined by the smallest number of steps S_{v_i} from the entry node to reach v_i, where the level of a node is equal to $1 + S_{v_i}$. In LBL, we consider the main or entry block function in the CFG as the first level layer, and follow (in breadth-first search manner) other levels for labeling them. For nodes at the same level, we follow the same labeling mechanism in DBL. Fig. 4(c) shows an example for the result of LBL, where the entry block is assigned with label 0. In the second level, there are two nodes with the same density values, and the centrality factor values are used. The process ends by labeling the last level nodes. Note that the entry block will always have the label 0 when using the LBL method.

Both density- and level-based labeling follow the strict predefined rules to guarantee consistency of representation and ensures that any modification applied to the graph will be reflected in the labels' assignment. Fig. 4(b) and Fig. 4(d) show the labeling of the generated graphs using GEA. It is worth noting that the labels' assignment varies for each graph, even when they share a sub-graph. Labels' assignment over GEA results in changing the labels, and the feature extraction process, hence affecting the detector's behavior.

2) Feature Representation: For feature generation and representation, we apply a random walk and use a method based on the n-gram model to approximate the graph.

Random Walk: A random walk describes random steps in the graph space, and is used to estimate the graph state space. Let G be an undirected graph with a marker placed at v_i, initially the entry block. At each step, the marker moves to an adjacent vertex v_j with probability $\frac{1}{deg(v_i)}$, where $deg(v_i)$ is the degree of v_i. The marker keeps track of the visited vertices' labels as it moves, e.g., random walk over the original sample graph in Fig. 4 may generate $W =$ "10 9 2 1 2 ..." when using DBL and $W' =$ "0 2 4 3 4 ..." when using LBL. We define the length of the random walk as $|W|$ (the number of labeled nodes collected by a random walk of length $|W|$ is ($|W|$+1). In Soteria, $W = 5 \times |V|$, and repeat the walk ten times over DBL and ten times over LBL, resulting in 20 vectors. The use of random walk helps to randomize the feature extraction process, making it difficult to generate practical AEs. We observed that the repetition of the process improves the quality of the random walks' feature representation, corresponding to the underlying graph.

n-grams: The n-gram technique can be used in different models for feature representation of text, documents, graphs, etc. Unique terms or n-gram are extracted from the entire corpus before counting the frequencies in individual samples. Inspired by node2vec [27], we use n-gram representation of the graphs from the sequences of nodes obtained by the random walk. From the derived random walks with the lengths specified above, we extract n-grams of lengths 2, 3, and 4 as a feature representation of the CFG. Given that, the number of n-grams is large even for small graphs. We select and use the top 500 discriminative features for each LBL and DBL (thus, 1,000 features in total). The selection of the top discriminative feature is based on the frequency of W.

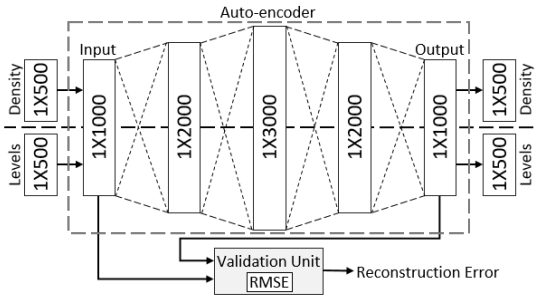


Fig. 5: The proposed AEs detector: The detector consists of five fully connected layers auto-encoder. The input to the auto-encoder is density- and level-based feature vectors, where the output is the reconstructed feature vectors. A validation unit is used to calculate the reconstruction error. A sample is considered as AE if reconstruction error exceeds a threshold.

3) Building Detection Model: The core of the detection model is an auto-encoder that consists of five fully connected dense layers (as shown in Fig. 5). The auto-encoder reconstructs the given input at the output layer, and consists of four main blocks; an input layer, an output layer, hidden layers, and a validation unit, which are described in the following.

- **Input Layer.** This layer is a one-dimensional vector of size 1×1000 fed by the density- and level-based features vectors.
- **Hidden Layers.** These layers consist of three fully connected dense layers, and extract a deep representation of the features. The design is based on decoding the features from 1×1000 to 1×2000 and 1×3000. Afterward, a third layer encodes the features presentation to 1×2000. This structure eliminates the features dependencies in the reconstruction process, as the extracted features are mutually independent.
- **Output Layer.** This layer is fully connected to the third hidden layer. With a shape of 1×1000, the output layer reconstructs the features seen at the input as its output, which is then returned as a density- and level-based vector.
- **Validation Unit.** The validation unit computes the reconstruction error (RE) by calculating the Root Mean Square Error (RMSE) between the original input x and the reconstructed output $\hat{x}$. If the RMSE exceeds the threshold, set to be 50%, the sample x is labeled as AE.

C. Classifier

As the detector distinguishes between adversarial and clean samples, the classifier distinguishes clean samples into benign or one of three malicious families: Gafgyt, Mirai, and Tsunami. For this purpose, two CNN classifiers are utilized to incorporates separately the density- and level-based features.

1) CNN Classifiers: The input to the classifier in Soteria is a one dimensional (1D) vector of size 1×500 representing the density- or level-based extracted features. Fig. 7 shows the structure of the classifier, which consists of three blocks: convolutional blocks (ConvB) 1 and 2 and a classification block (CB). All layers use the Rectified Linear Units (ReLU)

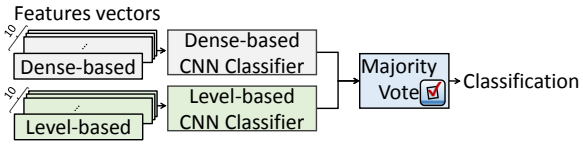


Fig. 6: Soteria classification process. The CNN-based models' inputs are the dense- and level-based feature vectors. The classification decision is the majority vote of the CNN classifiers output probabilities over the feature vectors.

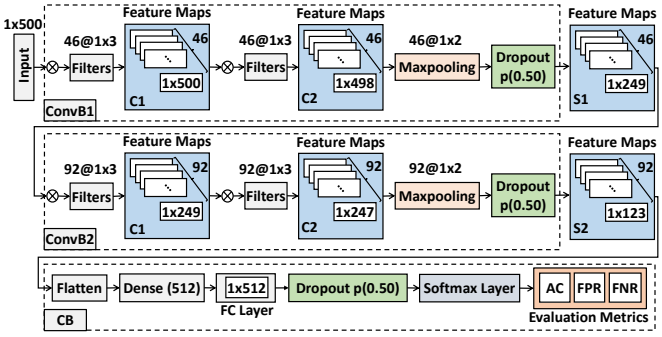


Fig. 7: The structure of Soteria classifiers. The classifiers consist of four convolutional layers with max-pooling and dropout functions. The output of the classifier is the softmax probability of each class.

activation function, and dropout regularization to prevent model over-fitting. We describe the CNN structure in the following using the notation p as the dropout probability, s as the stride, m as the max-pooling size.

- **ConvB1.** ConvB1's input is the extracted features, and consists of two consecutive convolutional layers with 46 filters of size 1×3, that operate convolutions with $s = 1$ with no padding to generate feature maps of size 46×498. Each convolutional layer is followed by a max-pooling with $s = m = 2$ and a dropout with $p = 0.25$.
- **ConvB2.** Similar to ConvB1, except for the number of filters. ConvB2 consists of two convolutional layers with 92 filters of size 1×3, followed by max-pooling and dropout.
- **CB.** CB's input is the flattened feature maps of ConvB2, fed to a fully connected layer of size 512 with a dropout $p = 0.5$. The output of the fully connected layer is fed to a softmax layer for the classification.

2) Majority Voting: For each sample, we perform ten random walks and generate 20 feature vectors (from both DBL and LBL). These feature vectors are forwarded to their corresponding CNN classifiers. The final output is based on the majority voting unit, where the class with the highest vote is used as the sample's label (see Fig. 6).

IV. Dataset and Evaluation

A. Dataset

To evaluate Soteria, we assembled a dataset of IoT benign samples and IoT malware. We collected 13,798 malware samples, randomly selected from CyberIOCs [28] during the period of January 2018 to late February of 2019. For the benign samples, we manually assembled a dataset of 3,016 samples from source-code projects available on GitHub [29]. Next, we used radare2 [30] to obtain the CFGs of the samples. Throughout the study, wherever required, we use 80% our dataset for training and validation, and 20% for evaluation.

TABLE II: Distribution of IoT samples across the benign and malicious families. Gafgyt is the most popular IoT family with 66.18% of the dataset samples, while Tsunami is the least popular with only 262 samples (1.55% of the samples). The dataset is split into the train (80%) and test (20%) subsets.

Class	# of Samples			% of Samples
	# Train	# Test	# Total	
Benign	2,416	600	3,016	17.94%
Gafgyt	8,911	2,217	11,128	66.18%
Mirai	1,935	473	2,408	14.33%
Tsunami	210	52	262	1.55%
Overall	13,472	3,342	16,814	100%

Malware Family (Class). To determine the family label of the malware, we inspect the malware samples through *VirusTotal* [31]. The scan results from the *VirusTotal* are then passed through *AVClass* [32] to label them with their family class. *VirusTotal* scans include scan results from multiple anti-virus software, each of which assign a family name to the malware. *AVClass* further uses majority vote to determine the family label. Soteria classifies the samples into different classes, i.e., family labels and benign. Table II shows the IoT samples' distribution across classes.

Adversarial Dataset. Recall that we utilize the GEA to the generate the AEs to evaluate Soteria's robustness. These AEs are generated from the test dataset (20% of the samples per class). Towards this, we start by selecting three samples from each class, i.e., one from each sizes, small, medium, and large. We define small, medium, and large by minimum, median, and maximum number of nodes in the dataset. Taking a sample from a class for each size, as the targeted sample, we generated adversarial examples by applying GEA over every sample in the test dataset of all the classes except for the targeted sample class. For example, if we select a sample of size *Small* from the benign dataset, we then apply GEA over this sample and each of the samples in the test dataset of Gafgyt, Mirai, and Tsunami, giving us a total of 2,742 AEs (it can be seen in Table II that Gafgyt, Mirai, and Tsunami have 2,217, 473, and 52 samples, respectively, aggregating to 2,742 AEs). The number of generated AEs of each class is in Table III.

B. Feature Analysis

We extract features from 200 random samples from each class. Recall that we extract density- and level-based features from each sample. We use both of these feature vectors together to create a combined feature vector of size 1×1000. We used Principal Component Analysis (PCA) [33] with a dimension of two. PCA converts a set of observations of possibly correlated variables into a set of values of linearly

TABLE III: GEA selected targeted samples. These samples are used to generate the AEs to evaluate Soteria. Three samples from each class are selected of different sizes (number of nodes), *i.e.,* small, medium, and large.

Class	Size	# Nodes	# AEs
Benign	Small	10	2742
	Medium	50	2742
	Large	443	2742
Gafgyt	Small	13	1125
	Medium	64	1125
	Large	133	1125
Mirai	Small	12	2869
	Medium	48	2869
	Large	235	2869
Tsunami	Small	15	3290
	Medium	46	3290
	Large	79	3290

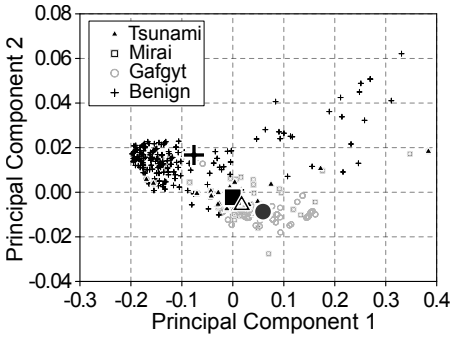
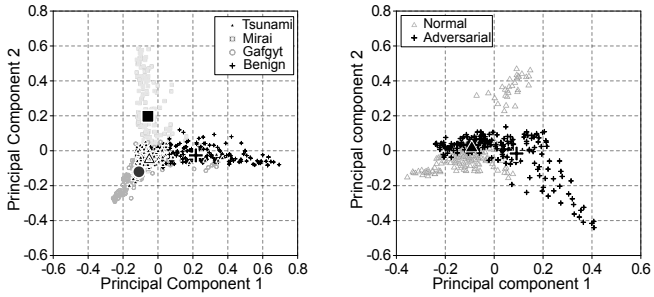
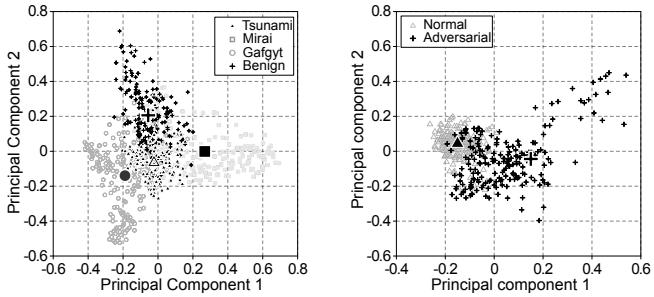


Fig. 8: The PCA comparison between the benign and malware families using features used in Alasmary *et al.* [3].



(a) Benign and malware (b) Normal and adversarial

Fig. 9: Soteria: Dense-based labeling feature vector comparison. Fig. 9(a) shows the PCA distribution of benign and malware samples. Fig. 9(b) shows the PCA distribution comparison between the normal and GEA generated AEs.



(a) Benign and malware (b) Normal and adversarial

Fig. 10: Soteria: Level-base labeling feature vector comparison. Fig. 10(a) shows the PCA distribution of benign and malware samples. Fig. 10(b) shows the PCA distribution comparison between the normal and GEA generated AEs.

uncorrelated variables called principal components.

Baseline Comparison. Prior works, like, Alasmary *et al.* [3] and Abusnaina *et al.* [9] use graph theoretic features extracted from the general structure of the CFG. With the comparative analysis of such features with our feature considerations, we exhibit our feature sets to be more discriminative. Fig. 8, and Figures 9(a), 10(a), and 11(a) show the PCA visualization of the feature vectors between the classes of features considered in the prior works and our features design, respectively. Notice that our feature representation is more discriminative of the classes. Additionally, we notice that the malicious classes in the figures are indistinguishable using the graph theoretic features. Table IV shows the distribution of the discriminative features over the four classes with 51 and 129 density-based and level-based features, respectively, shared between classes.

AE vs. Clean Features. To detect AEs, i.e., distinguish the AEs from the clean samples, understanding the differences in feature representation between clean and AEs is important. To examine this, we applied PCA on the clean and adversarial feature vectors, the results of which are shown in Fig. 9(b), 10(b), and 11(b). Notice that the clean and AEs are distinguishable, particularly when using the combined feature vectors.

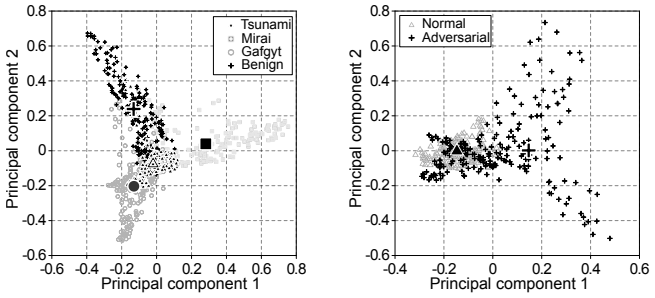
C. Evaluation and Analysis

Recall that Soteria has two major functionality, AE detection and classification. Below, we present the evaluation of Soteria's performance and also compare it with the baseline.

1) Adversarial Example Detector: We evaluated AE detector of Soteria by its ability to detect adversarial examples and distinguish them from the clean samples, regardless of their class. Fig. 9(b), 10(b), and 11(b) show the spatial differences between clean and adversarial samples.

Training Parameters. We trained Soteria on reconstructing the training data in Table II. The reconstruction error (RE) is the RMSE between the original and reconstructed samples, we set the number of epochs to 100 with a batch size of 128.

Testing. Given the trade-off between adversarial detection sensitivity (false negatives) and the clean samples misdetection (false positive), setting a proper RE threshold is essential. We calculate the RE and set the threshold (T_h) as $T_h = \mu(\vec{R_E}) + \sigma(\vec{R_E})$, where $\vec{R_E}$ is a vector of all RE values of the training samples, and μ and σ are the mean and standard



(a) Benign and malware (b) Normal and adversarial

Fig. 11: Soteria: Combined labeling feature vector comparison. Fig. 11(a) shows the PCA distribution of benign and malware samples. Fig. 11(b) shows the PCA distribution comparison between the normal and GEA generated AEs.

TABLE IV: Distribution of dense- and level-based feature vectors extracted by $n-grams$ technique from the random walk traces among the IoT benign and malware classes.

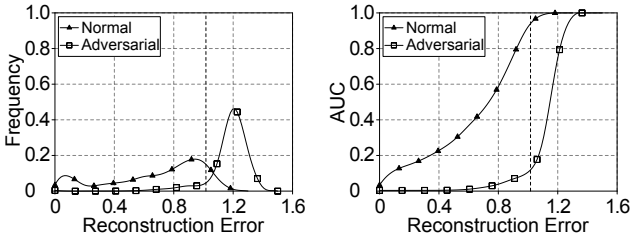
Class	# Features			% Features		
	Dense	Levels	Total	Dense	Levels	Total
Benign	153	290	443	30.6%	58.0%	44.3%
Gafgyt	445	450	895	89.0%	90.0%	89.5%
Mirai	162	251	413	32.4%	50.2%	41.3%
Tsunami	114	240	354	22.8%	48.0%	35.4%
Shared	51	129	180	10.2%	25.8%	18.0%

deviation of the training samples RE, respectively. Fig. 12 shows the RE distribution over the clean and adversarial features vectors. To consider a sample as adversarial, half of its feature vectors should have a RE higher than the threshold.

Performance. Table V shows Soteria's performance against AEs. Overall, the detector detects 97.79% of the AEs. In most cases (9 out of 12), the detector was able to detect AEs with an accuracy greater than 99%. Furthermore, Table VI shows the detection performance against clean samples. Notice that only samples from Gafgyt family were misdetected as AEs, mainly because of the high number of discriminative features associated with this family, as shown in Table IV. In conclusion, we detected AEs and distinguish them from the clean samples with high accuracy. Detected samples are labeled as adversarial and will not be forwarded to the classifier.

Analysis. To show the importance of setting the right threshold, we re-implement the threshold as $T_h = Mean(\vec{R_E}) + \alpha \times SDV(\vec{R_E})$, where α is an arbitrary value. We test the detector performance against the clean and adversarial samples by varying α from 0 to 2.0. Fig. 13 shows the effect of α on the detection error. With $\alpha = 0$, all AEs were detected, although more than 60% of the clean samples were classified as AEs. With $\alpha = 2.0$, all clean samples were correctly detected and no AEs were detected by Soteria. Note that our selected threshold was chosen without access to the test dataset.

2) Classifier: The classifier aims to correctly distinguish a sample into the aforementioned classes (Benign, Mirai, Gafgyt, or Tsunami). We evaluate the performance of Soteria alongside the existing approaches.



(a) RE: Distribution (b) RE: AUC

Fig. 12: Reconstruction Error (RE) comparison between normal and the generated AEs. Fig. 12(a) shows the distribution frequency of the RE among the normal and adversarial samples. Fig. 12(b) represents the accumulated frequencies of samples and their corresponding RE. The vertical dashed line is the chosen threshold for Soteria AEs detector.

TABLE V: GEA: Detector Performance over adversarial samples. The detector was able to detect an overall percentage of 97.79% of the AEs. DE refers to the detected samples.

Class	Size	# AE	# DE	% DE
Benign	Small	2,742	2,741	99.96%
	Medium	2,742	2,739	99.89%
	Large	2,742	2,340	85.34%
Gafgyt	Small	1,125	1,115	99.11%
	Medium	1,125	1,125	100%
	Large	1,125	1,120	99.55%
Mirai	Small	2,869	2,865	99.86%
	Medium	2,869	2,864	99.82%
	Large	2,869	2,680	93.67%
Tsunami	Small	3,290	3,289	99.97%
	Medium	3,290	3,287	99.91%
	Large	3,290	3,248	98.72%
Overall		30,078	29,413	97.79%

Training Parameters. We set the number of epochs to 100 with a batch size of 128 and evaluated the performance of each model individually and against the majority voting.

Performance. We evaluated Soteria's classifier's performance against two existing models: 1) Graph-based: Alasmary *et al.* [3] propose a malware detector based on features extracted from the general structure of the CFG. , and 2) Image-based: Cui *et al.* [5] use image-based design where each sample is represented as an image of a fixed size to detect malware. We implemented the above two systems. Table VII shows the performance of the models for the different classes. The model accuracy over each class is measured by the number of samples correctly classified over the total number of samples that belong to that class. For image-based model, we implemented the four models mentioned in their design. The evaluation of 96×96 and 192×192 based models shows poor performance, with an overall accuracy rate of 66.37%. Therefore, we did not

TABLE VI: GEA: Detector Performance over clean samples. Only 6.16% of the clean samples were misclassified as AEs. All Benign clean samples passed the detector. DE refers to the detected samples (lower is better).

Class	# Samples	# DE	% DE
Benign	600	0	0%
Gafgyt	2,217	206	9.29%
Mirai	473	0	0%
Tsunami	52	0	0%
Overall	3,342	206	6.16%

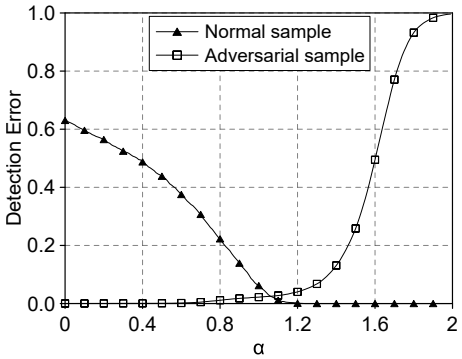


Fig. 13: Effect of varying the detector threshold (α) on the detection error. The selected α in Soteria is the intersection between the error rates of normal and adversarial samples.

include it in the comparison. Our evaluation shows that Soteria outperforms the existing systems, as particularly shown in the Tsunami classification and overall accuracy rate.

Analysis. Recall that the accuracy of our AE detector was 97.79%, meaning that 2.21% of the AEs were not detected by Soteria, and were forwarded to the classifier. Given it's application, it is important to understand the classifier's behavior against those samples. Table VIII shows the classifier's behavior over these samples. The classifier detects them as benign or Gafgyt, with a large percentage (76.1%) of the samples classified as benign. It is worth noting that the targeted classification is not valid in this design, as Fig. 9(b), Fig. 10(b), and Fig. 11(b) show a clear difference in the feature representation between clean and AEs. However, due to the variety in the benign samples' features distribution, the adversarial examples that pass the detector are likely to be classified as benign. This can be critical, even with a detection rate of as high as 97.79%, given the application domain.

V. Discussion

System Robustness. Our evaluation shows that Soteria is robust, with the ability to detect AEs with an accuracy of 97.79%, and a trade-off of detecting 206 Gafgyt samples as adversarial. Moreover, Soteria outperforms other systems over the same training and testing datasets. The compared systems had an overall low Tsunami classification accuracy, due to the small dataset. Soteria, on the other hand, and using a majority voting system, achieved an accuracy of 100% in classifying

TABLE VII: Classification performance of Soteria dense-, level-, and voting-based classification systems in classifying normal (non-adversarial) samples.

Class	Model Accuracy					
	Soteria			[9]	[5]	
	DBL	LBL	Voting		24×24	48×48
Benign	99.45	99.70	**100**	99.00	99.00	99.50
Gafgyt	99.70	97.00	**100**	98.55	98.87	99.14
Mirai	**99.49**	98.73	99.36	97.67	92.81	92.81
Tsunami	**100**	**100**	**100**	84.61	32.69	59.61
Overall	99.63	97.77	**99.91**	98.29	97.01	97.70

TABLE VIII: Soteria's classifier predictions over AEs misdetected by the detector. Most of the misdetected samples are generated using GEA with large size selected samples.

Class	Size	# AE	Classification			
			Benign	Gafgyt	Mirai	Tsunami
Benign	Small	1	1	0	0	0
	Medium	3	1	2	0	0
	Large	402	287	115	0	0
Gafgyt	Small	10	10	0	0	0
	Medium	0	0	0	0	0
	Large	5	4	1	0	0
Mirai	Small	4	4	0	0	0
	Medium	5	5	0	0	0
	Large	181	145	36	0	0
Tsunami	Small	1	1	0	0	0
	Medium	3	3	0	0	0
	Large	42	39	3	0	0

Tsunami sample. In fact, the majority voting classifier only failed in classifying three Mirai samples in the evaluation, classifying them as benign samples.

Operation Mode for Detector. Soteria is used to distinguish AEs and detect them. To enable Soteria's operation, the extracted features distribution of normal and AEs should be different. Moreover, we argue that the detector should not be aware of the AEs and their patterns in the training process, as this will bias the detector's performance towards specific attacks, decreasing the robustness against other attacks.

Adversarial Capabilities. In section II-D, we discuss the threat model and adversarial capabilities. We assumed that the adversary can access the source code of the samples, and can modify and merge them. Moreover, he has prior knowledge of the design and its internal architecture. Soteria's success implies that the adversary cannot generate practical AEs. What the adversary does not have in Soteria is the ability to know in advance what features are being used for the classifier, since those features are randomized for every run of the system. For instance, inserting a single block with a low density near the exit block will not highly affect the labeling of the sample, and will not be detected as an AE by Soteria. However, Soteria can classify the sample to its original class, since the labels are intact. Moreover, the adversary needs to ensure that the labels change in such a way the classification decision will be toggled, without being detected by the AE detector, which

happened in our evaluation in 2.21% of the generated AEs. Finally, and due to the change in the labeling, the adversary cannot force the classifier into a targeted misclassification.

Alternative Features for Classifier. In Soteria, we built a classifier that is based on the utilized features from the detector design process. However, the classifier can be replaced, with some caveats. The detector decision is based on the extracted CFG. Appended binaries at the end of the file will not affect the detector decision. Clean samples with adversarial binaries appended to them will not be detected as AEs by Soteria. While this is an advantage of Soteria classifier, it is equally a serious shortcoming with other approaches, such as image-based malware classifiers [34]. Ideally, the classifier should be at least as good as the classifier proposed in this paper, meaning that it should only consider the executable binaries in the classification process. Moreover, the discriminative features are highly distinguishable among classes, and the feature extraction process is immune to feature space manipulation.

Limitations. Our work has two major limitations. ❶ **CFG-based Features:** CFG-based features are effective compared to other feature designs. However, CFG does not necessarily reflect the actual code. Editing the code without even changing the functionality (by creating an equivalence) would affect the structure of the CFG, which might be exploited by the adversary to evade detection in the first place. For example, an adversary may inject a sample of code that would not result in a new branching, but would still affect the structure of the CFG. While such an event is well within the scope of our adversary model, and would not affect the classification results, it would only affect the feature space, requiring us to retrain Soteria to capture the new feature space. ❷ **Binary Obfuscation:** Obtaining a representative CFG would not be possible under obfuscation, typically done using string obfuscation, resulting in hiding parts of the code, or function obfuscation, resulting in an incomplete CFG. An incomplete CFG may result in an incomplete feature representation of the sample, and thus a misclassification. Obfuscation is a shortcoming of our work, and deobfuscation is an active research area in its own right, where developed tools can be used as the basis for our work to obtain representative CFGs.

VI. Related Work

Machine and deep learning algorithms are widely leveraged towards securing software against adversaries in general and detecting malware in particular. For instance, Alasmary *et al.* [3] analyzed two prominent malware, IoT and Android, based on the CFG-graph representation of the malicious software. Moreover, Alam *et al.* [35] analyzed the malware and proposed a malware detection system to detect malware with even small CFGs and then addressing the changes occurred in the frequencies of opcodes. Bruschi *et al.* [36] proposed a malware detection method that uses two CFG techniques to compare and detect malware based on two CFGs of malware code and other known malware.

Several research works have been proposed to defend against adversarial machine learning. Most of these approaches are image-based methods. For example, Goodfellow *et al.* [10] proposed to train the model with a set of AEs to minimize the test error between the real and AEs of the model's result. Papernot *et al.* [37] designed a network distillation model to defend against adversarial attacks such as fast gradient sign method [10] and L-BFGS attack [38]. Cui *et al.* [39] introduced a malware detection method for malicious codes using deep learning by transferring the malicious code into grayscale images. Ni *et al.* [40] proposed a malware family classification system that converts malicious codes of nine different malware families into grayscale images. Metzen *et al.* [13] proposed a detection method for adversarial perturbation over trained AEs. Moreover, Rozsa *et al.* [41] proposed a machine learning model that tested the adversarial examples. They correlate their robustness of the three adversarial attacks to the accuracy of eight deep network classifiers. In addition, Miyato *et al.* [26] proposed a detection method on the text domain. They trained the model over adversarial examples that apply small perturbation to the word that is embedded in RNN.

Several methods have been proposed to generate adversarial examples that can manipulate the desired output to fool the classifiers [9], [10], [11], [12]. Adversaries can make small modifications to the malware to misclassify them as benign, yet they remain malware files [42], [43]. Other methods apply and add small noise or perturbation to optimize the images to generate the adversarial examples [44], [10], [45]. For example, Carlini and Wagner [45] proposed three adversarial attacks against distilled neural networks that break many defenses models. Moreover, Moosavi-Dezfooli *et al.* [12] proposed a DeepFool method that generates minimal perturbation to change the classification labels based on iterative linearization of the classifiers. Recently, Abusnaina *et al.* [9] proposed adversarial attacks over the CFGs of malware binaries through designing two adversarial attacks to craft the IoT detector.

VII. Conclusion

In this paper, with Soteria, we address the need to detect adversarial machine learning attacks by proposing an adversarial machine learning detector for IoT malware. Particularly, Soteria defends the CFG-based classifiers for malware detection against the AEs. The first component, the AE detector, is a Control Flow Graph (CFG)-based model that can detect adversarial samples without training the model over adversarial samples (as shown by prior works). The model computes the reconstruction error between the input data and the reconstructed output of the auto-encoder, and uses a threshold to detect the adversarial samples with an overall accuracy of 97.79%. Additionally, the second component of Soteria performs a family-based classification with an accuracy of 99.91% on the clean samples. These two models operate independently, increasing the robustness of Soteria.

Acknowledgement. This work was supported by Cyber-Florida Collaborative Seed Award and NRF under grant 2016K1A1A2912757.

REFERENCES

[1] 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 *USENIX Security*, 2012, pp. 491–506.

[2] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *USENIX Security*, 2003.

[3] 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*, 2019.

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

[5] Z. Cui, F. Xue, X. Cai, Y. Cao, G.-g. Wang, and J. Chen, "Detection of malicious code variants based on deep learning," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3187–3196, 2018.

[6] 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.

[7] D. Kong and G. Yan, "Discriminant malware distance learning on structural information for automated malware classification," in *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD*, 2013, pp. 1357–1365.

[8] J. Yan, G. Yan, and D. Jin, "Classifying malware represented as control flow graphs using deep graph convolutional neural network," in *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, 2019, pp. 52–63.

[9] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, and A. Mohaisen, "Adversarial learning attacks on graph-based IoT malware detection systems," in *the 39th IEEE International Conference on Distributed Computing Systems, ICDCS*, 2019.

[10] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *the 3rd International Conference on Learning Representations, ICLR*, 2015.

[11] W. Hu and Y. Tan, "Generating adversarial malware examples for black-box attacks based on GAN," *arXiv preprint arXiv:1702.05983*, vol. abs/1702.05983, 2017.

[12] 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.

[13] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff, "On detecting adversarial perturbations," in *the 5th International Conference on Learning Representations, ICLR*, 2017.

[14] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial malware binaries: Evading deep learning for malware detection in executables," in *The 26th European Signal Processing Conference, EUSIPCO*, 2018, pp. 533–537.

[15] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet, "Deceiving end-to-end deep learning malware detectors using adversarial examples," in *Workshop on Security in Machine Learning (NIPS)*, 2018.

[16] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth, "Evading machine learning malware detection," *Black Hat*, 2017.

[17] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers," in *the 23rd Network and Distributed System Security Symposium, NDSS*, 2016, pp. 21–24.

[18] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial malware binaries: Evading deep learning for malware detection in executables," in *the 26th European Signal Processing Conference, EUSIPCO*, 2018, pp. 533–537.

[19] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems, NIPS*, 2014, pp. 2672–2680.

[20] W. Xu, D. Evans, and Y. Qi, "Feature squeezing: Detecting adversarial examples in deep neural networks," in *25th Annual Network and Distributed System Security Symposium, NDSS*, 2018.

[21] D. Meng and H. Chen, "Magnet: A two-pronged defense against adversarial examples," in *ACM Computer and Communications Security, CCS*, 2017, pp. 135–147.

[22] F. Liao, M. Liang, Y. Dong, T. Pang, X. Hu, and J. Zhu, "Defense against adversarial attacks using high-level representation guided denoiser," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2018, pp. 1778–1787.

[23] G. S. Dhillon, K. Azizzadenesheli, Z. C. Lipton, J. Bernstein, J. Kossaifi, A. Khanna, and A. Anandkumar, "Stochastic activation pruning for robust adversarial defense," in *the 6th International Conference on Learning Representations, ICLR*, 2018.

[24] N. Papernot, P. D. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *IEEE Security and Privacy, SP*, 2016, pp. 582–597.

[25] P. Samangouei, M. Kabkab, and R. Chellappa, "Defense-gan: Protecting classifiers against adversarial attacks using generative models," in *the 6th International Conference on Learning Representations, ICLR*, 2018.

[26] T. Miyato, A. M. Dai, and I. J. Goodfellow, "Adversarial training methods for semi-supervised text classification," in *the 5th International Conference on Learning Representations, ICLR*, 2017.

[27] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *the 22nd ACM International Conference on Knowledge Discovery and Data Mining, KDD*, 2016, pp. 855–864.

[28] Developers. (2019) Cyberiocs. Available at [Online]: https://freeiocs.cyberiocs.pro/.

[29] Developers. (2019) Github. Available at [Online]: https://github.com/.

[30] Developers. (2019) Radare2. Available at [Online]: https://https://rada.re/r/.

[31] Developers. (2019) VirusTotal. Available at [Online]: https://www.virustotal.com.

[32] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVclass: A tool for massive malware labeling," in *Processing of the International Symposium on Research in Attacks, Intrusions, and Defenses, RAID*, 2016, pp. 230–253.

[33] T. P. Minka, "Automatic choice of dimensionality for PCA," in *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS)*, 2000, pp. 598–604.

[34] X. Liu, J. Zhang, Y. Lin, and H. Li, "ATMPA: attacking machine learning-based malware visualization detection methods via adversarial examples," in *International Symposium on Quality of Service, IWQoS*, 2019, pp. 38:1–38:10.

[35] 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.

[36] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *Detection of Intrusions and Malware, and Vulnerability Assessment Conference, DIMVA*, 2006, pp. 129–143.

[37] N. Papernot, P. D. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against deep learning systems using adversarial examples," vol. abs/1602.02697, 2016. [Online]. Available: http://arxiv.org/abs/1602.02697

[38] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *International Conference on Learning Representations, ICLR*, 2014.

[39] Z. Cui, F. Xue, X. Cai, Y. Cao, G. Wang, and J. Chen, "Detection of malicious code variants based on deep learning," *Trans. Industrial Informatics*, vol. 14, no. 7, pp. 3187–3196, 2018.

[40] S. Ni, Q. Qian, and R. Zhang, "Malware identification using visualization images and deep learning," *Computers & Security*, vol. 77, pp. 871–885, 2018.

[41] A. Rozsa, M. Günther, and T. E. Boult, "Are accuracy and robustness correlated," in *the 15th IEEE International Conference on Machine Learning and Applications, ICMLA*, 2016, pp. 227–232.

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

[43] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," vol. abs/1606.04435, 2016.

[44] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *the 5th International Conference on Learning Representations, ICLR*, 2017.

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