Analyzing and Detecting Emerging Internet of Things Malware: A Graph-based Approach

Hisham Alasmary[‡†], Aminollah Khormali[†], Afsah Anwar[†], Jeman Park[†],
Jinchun Choi[¶†], Ahmed Abusnaina[†], Amro Awad[†], DaeHun Nyang[¶], and Aziz Mohaisen[†]
[†]University of Central Florida [‡]King Khalid University [¶]Inha University

***Abstract*—The steady growth in the number of deployed Internet of Things (IoT) devices has been paralleled with an equal growth in the number of malicious software (malware) targeting those devices. In this work, we build a detection mechanism of IoT malware utilizing Control Flow Graphs (CFGs). To motivate for our detection mechanism, we contrast the underlying characteristics of IoT malware to other types of malware—Android malware, which are also Linux-based—across multiple features. The preliminary analyses reveal that the Android malware have high density, strong closeness and betweenness, and a larger number of nodes. We show that IoT malware samples have a large number of edges despite a smaller number of nodes, which demonstrate a richer flow structure and higher complexity. We utilize those various characterizing features as a modality to build a highly effective deep learning-based detection model to detect IoT malware. To test our model, we use CFGs of about 6,000 malware and benign IoT disassembled samples, and show a detection accuracy of ≈99.66%.**

***Index Terms*—Malware; Android; IoT; Graph Analysis; IoT Detection.**

I. Introduction

THE Internet of Things (IoT) is a new networking paradigm interconnecting a large number of devices, such as voice assistants, sensors, and automation tools, with many promising applications [2]. The persistent interconnection of IoT devices is destined to augur into a wide spectrum of implementations, varying from everyday requirements of the general population to very sophisticated industrial usages. Moreover, the anticipated rise in the number of IoT devices over the years and in every industry reflects the prevalence of interconnected devices and their implications.

This work was supported in part by AFRL (Air Force Research Lab) summer program, in part by NSF under Grant CNS-1809000 and Grant CNS-1814417, in part by NRF (National Research Foundation of Korea) under Grant 2016K1A1A2912757, and in part by Cyber Florida Seed Grant. An earlier version of this work has appeared in CSoNet 2018 [1]. (*corresponding authors: DaeHun Nyang; Aziz Mohaisen.*)

H. Alasmary is with the Department of Computer Science at King Khalid University, Abha 61421, Saudi Arabia, and also with the Department of Computer Science at University of Central Florida, Orlando, FL 32816, USA.
A. Khormali, A. Anwar, J. Park, A. Abusnaina, and A. Mohaisen are with the Department of Computer Science at University of Central Florida, Orlando, FL 32816, USA (mohaisen@ucf.edu).
A. Awad is with the Department of Electrical & Computer Engineering at University of Central Florida, Orlando, FL 32816, USA.
J. Choi is with the Department of Computer Engineering at Inha University, Incheon 22212, South Korea, and also with the Department of Computer Science at University of Central Florida, Orlando, FL 32816, USA.
D. Nyang is with Department of Computer Engineering at Inha University, Incheon 22212, South Korea (nyang@inha.ac.kr.).

The increasingly persistent connection of IoT devices makes their role lie somewhere on the continuum between advantage and susceptibility. Each of those devices runs multiple pieces of software, or applications, which are increasing in complexity, and could have vulnerabilities that could be exploited, resulting in various security threats and consequences, i.e DDoS attacks [3]–[6].

One of the prominent threats to these embedded devices, from the perspective of the software, is malware. IoT software is different from well-understood ones on the other platforms, such as Android applications, Windows binaries, and their corresponding malware. Most of IoT embedded devices run on a Linux base that uses small versions of libraries to achieve Linux-like capabilities. In particular, *Busybox* is widely used to achieve the desired functionality; based on a light-weighted structure, it supports utilities needed for IoT devices.

On the other hand, and due to common structures, the Linux capabilities of the IoT systems inherit and extend the potential threats to the Linux system. Executable and Linkable Format (ELF), a standard format for executable and object code, is sometimes exploited as an object of malware. The ELF executable built by the adversary propagates malware over networks and carries out attacks following the command preassigned in the binary or from Command & Control server.

Considering their uniqueness of limited computation and limited power capability compared to their contemporary systems [7], [8], studying the characteristics of malicious software is important. More specifically, understanding IoT software through analysis, abstraction, comparison (with other types of malware; e.g., Android) and classification (from benign IoT software; i.e., IoT malware detection) is an essential problem to mitigate those security threats [2], [9]. In this regard, we deeply look into the IoT malware samples to understand their constructs and unique features.

Admittedly, there has been a large body of work on software analysis in general, and a few attempts on analyzing IoT software in particular. However, the efforts on IoT software analysis have been very limited with respect to the samples analyzed and the approaches attempted. The limited existing literature on IoT malware, despite malware analysis, classification, and detection being a focal point of analysts and researchers [10]–[13], points at the difficulty, compared to other malware types. Understanding the similarity and differences of IoT malware compared to other prominent malware types will help analysts understand the differences and use them to build detection systems upon those differences.

Starting with a new dataset of IoT malware samples, we pursue a graph-theoretic approach to malware analysis. In this approach, each malware sample is abstracted into a Control Flow Graph (CFG), which could be used to extract representative static features of the application. As such, graph-related features from the CFG can be used as a representation of the software, and classification techniques can be built to tell whether the software is malicious or benign, or even what kind of malicious purposes the malware serves (e.g., malware family-level classification and label extrapolation). Given that malware analysis is quite a constant topic in the security research community, it would be intellectually and technically beneficial to explore how existing and new approaches would be a useful tool in understanding their differences with newer types of malware. To figure out how different the IoT malware is from other types of emerging malware, such as Android mobile applications, we perform a comparative study of the graph-theoretic features in both types of software, which highlights the difference of CFG between IoT malware and Android malware.

Contributions. We make the following contributions:

1) Using CFGs as our analysis vector, we compare and contrast the IoT malware with the Android malware by augmenting various graph-theoretic features, such as nodes count, edges count, degree centrality, betweenness centrality, diameter, radius, distribution of shortest path, etc. Although, both the platforms use Linux as base operating system, our results surprisingly reveal compelling differences between the two malware categories.
2) Towards analyzing the CFGs, we disassemble a large number of samples. Namely, we use close to 9,000 samples in total for our analysis. We use a dataset of 2,962 IoT malware samples and a dataset of 2,891 Android malware samples collected from different sources. Additionally, we assemble a dataset of benign files capable of running on IoT devices towards effective malware detection. The datasets, for Android malware, IoT malware, and IoT benign samples, and their associated CFGs, will be made public to the community for reproducibility.
3) Using the different features as described above, grouped under seven different groups as a modality for detecting IoT malware, we design a deep learning-based detection system that can detect malware with an accuracy of $\approx 99.66\%$. Additionally, the system has the ability to classify malware into their respective families with an accuracy of $\approx 99.32\%$.

Organization. The rest of this paper is organized as follows. In section II we review the related work. In section III, we introduce the dataset, data representation and augmentation. The methodology and approach of this paper, including control flow graph definitions, and graph-theoretic metrics are outlined in section IV. In section V, we present the malware contrast results for IoT and Android samples, followed by IoT detection in section VI, including detection algorithms, evaluation metrics, the flow of detection system, comparison, discussion, and evaluation. The concluding remarks are in section VII.

II. Related Work

Graph-based Approach. The limited number of works have been done on analyzing the differences between Android (or mobile) and IoT malware, particularly using abstract graph structures. Hu *et al.* [14] designed a system, called SMIT, which searches for the nearest neighbor in malware graphs to compute the similarity across function using their call graphs. They focused on finding the graph similarity through an approximate graph-edit distance rather than approximating the graph isomorphism since few malware families have the same subgraphs with others. Shang *et al.* [12] analyzed code obfuscation of the malware by computing the similarity of the function call graph between two malware binaries – used as a signature – to identify the malware. Christodorescu and Jha [15] analyzed obfuscation in malware code and proposed a detection system, called SAFE, that utilizes the control flow graph through extracting malicious patterns in the executables. Bruschi *et al.* [20] detected the self-mutated malware by comparing the control flow graph of the malware code to the control flow graphs for other known malware. Moreover, Tamersoy *et al.* [18] proposed an algorithm to detect malware executables by computing the similarity between malware files and other files appearing with them on the same machine, by building a graph that captures the relationship between all files. Yamaguchi *et al.* [21] introduced the code property graph which merges and combines different analysis of the code, such as abstract syntax trees, control flow graphs and program dependence graphs in the form of joint data structure to efficiently identify common vulnerabilities. In addition, Caselden *et al.* [16] generated a new attack polymorphism using hybrid information and CFG, called HI-CFG, which is built from the program binaries, such as a PDF viewer. The attack collects and combines such information based on graphs; code and data, as long as the relationships among them. Moreover, Wüchner *et al.* [17] 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. Jang *et al.* [19] built a tool to classify malware by families based on the features generated from graphs.

Android Malware Detection. Gascon *et al.* [22] detected Android malware by classifying their function call graphs. They found reuse of malicious codes across multiple malware samples showing that malware authors reuse existing codes to infect the Android applications. Zhang *et al.* [23] proposed a detection system for Android malware by constructing signatures through classifying the API dependency graphs and used that signature to uncover the similarities of Android applications behavior. Ham *et al.* [24] detected Android malware using the Support Vector Machine (SVM). Milosevic *et al.* [25] proposed a dynamic detection system for Android malware and low-end IoT devices by analyzing a few features extracted from the memory and CPU usage, and achieved a classification accuracy of 84% with high precision and recall.

IoT Malware Detection. Pa *et al.* [26] proposed IoTPOT, an IoT honeypot and sandbox to analyze and capture IoT telnet-

TABLE I: Summary of the related works. Abbreviations: SVM (Support Vector Machine), CNN (Convolutional Neural Network), NB (Naive Bayes), LR (Logistic Regression), DT (Decision Tree-based J48), and RF (Random Forest).

Author	Platform	Dataset	Sample size	Task	Approach
Hu *et al.* [14]	x86	malware	102,391	Analysis	Function Call Graph
Shang *et al.* [12]	x86	malware, benign	51	Analysis	Function Call Graph
Christodorescu and Jha [15]	x86	malware, benign	14	Detection	Control Flow Graph
Caselden *et al.* [16]	x86	benign programs	2	Analysis	Information Flow Graph, Control Flow Graph
Wüchner *et al.* [17]	x86	malware, benign	7,501	Detection	Quantitative data Flow Graphs
Tamersoy *et al.* [18]	x86	malware, benign	43,353,581	Detection	File-Relation Graph
Jang *et al.* [19]	x86	malware, benign	3,768	Classification	System Call Graph
Bruschi *et al.* [20]	Linux	malware, benign	572	Analysis	Control Flow Graph
Yamaguchi *et al.* [21]	Linux	vulnerabilities	88	Analysis	Code Property Graph
Gascon *et al.* [22]	Android	malware, benign	147,950	Detection	Function Call Graph/Machine Learning (SVM)
Zhang *et al.* [23]	Android	malware, benign	15,700	Detection	Weighted Contextual API Dependency Graphs
Ham *et al.* [24]	Android	malware, benign	28	Detection	Machine Learning (SVM)
Milosevic *et al.* [25]	Android	malware, benign	2,199	Detection	Classifier (NB, LR, DT)
Pa *et al.* [26]	IoT	malware	106	Collection	IoT Honeypot
Su *et al.* [27]	IoT	malware, benign	865	Detection	Deep Learning (CNN)
Wei and Qiu [28]	IoT	malware, benign	554	Detection	Algorithm
Hossain *et al.* [29]	IoT	N/A	N/A	Forensic	Digital ledger (Blockchain)
Shen *et al.* [30]	IoT	malware	N/A	Detection	Theoretical analysis
Antonakakis *et al.* [31]	IoT	malware	1,028	Analysis	Static analysis
Kolias *et al.* [32]	IoT	malware	N/A	Analysis	Analyze Mirai source code
Donno *et al.* [33]	IoT	malware	N/A	Analysis	Analyze Mirai source code
THIS WORK	IoT, Android	malware	5,853	Analysis	Control Flow Graph
THIS WORK	IoT	malware, benign	5,961	Detection	Control Flow Graph/Deep Learning (CNN)

based attacks targeting IoT environment that run on multiple CPU architectures. Su *et al.* [27] proposed an IoT detection system capable of capturing DDoS attacks on IoT devices by generating gray-scale images from malware binaries as feature vectors. Their system achieved an accuracy of 94% using deep learning. Wei and Qiu [28] analyzed IoT malicious codes and built a detection system by monitoring the code run-time on the background of the IoT devices. Moreover, Hossain *et al.* [29] proposed an IoT forensic system, named Probe-IoT, that investigates IoT malicious behaviors using distributed digital ledger. Shen *et al.* [30] proposed an intrusion detection system for the low-end IoT networks that run on the cloud and fog computing to overcome malware propagation and to preserve multistage signaling privacy on IoT networks.

Other research works have been done for detecting and analyzing IoT botnets. For examples, Antonakakis *et al.* [31] analyzed Mirai botnets which launch DDoS attacks using IoT devices. Kolias *et al.* [32] examined the operation and communication life-cycle of Mirai botnets used for launching and observed traffic signatures that can be used for their detection. Donno *et al.* [33] analyzed a taxonomy of DDoS attacks, more specifically for a Mirai botnet, and classified these attacks into malware families and found out the relationship between them. [Table I highlights literature works on analyzing and detecting different malware types on different operating systems platforms using various approaches.]

III. Dataset

The goal of this study is to understand the underlying differences between modern Android and emerging IoT malware through the lenses of graph analysis. The abstract graph structure through which we analyze malware is the control flow graph (CFG), previously used in analyzing malware as shown above. Unique to this study, however, we look into the various algorithmic and structural properties of those graphs to understand code complexity, analysis evasion techniques (e.g., decoy functions, obfuscation, etc.). Finally, we use the aforementioned characteristics, the algorithmic and structural properties of the graphs, to build a system to distinguish malware from the benign binaries.

Towards this goal, we start by gathering datasets required to accomplish the end goal of this study. As such, we create a dataset of binaries and cluster them under three different categories: Android malware samples, IoT malware samples, and benign IoT samples. For our IoT malware dataset, we collected a new and recent IoT malware, up to late January of 2019, using CyberIOCs [34]. For our Android dataset, various recent Android malware samples were obtained from a security analysis vendor [35].

Finally, to test our proposed IoT malware detector, we manually assembled a dataset of benign samples from source files on GitHub [36]. For our analysis and detection, we augment the datasets by reversing the samples to address various analysis issues. Using an off-the-shelf tool, we then disassemble the malware samples to obtain the CFG corresponding to each of them. We use the CFG of each sample as an abstract representation and explore various graph analysis measures and properties. The rest of this section highlights the details of the dataset creation and associated analysis.

A. Dataset Creation

Our IoT malware dataset is a set of 2,962 malware samples, randomly selected from CyberIOCs [34]. Additionally, we also obtained a dataset of 2,891 Android malware samples from [35] for contrast. These datasets represent each malware type. We reverse-engineered the malware datasets using *Radare2* [37], a reverse engineering framework that provides various analysis capabilities including disassembly. To this end, we disassemble the IoT binaries, which in the form of

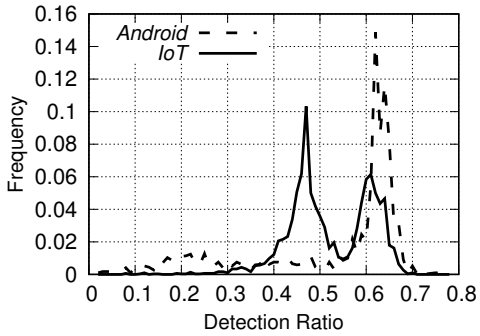


Fig. 1: Android and IoT malware detection rate on VirusTotal.

Executable and Linkable Format (ELF) binaries, as well as the Android Application Packages (APKs) using the same tool, *Radare2*. Which is an open source command line framework that supports a wide variety of malware architecture and has a Python API, which facilitated the automation of our analysis.

Labeling. To determine if a file is malicious, we uploaded the samples on *VirusTotal* [38] and gathered the scan results corresponding to each of the malware. We observe that each of the IoT and Android malware is detected by at least one of the antivirus software scanners listed in VirusTotal, whereas the Android dataset has a higher rate.

Differences. We notice that the IoT malware have a lower detection rate compared to the Android malware, which is perhaps anticipated given the fact that the IoT malware samples are recent and emerging threats, with fewer signatures populated in antivirus scanners compared to the well-understood Android malware, as shown in figure 1. In particular, we plot the detection ratio (across multiple scanners, where 1 means that the sample is detected by all scanners) against the frequency of samples with the given detection ratio. We notice that the Android samples a distribution focused around 0.6–0.7 detection ratio, were the larger number of IoT samples have detection concentration around the ratio of 0.4–0.5.

To examine the diversity and representation of malware in our dataset, we label them by their family (class) using *AVClass* [39], a tool that ingests the *VirusTotal* results and provides a family name for each sample through various heuristics of label consolidation. We gather a new IoT malware dataset and a larger Android malware dataset compared to the ones used in our prior work [1]. Moreover, we ignore IoT malware families with less than 10 samples. Table II shows the IoT malware families and top three Android families, with their share in their corresponding datasets. Overall, we notice the IoT malware belong only to three families, while the Android malware belong to 180 unique families.

Processing. In a preprocessing phase, we first manually analyzed the samples to understand their architectures and whether they are obfuscated or not, then used *Radare2*'s Python API, *r2pipe*, to automatically extract the CFGs for all malware samples not obfuscated—in this work we assume it is possible to obtain the CFG, and addressing obfuscation is an orthogonal contribution that we defer for future work. Then, we used an off-the-shelf graph analysis tool, *NetworkX*, to compute various graph properties. Using those calculated properties, we then analyze and compare IoT and Android malware. Figure 2 shows the analysis workflow we follow to perform our analysis, as well as the IoT detection flow system. After disassembling the binaries, we look into the main function in the assembly instruction and extract the CFG from that point. Otherwise, we extract the CFG for those without main from the entry point. Then, we use the extracted CFGs for further analysis.

TABLE II: Dataset: Top 3 Android and IoT families.

Android Family	# of samples	IoT Family	# of samples
Smsreg	1061	Gafgyt	1,351
Smspay	381	Mirai	1,349
Zdtad	139	Tsunami	262

IV. Methodology

We use the CFGs of the different malware samples as abstract characteristics of programs for their analysis.

Program Formulation. For a program P, we use $G = (V, E)$ capturing the control flow structure of that program as its representation. In the graph G, V is the set of nodes, which correspond to the functions in P, whereas E is the set of edges which correspond to the call relationship between those functions in P. More specifically, we define $V = \{v_1, v_2 \ldots, v_n\}$ and $E = \{e_{ij}\}$ for all i, j such that $e_{ij} \in E$ if there is a flow from v_i to v_j. We use $|V| = n$ to denote the size of G, and $|E| = m$ to denote the number of primitive flows in G (i.e., flows of length 1). Based on our definition of the CFG, we note that G is a directed graph. As such, we define the following centralities in G. We define $A = [a_{ij}]^{n \times n}$ as the adjacency matrix of the graph G such that an entry $a_{ij} = 1$ if $v_i \to v_j$ and 0 otherwise.

A. Graph Algorithmic Properties

Using this abstract structure of the programs, the CFG, we proceed to perform various analyses of those programs to understand their differences and similarities. We divide our analysis into two broader aspects: general characteristics and graph algorithmic constructs. To evaluate the general characteristics, we analyze the basic characteristics of the graphs. In particular, we analyze the number of nodes and the number of edges, which highlight the structural size of the program. Additionally, we evaluate the graph components to analyze patterns between the two malware types. Components in graphs highlight unreachable codes, which are the result of decoys and obfuscation techniques, as can be observed in the example of Android malware sample in figure 3. This can be a result of obfuscating the parent node of the branching component. Moreover, we assess the graph algorithmic constructs; in particular, we calculate the theoretic metrics of the graphs, such as the diameter, radius, average closeness centrality, etc. We now define the various measures used for our analysis.

Definition 1 (Degree Centrality): For a graph $G = (V, E)$ as above, the degree centrality is defined as the number of relations or number of edges of a node. Mathematically, it is defined as, $D^+ = [d_i^+ / \sum_{j=1}^{n} d_j^+]^{1 \times n}$ and $D^- = [d_i^- / \sum_{j=1}^{n} d_j^-]^{1 \times n}$ for the in- and out-degrees of the graph.

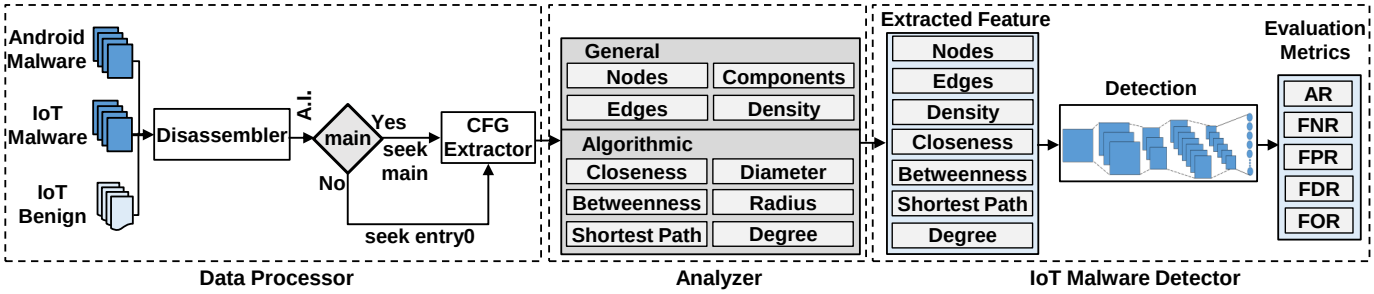


Fig. 2: Pipeline of analysis and detection using CFGs. Abbreviations: A.I. (Assembly Instructions), AR (Accuracy Rate), FNR (False Negative Rate), FPR (False Positive Rate), FDR (False Discovery Rate), and FOR (False Omission Rate).

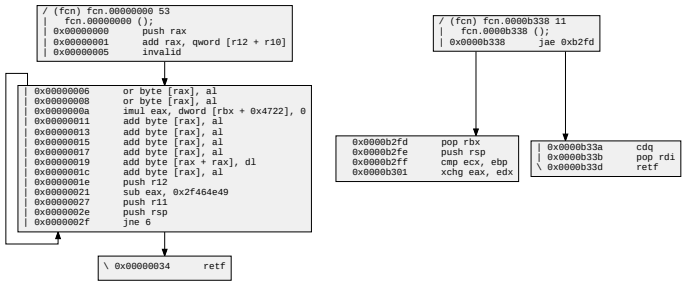
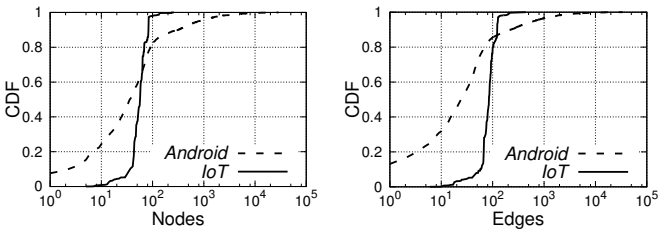


Fig. 3: CFG of a malware highlighting unreachable codes, depicting use of decoy or obfuscation techniques in malware.



(a) Number of nodes' distribution. (b) Number of edges distribution.

Fig. 4: A characterization of the number of nodes and edges comparing Android to IoT malware samples. Notice that the x-axis is logarithmic scale.

Definition 2 (Density): The density of a graph is defined as the closeness of an edge to the maximum number of edges. For a graph $G = (V, E)$, the graph density can be represented as the average normalized degree; that is, $Density = 1/n \sum_{i=1}^{n} \deg(v_i)/n - 1$, where $V = \{v_1, v_2, \ldots, v_n\}$.

Definition 3 (Shortest Path): For a graph $G = (V_i, E_i)$, the shortest path is defined as: $v_i^x, v_i^{x_1}, v_i^{x_2}, v_i^{x_3}, \ldots v_i^y$ such that $length(v_i^x \rightarrow v_i^y)$ is the shortest path. It finds all shortest paths from $v_i^x \rightarrow v_i^y$, for all $v_i^{x_j}$, which is arbitrary, except for the starting node v_i. The shortest path is then denoted as: $S_{v_i^x}$.

Definition 4 (Closeness centrality): For a node v_i, the closeness is calculated as the average shortest path between that node and all other nodes in the graph G. This is, let $d(v_i, v_j)$ be the shortest path between v_i and v_j, the closeness is calculated as $c_c = \sum_{\forall v_j \in V \nearrow v_i} d(v_i, v_j)/n - 1$.

Definition 5 (Betweenness centrality): For a node $v_i \in V$, let $\Delta(v_i)$ be the count of shortest paths via v_i and connecting nodes v_j and v_r, for all j and r where $i \neq j \neq r$. Furthermore, let $\Delta(.)$ be the total number of shortest paths between such nodes. The betweenness centrality is defined as $\Delta(v_i)/\Delta(.)$.

Definition 6 (Connected components): In graph G, a connected component is a subgraph in which two vertices are connected to each other, and which is connected to no additional vertices in the subgraph. The number of components of G is the cardinality of a set that contains such components.

Definition 7 (Diameter and Radius): The diameter of a graph $G = (V, E)$ is defined as the maximum length of shortest path between any two pairs of nodes in G, while the radius is the minimum shortest path between any two nodes in G. This is, let $d(v_i, v_j)$ be the shortest path length between two nodes in G, then the diameter is $\max_{\forall i \neq j} d(v_i, v_j)$ while the radius is $\min_{\forall i \neq j} d(v_i, v_j)$.

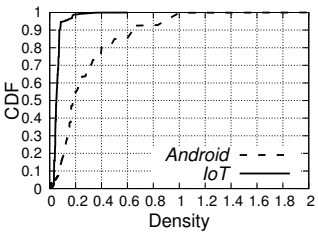
In this work, we use a normalized version of the centrality, for both the closeness and betweenness, where the value of each centrality ranges from 0 to 1.

V. Results

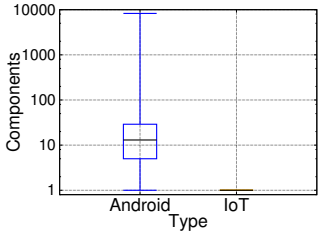
A. General Analysis

Figures 4a and 4b represent the logarithmic scale to show the skewness to the large values and to show the difference of percent change between the Android and IoT malware in terms of two major metrics of evaluation of graphs, namely the nodes and edges.

Size Analysis: Nodes. The Android and IoT malware samples have at least 28,691 and 260 nodes, respectively. Figure 4a represents the CDF logarithmic scale for the number of nodes in both malware datasets to highlight the percent change towards large node values of the Android samples. We note that those numbers are not close to one another, highlighting a different level of complexity and the flow-level. In addition, we notice a significant difference in the topological properties in the two different types of malware at the node count level. This is, while the Android malware samples seem to have a variation in the number of nodes per sample, characterized by the slow growth of the y-axis (CDF) as the x-axis (the number of nodes) increases. On the other hand, the IoT malware have less variety in the number of nodes: we also notice that the dynamic region of the CDF is between around 1 and 60 nodes (slow curve), corresponding to around [0–0.15] of the CDF (this is, 60% of the samples have 1 to 60 nodes, which is a relatively small number). Furthermore, with the Android

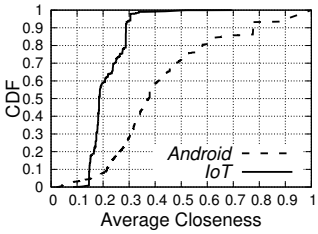


(a) The distribution of density.

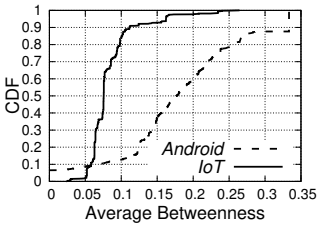


(b) The distribution of components.

Fig. 5: A characterization of the density and the number of components comparing Android to IoT malware samples.



(a) Average of closeness centrality



(b) Average of betweenness centrality

Fig. 6: A characterization of the closeness and betweenness centralities comparing Android to IoT malware samples.

malware, we notice that a large majority of the samples (almost 80%) have around 100 nodes in their graph. This characteristic seems to be unique and distinguishing, as shown in figure 4a.

Size Analysis: Edges. The top 1% of the Android and IoT malware samples have 33,887 and 439 edges, respectively, which shows a great difference between them. In particular, Figure 4b represents the CDF logarithmic scale of the edges count for both malware datasets. The Android samples have a large number of edges in every sample that can be shown from the slow growth on the y-axis. Similar to the node dynamic region for the IoT, the IoT samples seem to have a smaller number of edges; the active region of the CDF between around 1 to 90 edges correspond to around [0–0.15] (about 15% of the samples). Additionally, we notice that the smallest 60% of the Android samples (with respect to their graph size) have around 40 edges whereas the percentage of the IoT samples have around 90 edges.

This combined finding of the number of edges and nodes in itself is very intriguing: while the number of nodes in the IoT malware samples is relatively smaller than that in the Android malware, the number of edges is higher. This is striking, as it highlights a simplicity at the code base (smaller number of nodes) yet a higher complexity at the flow-level (more edges), adding a unique analysis angle to the malware that is only visible through the CFG structure.

Graph Density Analysis. Figure 5a shows the density of the datasets, where we notice almost 90% of the IoT samples have a density around 0.07 whereas the Android samples have a diverse range of density over around 0.65. By examining the CDF further, we notice that the density alone is a very discriminative feature of the two different types of malware: if we are to use a cut-off value of around 0.08 – 0.09, for example, we can successfully tell the different types of malware apart with an accuracy exceeding 90%.

Graph Components Analysis. Figure 5b shows a boxplot illustration of the number of components in both the IoT and Android malware's CFGs. We notice that all IoT samples (100%) have only one component that represents the whole control graph for each sample. These samples have a range of file sizes from 1,100 – 2,300,000 bytes. The Android malware have a large number of components. We find that 13.83%, or 400 Android samples, have only one component, where their size ranges from around 4,200 – 9,400,000 bytes. On the other hand, 2,491 samples (around 86.17%) have more than one component. We note that the existence of multiple components in the CFG is indicative of the unreachable code in the corresponding program (possible a decoy function to fool static analysis tools). As such, we consider the largest component of these samples for further CFG-based analysis. However, we notice that 298 Android samples have the same node counts in the first and second largest components. Furthermore, we find 197 samples that have the same number of node and edge counts in the first and second largest components. The number of nodes and edges in these samples ranges from 0 – 18, but the file sizes range from around 12,000 – 25,700,000 bytes.

Root Causes of Unreachable Code / Components. Figure 5b shows the boxplot of the number of components for both the Android and IoT malware. The boxplot captures the median and 1st and 3rd quartile, as well as the outliers. We notice that the median of the number of components in IoT samples is 1, whereas the majority of Android malware lies between 5 and 29, with a median of 13 components. We notice this issue of unreachable code to be more prevalent in the Android malware but not in the IoT malware, possibly for one of the following reasons. 1) The Android platforms are more powerful, allowing for complex software constructs that may lead to unreachable codes, whereas [majority of] the IoT platforms are constrained, limiting the number of functions (software-based). 2) The Android Operating System (OS) is advanced and can handle large code bases without optimization, whereas the IoT OS is a simple environment that is often time optimized through tools that would discard unreachable codes before deployment.

B. General Algorithmic Properties and Constructs

The aforementioned analysis represents the general trend for the graphs, while there are different graph algorithmic properties towards further analysis for the resulting graph to uncover deeper characteristics. These algorithmic features provide more information about the graph constructs, We elaborate on further analysis using those features in the following.

Graph Closeness Centrality Analysis. Figure 6a depicts the CDF for the average closeness centrality for both datasets. To reach to this plot, we generalize the definition in 4 by aggregating the average closeness for each malware sample and obtaining the average. As such, we notice that around 5% of the IoT and Android have around 0.14 average closeness centrality. This steady growth in the value continues for the

Android samples as shown in the graph; 80% of the nodes have a closeness of less than 0.6. On the other hand, the IoT samples closeness pattern tend to be within the small range: the same 80% of IoT samples have a closeness of less than 0.29, highlighting that the closeness of 0.3 can also be used as a distinguishing feature of the two different types of the malware, but with low detection rate of around 65%.

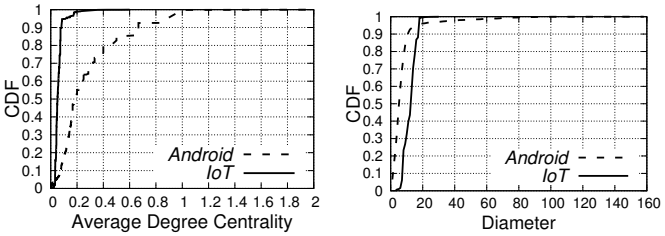
Graph Betweenness Centrality Analysis. Figure 6b shows the average betweenness centrality for both datasets. The average betweenness is defined by extending definition 5 in a similar way to extending the closeness definition. Similar to the closeness centrality, 10% of the IoT and Android samples have almost 0.06 average betweenness centrality, which continues with a small growth for the Android malware to reach around 0.26 average betweenness after covering 80% of the samples. However, we notice a significant increase in the IoT curve where 80% of the samples have around 0.09 average betweenness that shows a slight increase when covering a large portion of the IoT samples. This huge gap noticed in figures 6a and 6b is quite surprising although explained by correlating the density of the graph to both the betweenness and the closeness: Android samples tend to have a higher density, thus an improved betweenness, which is not the case of IoT.

Graph Degree Centrality Analysis. Figure 7a shows the average of degree centrality in the largest components. We notice that 10% of the IoT and Android malware have an average degree centrality of around 0.03 and 0.09, respectively. The slow growth continues with Android malware to reach around 0.42 after covering 80% of the samples. However, there is a significant increase in the IoT samples; around 0.08 after covering the same 80% of the samples. This huge gap can also be used as a feature to detect IoT malware.

Diameter, Radius, and Shortest Paths Analysis. Figure 7b shows the diameter of the graphs. Almost 10% of the IoT samples have a diameter of around 8 that can be noticed from the slow growth in the CDF, whereas the Android malware have around 1. After that, there is a rapid increase in the CDF curve for the diameter in the 80% of both samples, reaching around 10 and 18 for the Android and IoT, respectively. Similarly, figure 8a shows the CDF of the radius of the graphs. We notice that 15% of the Android samples have a radius of around 1, while the IoT samples have around 4. In addition, 80% of the Android samples have around 4 while the IoT have around 7. This shows the significant increase for both datasets. As a result, from these two figures, we can define a feature vector to detect the Android and IoT samples.

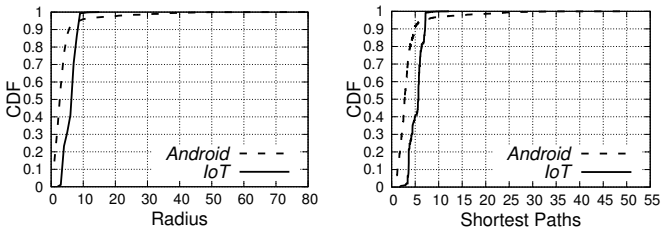
Figure 8b represents the average shortest path for the graphs. Similar to the other feature vectors, we notice almost 80% of the IoT malware have an average shortest path greater than 5, whereas the Android malware have an average of less than 5.

Upon increasing the number of Android malware samples to be similar to the IoT samples, we notice that the gap between both datasets can still be noticed, showing the new trend shift of the IoT malware to accommodate the low-end IoT devices with less computational resources. This, in turn, can lead to differentiating between the IoT malware and Android, and possibly to other malware, such as Windows malware.



(a) Average of degree centrality. (b) The distribution of diameter.

Fig. 7: A characterization of the degree centrality and diameter comparing Android to IoT malware samples.



(a) The distribution of radius. (b) Average shortest paths.

Fig. 8: A characterization of the radius and average shortest path comparing Android to IoT malware samples.

VI. IoT Detection

This section is devoted to the detection of the IoT malware based on the aforementioned CFGs features. In order to investigate the robustness of the classifier, we conducted two experiments that detect the IoT malware samples from the benign ones; and classify the IoT samples to the corresponding family. In addition, we utilized both traditional Machine Learning algorithms, such as Linear Regression (LR) classifier, Support Vector Machine (SVM), and Random Forest (RF) as well as more advanced deep learning methods, such as Convolutional Neural Network (CNN) in our experiments. A brief description of these algorithms is in the following.

A. Detection Algorithms

Logistic Regression (LR). LR is a method borrowed from the field of statistics for linear classification of data into discrete outcomes. LR is a popular statistical modeling method where the probability of dichotomous outcome event is transformed into a set of explanatory variables as followed:

$$\text{logit}(P_1) = \ln\left(\frac{P_1}{1-P_1}\right) = \beta_0 + \beta_1 x_1 + ... + \beta_n x_n = \beta_0 + \sum_{i=1}^{n} \beta_i x_i,$$

where, $x_1, x_2, \ldots, x_n$ are the variables and β_1, β_2, ..., β_n are corresponding coefficients and β_0 is the intercept. Maximum Likelihood Estimation (MLE) method is used to estimate the value of these coefficients. MLE aims to maximize the log likelihood in an iterative process. Interested readers are referred to [40] for more information about logistic regression.

Support Vector Machine (SVM). SVM classifies the data by finding the best hyper-plane that separates the data from the two classes. SVM selects a class t by applying:

$$f(x_t) = \underset{n}{argmax}[(w_n \times x_t) + b_n], n = 1, ..., N,$$

where, $f(x_t)$ is the feature vector of sample, n is a binary classifier, w_n is the weight vector and b_n is the cut-off of the classifier. Both w_n and b_n master and learn from training. For training a new classifier to achieve a preferable class, the training analyses are considered as positive examples, which are included in the class, while the remaining attempts are negative examples. To classify a new analysis, the classifier computes the margin and selects the hyper-plane with the largest margin between the two classes [41].

Random Forest (RF). RF classifier is a powerful classification algorithm specifically for nonlinear classification tasks as they offer good accuracy, low over-fitting, and controlled output variance [42]. Incorporation of random feature selection with bagging is used to train T decision trees (weaker learners), which allows a variance reduction in the output of individual trees [43]. In this study, we set the number of weak learners to $T = 60$ as it offers the best performance in our case. Generally, a T-sized random forest model is grown as followed:

- A bootstrap sample is chosen from the training set to grow each tree. Usually, two-thirds of samples are used to grow each tree and the remaining samples are used to calculate the out-of-bag error.
- n variables out of N variables are randomly selected in the training process. Generally, n less than $\sqrt{N}$ is considered as the starting point.
- One variable, out of n selected variables, is used at each node to conduct the best split.

Convolutional Neural Network (CNN). The general design of the CNN consists of several layers, including convolution, activation, pooling, and a dropout followed by a classification layer. The convolution layer extracts a feature map by applying a convolutional filter to the input data. The pooling layer makes features more distinct and reduces the amount of data. Final discrimination of the input data is conducted in the classification layer. In this study, the input X of the CNN model is a one-dimensional (1D) vector containing extracted features formatted as 1×23. The CNN design consists of three blocks, namely convolutional block 1 (CB1), convolutional block 2 (CB2), and classification block (CL). The detailed description of these blocks are as follows:

- **CB1.** This block is made up of 1D convolutional layer with padding and 46 filters ${F_{b1}}'$ of size 1×3. The filters convolve over the input data X with a stride of 1. The output of this layer ${C_{b1}}'$ is a 2D tensor of size 23×46. The output of the first convolutional layer is then fed into a similar 1D convolutional layer without padding, resulting in a 2D tensor ${C_{b1}}''$ of size 21×46. Afterward, a max pooling with size and stride of 2 and dropout with probability of 0.25 are applied, which results into a 2D tensor S_{b1} of size 10×46.

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

- **CB2.** Fed by the output of CB1 S_{b1}, this block is similar to CB1 except for the number of filters ${F_{b2}}'$ in the convolutional layers. This block consists of a 1D convolutional layer with padding and 92 filters of size 1×3, convolving over the data with a stride of 1. The output of this layer is forwarded to a similar 1D convolutional layer without padding, resulting into a 2D tensor ${C_{b2}}''$ of size 8×92. Then, we perform max pooling with a size and stride of 2, followed by dropout with probability of 0.25. The output of this block is a tensor S_{b2} of size 4×92.

$$\begin{aligned}
{C_{b2}}'_i &= S_{b1} \otimes {F_{b2}}'_i, && i = 1:92\\
{C_{b2}}''_i &= {C_b1}'_i \otimes {F_{b2}}''_i, && i = 1:92\\
M_{b2\,i} &= \mathsf{maxpool}({C_{b2}}''_i, 2, 2), && i = 1:92\\
S_{b2\,i} &= \mathsf{dropout}(M_{b2\,i}, 0.25), && i = 1:92
\end{aligned}$$

- **CL.** The generated tensor in CB2 S_{b2} is then fed into this block, forwarding the tensor to flatten layer, converting it into 1D tensor of size 368, followed by a dense layer of size 512 resulting into a fully connected layer feature map FCL and a dropout with probability of 0.5 resulting into S_{FC}. Finally, S_{FC} is fed to the softmax layer as classification layer. The outputs of the softmax layer will be evaluated based on various metrics, such as accuracy rate (AR), false negative rate (FNR), etc. to measure the performance of the model.

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

We trained our model using 200 epochs with a batch size of 100. Each epoch took an average time of 0.7 seconds on a system comprised of an i5-8500 CPU, 32GB DDR4 RAM, and NVIDIA GTX980 Ti Graphics Processing Unit (GPU). 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. The architecture of the CNN design is shown in Figure 9. We refer the interested reader to [44] for more details on CNN internals.

B. Evaluation Metrics

In order to investigate the generalization of the classifier, the K-fold cross validation method [45] is used. Although K is an unfixed parameter, K=10 is commonly used in the literature [46]–[48]. For a 10-fold cross-validation, the dataset is partitioned into ten different partitions. Then, the model is trained over nine partitions and tested on the remaining partition. This process is repeated ten times until all portions are evaluated as test data, and the average result is reported. The confusion matrix is used to evaluate the performance of

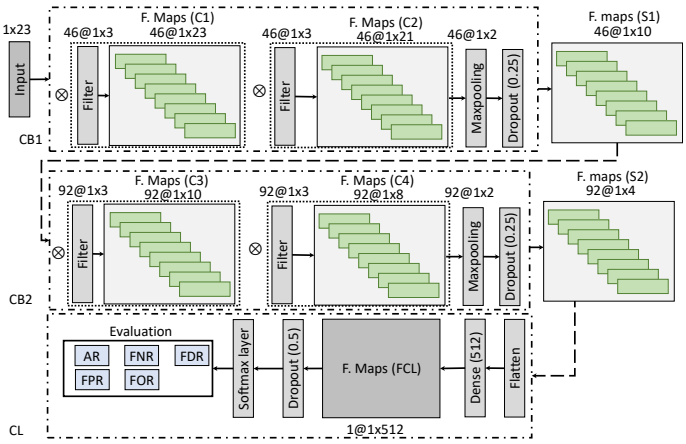


Fig. 9: The internal design of the architecture with a 1D convolutional neural network of multiple layers followed by a softmax classifier and used for our the detection task in this work. Notice that 46@1x3, for example, stands for "applying 46 filters, each of size 1x3 on the input data.

the classifiers, which are shown in table III and table IV. For evaluation, we use the following defined metrics. For classes C1 and C2: True Positive (TP) is all C1 classified correctly, True Negative (TN) is all C2 classified as C2, False Positive (FP) is all C2 classified as C1, and False Negative (FN) is all C1 classified as C2. Moreover, the Accuracy Rate (AR), False Discovery Rate (FDR), False Positive Rate (FPR), False Omission Rate (FOR), and False Negative Rate (FNR) are calculated as follows:

$$\begin{aligned} \text{AR} &= [(\text{TP}+\text{TN})/(\text{TP}+\text{FP}+\text{FN}+\text{TN})] \times 100 \\ \text{FDR} &= (\text{FP}/(\text{FP}+\text{TP})) \times 100 \\ \text{FPR} &= (\text{FP}/(\text{FP}+\text{TN})) \times 100 \\ \text{FOR} &= (\text{FP}/(\text{FN}+\text{TP})) \times 100 \\ \text{FNR} &= (\text{FN}/(\text{FN}+\text{TN})) \times 100 \end{aligned}$$

C. System Flow

For IoT samples, we extract 23 different features from general and algorithmic characteristics of the CFGs, and categorize them into seven groups as shown in figure 2. Five different features are extracted from each of the four feature categories of average closeness centrality, average betweenness centrality, average degree centrality, and average shortest paths represent minimum, maximum, median, mean, and standard deviation values for the extracted parameters. Other remaining features are the nodes count, edges count, and density. These features are used to train machine/deep learning-based models, including LR, SVM, RF, and CNN. The performance of these models is evaluated based on 10-fold cross validation method, which highlights the generalizability of the trained models. Furthermore, standard metrics such as AR, FNR, FPR, FDR, and FOR are used to evaluate the model performance.

D. Comparison, Discussion, and Evaluation

CFG of a program represents the flow of control from a source to an exit node. A CFG can be exploited by adversaries to reveal details pertaining to the nature of programs, specifically, the flow of control, the flow of functions, the start and the exit nodes, the functions that force a program to go into infinite loop, etc. Moreover, it also gives the user a hint about packing and obfuscation. In this study, we conduct an empirical study of the CFGs corresponding to 5,853 malware samples of IoT and Android. We generate the CFGs to analyze and compare the similarities and differences between the two highly prevalent malware types using different graph algorithmic properties to compute various features.

Comparison. Based on the above highlights of the CFGs, we observe a major difference between the IoT and Android malware in terms of the nodes and edges count, which are the main evaluation metric of the graph size. Our results show that unlike the Android samples, the IoT malware samples are more likely to contain a lesser number of nodes and edges. Even though around 4.4% of the IoT malware, or 131 samples, have less than 20 nodes and 31 edges, we notice they have various file sizes ranging from around 1,100 to 1,000,000 bytes per sample. This finding can be interpreted by the use of different evasion techniques from the malware authors in order to prevent analyzing the binaries statically.

With a high number of nodes and edges in Android malware, we observe that the CFGs of 86.16%, or 2,491 Android samples, have more than one component, marking unreachable functions, perhaps as a sign of using decoy functions or obfuscation techniques to circumvent static analysis. In addition, the prevalence of unreachable code indicates the complexity of the Android malware: these malware samples have a file size ranging from 12K to 114M bytes, which is quite large in comparison to the IoT malware (1.1K - 1M bytes).

Discussion. After analyzing different algorithmic graph structures, we observe a major variation between the IoT and Android malware graphs. We clearly notice a cut-off value for the density, average closeness, average betweenness, diameter, radius, average shortest path, and degree centrality for both datasets that can be applied to the detection system and reach an accuracy range of around 65% – 90% based on the feature vector being applied. For example, the cut-off points for the closeness centrality can set apart 65% of the malware, while the density of graphs can differentiate 90% of the IoT malware and Android malware. We notice that those differences in properties are a direct result of the difference in the structural properties of the graphs, and can be used for easily classifying different types of malware based on their distinctive features.

In most of the characterizations we conducted by tracing the distribution of the properties of the CFGs of different malware samples and types, we notice a slow growth in the distribution curve of the Android dataset, whereas a drastically increase for the IoT dataset. These characteristics show that the Android malware samples are diverse in their characteristics with respect to the measured properties of their graphs, whereas the IoT malware is less diverse. We anticipate that due to the emergence of IoT malware, and expect that characteristic to change over time, as more malware families are produced. We also observe that the IoT malware samples are denser than Android malware. As shown in figure 5a, we observe that 4 Android malware have a density equal to 2.

By examining those samples, we found that they utilize an analysis circumvention technique resulting in infinite loops. Moreover, we found 32 Android samples, as in figure 7a, with a degree centrality greater than one, and CFGs that contain 3 – 32 nodes with file sizes ranging between 29,400 – 3,000,000 byte, where the parent node leads to a child loop operating in another sign of infinite loop which may be because of obfuscation of the other functions.

Our analysis shows the power of CFGs in differentiating Android from IoT malware. It also demonstrates the usefulness of CFGs as a simple high-level tool before diving into lines of codes. We correlate the size of malware samples with the size of the graph as a measure of nodes and edges. We observe that even with the presence of low node or edge counts, the size of malware could be very huge, indicative of obfuscation.

Evaluation. While there have been many studies on the usage of CFGs for malware detection, understanding the uniqueness and difference of CFGs corresponding to different malware still unexplored. Sun *et al.* [49] use component based CFGs to detect code reuse in Android application with a detection rate of 96.60% for malware variants. Bruschi *et al.* [20] propose a strategy to detect metamorphic malicious codes in a program by comparing the CFG of the program with that of CFG of known malware. In this study, to evaluate our detection system, we extracted the CFGs of 2,999 benign IoT samples to build a detection system for the IoT environment utilizing a similar insight: CFGs of benign and malicious samples differ significantly, and we can base our machine learning algorithms on those differences for detection.

We implemented four machine learning techniques for: 1) detection of IoT malware, and 2) classification of malware families. The main goal of the detection models is to identify whether a sample is benign or malicious. The goal of the classification models is to label each sample to one of the following classes: *benign*, *gafgyt*, *mirai*, or *tsunami*. Moreover, we evaluate our classification in terms of several standard evaluation metrics, e.g., AR, FNR, FPR, etc. (detailed list of these metrics are provided in figure VI-B).

For our evaluation, we use the 10-fold cross-validation method to generalize our results. In this method, we partition the dataset into 10 equal portions, where the model is trained over nine portions and then tested over the remaining portion. This process is repeated ten times until all of the portions are evaluated as test data, and the average result is reported.

The detailed results of our malware detection and classification models are listed in table III and table IV, respectively. We observed that all of the models are able to reach a high detection accuracy. Specifically, the CNN model detects IoT malware from benign samples with an accuracy rate of 99.66% with FNR and FPR of 0.33%. Furthermore, we found that, in general, all models are able to achieve high classification metrics. In particular, the CNN model is able to correctly classify IoT malware families with an accuracy rate of 99.32% with FNR of 2.93% and FPR of 0.45%.

Feature reduction. Although, there has been substantial work on feature reduction based on features' discriminative power [50], [51], in our study, and due to the limited number of initial features (only 23) we do not need such feature reduction. Additionally, although we might be able to score the deep features extracted by the convolutional layers for reduction, these features will lack interpretability.

TABLE III: Results of IoT malware detection. Here, FNR, FPR, FDR, FOR, and AR are percentages.

Model	FNR	FPR	FDR	FOR	AR
LR	3.66	1.35	1.36	3.64	97.47
SVM	3.32	1.35	1.35	3.32	97.65
RF	2.33	0.67	0.67	2.33	98.48
CNN	0.33	0.33	0.33	0.33	99.66

TABLE IV: Malware family-level classification of IoT samples. Here, FNR, FPR, FDR, FOR, and AR are percentages.

Model	FNR	FPR	FDR	FOR	AR
LR	8.88	1.79	12.27	2.05	97.22
SVM	10.53	1.78	13.09	2.01	97.23
RF	5.14	1.03	7.35	1.20	98.40
CNN	2.93	0.45	2.17	0.44	99.32

Future Work. Although CFG-based features are shown in this work to detect IoT malware with high accuracy, these features are vulnerable to obfuscation. For example, a function-level obfuscation of the IoT malware might lead to an increase in the number of components, reduced flow of control and reduced complexity, which will affect the accuracy of our detection system. Certain program-level obfuscations will prevent obtaining a CFG altogether. Moreover, our approach does not assume adversarial inputs that may attempt to tamper with the guarantees of the deep learning architecture, as shown in [52]. We notice that regardless of the static obfuscation, once executed the malware has to expose its true contents and behavior, loading the unobfuscated code in memory. The unobfuscated code can be then extracted using dynamic analysis for our detector. Addressing those issues with other static features and using them for detection, addressing adversarial learning attacks to harden defenses, and using dynamic analysis in tandem with static analysis for comprehensive CFGs are our future work.

VII. Conclusion

In this paper, we build a detection model to detect IoT malware by augmenting features generated from Control Flow Graphs (CFGs). Towards this, we conduct an in-depth graph-based analysis of three different datasets, namely, Android malware, IoT malware, and IoT benign samples, to highlight the similarity and differences between Android and IoT malware, and to build a detection system for the emerging IoT malware. Toward this goal, we first extract the CFGs as an abstract representation to characterize them across different graph features. We highlight interesting findings by analyzing the shift in the graph representation from the Android malware to the IoT malware. We observe decoy functions for circumvention. Toward IoT malware detection, we utilize various features extracted from the CFGs of IoT benign and malware datasets, such as the closeness, betweenness, and density to build a deep learning-based detection system. We evaluate the detection model by leveraging four different classifiers and achieve an accuracy rate of ≈99.66% with 0.33% FNR and 0.33% FPR using CNN. Moreover, we classify the IoT

malware based on their families and achieve an accuracy of ≈99.32% with 2.93% FNR and 0.45% FPR.

REFERENCES

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

[2] A. Gerber. (2018) Connecting all the things in the Internet of Things. Available at [Online]: https://ibm.co/2qMx97a.

[3] Z. Ling, J. Luo, Y. Xu, C. Gao, K. Wu, and X. Fu, "Security vulnerabilities of Internet of Things: A case study of the smart plug system," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 1899–1909, 2017.

[4] A. Wang, A. Mohaisen, W. Chang, and S. Chen, "Revealing DDoS attack dynamics behind the scenes," in *Proceedings of Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA*, 2015, pp. 205–215.

[5] K. Zhang, X. Liang, R. Lu, and X. Shen, "Sybil attacks and their defenses in the Internet of Things," *IEEE Internet of Things Journal*, vol. 1, no. 5, pp. 372–383, 2014.

[6] M. Frustaci, P. Pace, G. Aloi, and G. Fortino, "Evaluating critical security issues of the IoT world: Present and future challenges," *IEEE Internet of Things Journal*, vol. 5, no. 4, pp. 2483–2495, 2018.

[7] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for Internet of Things: A survey." *IEEE Internet of Things Journal*, vol. 3, no. 1, pp. 70–95, 2016.

[8] Y. Yang, L. Wu, G. Yin, L. Li, and H. Zhao, "A survey on security and privacy issues in Internet-of-Things," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1250–1258, 2017.

[9] L. Harrison. (2015) The Internet of Things (IoT) vision. Available at [Online]: https://bit.ly/2SrowO1.

[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, "AV-Meter: An evaluation of antivirus scans and labels," in *Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA*, 2014, pp. 112–131.

[12] S. Shang, N. Zheng, J. Xu, M. Xu, and H. Zhang, "Detecting malware variants via function-call graph similarity," in *Proceedings of the 5th International Conference on Malicious and Unwanted Software, MALWARE*, 2010, pp. 113–120.

[13] 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.

[14] X. Hu, T. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the ACM Conference on Computer and Communications Security, CCS*, 2009, pp. 611–620.

[15] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proceedings of the 12th USENIX Security Symposium*, 2003, pp. 169–186.

[16] 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*, 2013, pp. 164–181.

[17] 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.

[18] 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.

[19] J.-w. Jang, J. Woo, A. Mohaisen, J. Yun, and H. K. Kim, "Mal-Netminer: Malware classification approach based on social network analysis of system call graph," *arXiv preprint arXiv:1606.01971*, 2016.

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

[21] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the IEEE Symposium on Security and Privacy, S&P*, 2014, pp. 590–604.

[22] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proceedings of the ACM Workshop on Artificial Intelligence and Security, AISec*, 2013, pp. 45–54.

[23] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual API dependency graphs," in *Proceedings of the ACM Conference on Computer and Communications Security, CCS*, 2014, pp. 1105–1116.

[24] H. Ham, H. Kim, M. Kim, and M. Choi, "Linear SVM-Based android malware detection for reliable IoT services," *Journal of Applied Mathematics*, vol. 2014, pp. 594 501:1–594 501:10, 2014.

[25] J. Milosevic, M. Malek, and A. Ferrante, "A friend or a foe? detecting malware using memory and CPU features," in *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications*, 2016, pp. 73–84.

[26] 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*, vol. 24, pp. 522–533, 2016.

[27] J. Su, D. V. Vargas, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai, "Lightweight classification of iot malware based on image recognition," in *Proceedings of the 42nd IEEE Annual Computer Software and Applications Conference, COMPSAC*, 2018, pp. 664–669.

[28] D. Wei and X. Qiu, "Status-based detection of malicious code in Internet of Things (IoT) devices," in *Proceedings of the IEEE Conference on Communications and Network Security, CNS*, 2018, pp. 1–7.

[29] M. Hossain, R. Hasan, and S. Zawoad, "Probe-IoT: A public digital ledger based forensic investigation framework for IoT," in *IEEE Conference on Computer Communications Workshops, INFOCOM*, 2018.

[30] S. Shen, L. Huang, H. Zhou, S. Yu, E. Fan, and Q. Cao, "Multistage signaling game-based optimal detection strategies for suppressing malware diffusion in fog-cloud-based IoT networks," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1043–1054, 2018.

[31] 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 *26th USENIX Security Symposium*, 2017, pp. 1093–1110.

[32] C. Kolias, G. Kambourakis, A. Stavrou, and J. M. Voas, "DDoS in the IoT: Mirai and other Botnets," *IEEE Computer*, vol. 50, no. 7, pp. 80–84, 2017.

[33] M. D. Donno, N. Dragoni, A. Giaretta, and A. Spognardi, "DDoS-capable IoT malwares: Comparative analysis and Mirai investigation," *Security and Communication Networks*, vol. 2018, pp. 7 178 164:1–7 178 164:30, 2018.

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

[35] F. Shen, J. D. Vecchio, A. Mohaisen, S. Y. Ko, and L. Ziarek, "Android malware detection using complex-flows," in *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems, ICDCS*, 2017, pp. 2430–2437.

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

[37] ——. (2019) Radare2. Available at [Online]: https://https://rada.re/r/.

[38] ——. (2019) VirusTotal. Available at [Online]: https://www.virustotal.com.

[39] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVClass: A tool for massive malware labeling," in *19th International Symposium on Research in Attacks, Intrusions, and Defenses, RAID*, 2016.

[40] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied logistic regression*, 2013, vol. 398.

[41] 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.

[42] A. Verikas, A. Gelzinis, and M. Bacauskiene, "Mining data with random forests: A survey and results of new tests," *Pattern Recognition*, vol. 44, no. 2, pp. 330–349, 2011.

[43] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[44] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of Advances in neural information processing systems*, 2012, pp. 1097–1105.

[45] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI*, 1995, pp. 1137–1145.

[46] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi, "Exposure: Finding malicious domains using passive dns analysis." in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2011, pp. 1–17.

[47] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang, "Large-scale and language-oblivious code authorship identification," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2018, pp. 101–114.

[48] S. Lee and J. Kim, "Warningbird: Detecting suspicious urls in twitter stream," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium, NDSS*, 2012, pp. 1–13.

[49] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, "Detecting code reuse in android applications using component-based control flow graph," in *ICT Systems Security and Privacy Protection*, 2014.

[50] T. Zhu, Z. Qu, H. Xu, J. Zhang, Z. Shao, Y. Chen, S. Prabhakar, and J. Yang, "Riskcog: Unobtrusive real-time user authentication on mobile devices in the wild," *IEEE Transactions on Mobile Computing*, 2019.

[51] D. Koller and M. Sahami, "Toward optimal feature selection," Stanford InfoLab, Tech. Rep., 1996.

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