Graph-based Comparison of IoT and Android Malware

Hisham Alasmary[1], Afsah Anwar[1], Jeman Park[1],
Jinchun Choi[1,2], Daehun Nyang[2], and Aziz Mohaisen[1,3]

[1] University of Central Florida, Orlando, FL 32816, USA
[2] Inha University, Incheon, Republic of Korea
[1]hisham,afsahanwar,parkjeman,jc.choi@Knights.ucf.edu,
[2]nyang@inha.ac.kr,[3]mohaisen@ucf.edu

Abstract. The growth in the number of android and Internet of Things (IoT) devices has witnessed a parallel increase in the number of malicious software (malware) that can run on both, affecting their ecosystems. Thus, it is essential to understand those malware towards their detection. In this work, we look into a comparative study of android and IoT malware through the lenses of graph measures: we construct abstract structures, using the control flow graph (CFG) to represent malware binaries. Using those structures, we conduct an in-depth analysis of malicious graphs extracted from the android and IoT malware. By reversing 2,874 and 201 malware binaries corresponding to the IoT and android platforms, respectively, extract their CFGs, and analyze them across both general characteristics, such as the number of nodes and edges, as well as graph algorithmic constructs, such as average shortest path, betweenness, closeness, density, etc. Using the CFG as an abstract structure, we emphasize various interesting findings, such as the prevalence of unreachable code in android malware, noted by the multiple components in their CFGs, the high density, strong closeness and betweenness, and larger number of nodes in the android malware, compared to the IoT malware, highlighting its higher order of complexity. We note that the number of edges in android malware is larger than that in IoT malware, highlighting a richer flow structure of those malware samples, despite their structural simplicity (number of nodes). We note that most of those graph-based properties can be used as discriminative features for classification.

Keywords: Malware; Android; IoT; Graph Analysis.

1 Introduction

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 [1]. Each of those devices runs multiple pieces of software, or applications, which increase in complexity, could have vulnerabilities that could be exploited, resulting in various security threats and consequences. As a result, understanding IoT software through analysis, abstraction, and classification is an essential problem to mitigate those security threats [1,2].

There has been a large body of work on the problem of software analysis in general, and a few attempts on analyzing IoT software in particular. However, the effort on IoT software analysis has been very limited with respect to the samples analyzed and the approaches attempted. Starting with a new dataset of IoT malware samples, we pursue a graph-theoretic approach to malware analysis. Each malware sample can be 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 software it is (e.g., malware family level classification and label extrapolation).

The limited existing literature on IoT malware, and despite malware analysis, classification, and detection being a focal point of analysts and researchers [3,4,5,6], points at the difficulty, compared to other malware type. Understanding the similarity and differences of IoT malware compared to other prominent malware type will help analysts understand the differences and use them to build detection systems upon those differences. To understand how different the IoT malware is from other types of emerging malware, such as mobile applications, we perform a comparative study of those graph-theoretic features in both types of software to highlight the control flow graph shift in IoT malware to android application malware.

Contributions. In this paper, we make the following contributions. First, building on the existing literature of mobile apps analysis and abstraction using CFGs, we look into analyzing CFGs of emerging and recent IoT malware samples. Then, using various graph-theoretic features, such as degree centrality, betweenness, graph size, diameter, radius, distribution of shortest path, etc., we contrast those features in IoT malware to those in mobile applications, uncovering various similarities and differences. Therefore, the findings in this paper can be utilized to distinguish between IoT malware and android malware.

Organization. The rest of this paper is organized as follows. In section 2 we review the related work. In section 3 we introduce the methodology and approach of this paper, including the dataset, data representation and augmentation, control flow graph definition, and graph theoretic metrics. In section 4 we present the results. In section 5 we present discussion and comparison, followed by concluding remarks in section 6.

2 Related Work

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.* [7] 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.* [5] 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 [8] 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.* [9] detected the self-mutated malware by comparing the control flow graph of the malware code to the control flow graphs for other known malware.

Tamersoy *et al.* [10] 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.* [11] 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. Caselden *et al.* [12] 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.

Wuchner *et al.* [13] 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. In addition, Jang *et al.* [14] used a behavioral representation of the programs as quantitative data flow graphs to classify the malware families based on their system call structures by using multiple graph characteristics, such as degree centrality, graph density, etc., as a feature vector.

Android malware. Gascon *et al.* [15] detected android malware through classifying their function call graphs. They found the reuse of malicious codes across multiple malware samples showing that malware authors reuse existing codes to infect the android applications. Zhang *et al.* [16] proposed a detection system for the android malware by constructing signatures through classifying the API dependency graphs and used that signature to uncover the similarities of android applications behavior.

3 Methodology

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 various algorithmic and structural properties of those graphs to understand code complexity, analysis evasion techniques (decoy functions, obfuscation, etc.).

Towards this goal, we start by gathering various malware samples in two datasets, IoT and android. For our IoT dataset, we utilized samples gathered through the IoTPOT honeypot [17]. For our android dataset, various recent android malware samples, due to Shen *et al.* (obtained from a security analysis vendor) are utilized [18]. For our analysis, 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.

3.1 Dataset Creation

Our IoT malware dataset is a set of 2,874 malware samples, randomly selected from the IoTPOT [17], a telnet-based honeypot which is now extended to other services.

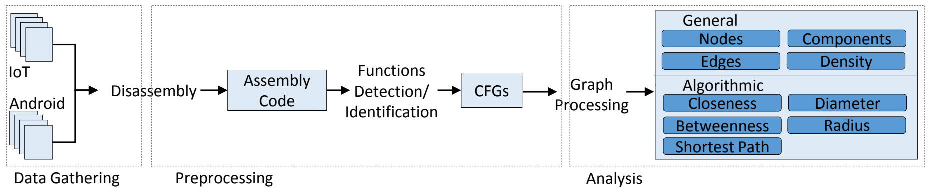

Fig. 1: Data flow diagram for the analysis process for the CFGs.

Additionally, we also obtained a dataset of 201 android malware samples from [18] for contrast. These datasets represent each malware type. We reverse-engineered the malware datasets using *Radare2* [19], a reverse engineering framework that provides various analysis capabilities including disassembly. To this end, we disassemble the IoT binaries, which in the form of Executable and Linkable Format (ELF) binaries, as well as the Android Application Packages (APKs) using the same tool, *radare2*. *Radare2* 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* [20] 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 while the android malware samples are detected by almost every antivirus software, the IoT malware has a low detection rate, which is perhaps anticipated given the fact that the IoT malware samples are recent and emerging threats, with fewer signatures populated in the antivirus scanners, compared to well-understood android malware. To further examine the diversity and representation of the malware in our dataset, we label them by their family (class attribute). To do so, we use *AV-Class* [21], a tool that ingests the *VirusTotal* results and provides a family name to each sample through various heuristics of label consolidation. Table 1 shows the top seven family labels and their share in both the IoT and android malware datasets. Overall, we noticed that the IoT malware belong to seven families, while the android malware belong to 39 unique families, despite the clear imbalance in the number of samples.

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. 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 1 shows the analysis workflow we follow to perform our analysis.

Program Formulation. We use the CFGs of the different malware samples as abstract characterizations of programs for their analysis. 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 $V = \{v_1, v_2 \dots, v_n\}$ 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

Table 1: Top 7 android and IoT families with their number of malware samples.

Android Family	# of samples	IoT Family	# of samples
Smsreg	72	Gafgyt	2,609
Smspay	34	Mirai	185
Dowgin	14	Tsunami	64
Zdtad	9	Singleton	7
Kuguo	9	Hajime	7
Revmob	8	Lightaidra	1
Smsthief	6	Ircbot	1

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 $v_i \rightarrow v_j$ and 0 otherwise.

3.2 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 code, which are the result of decoys and obfuscation techniques. 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 (Density). *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, such as: $Density = 1/n \sum_{i=1}^{n} \deg(v_i^{f_i})/n - 1$ for a benign graph. Other for the IoT and android graph are defined accordingly.*

Definition 2 (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 3 (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 $c_c = \sum_{\forall v_j \in V \not\backslash v_i} d(v_i, v_j)/n - 1$.*

Definition 4 (Betweenness centrality). *For a node $v_i \in V$, let $\Delta(v_i)$ be the total number of shortest paths that go through v_i and connect 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 $\Delta(v_i)/\Delta(.)$.*

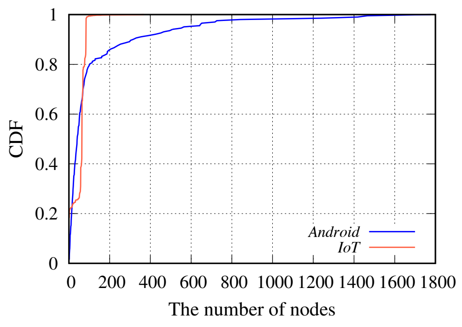

Fig. 2: CDF for the Nodes

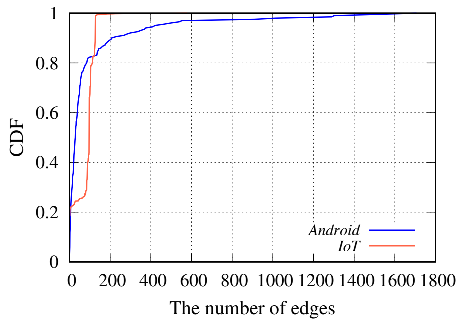

Fig. 3: CDF for the Edges

Definition 5 (Connected components). *A connected component in graph G is a subgraph in which two vertices are connected to each other by a path, 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 connected components.*

Definition 6 (Diameter and Radius). *The diameter of a graph $G = (V, E)$ is defined as the maximum shortest path length between any two pairs of nodes in the graph, while the radius is the smallest shortest path length among any two pairs of nodes in G. More precisely, let $d(v_i, v_j)$ be the shortest path length between two arbitrary nodes in G. The diameter is defined as $\max_{\forall i \neq j} d(v_i, v_j)$ while the radius is defined as $\min_{\forall i \neq j} d(v_i, v_j)$.*

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.

4 Results

4.1 General Analysis

Figures 2 and 3 show the difference between the android and IoT malware in terms of two major metrics of evaluation of graphs, namely the nodes and edges.
Nodes. It can be seen in figure 2 that the top 1% of the android and IoT malware samples have at least 1,777 and 367 nodes, respectively. We note that those numbers are not close to one another, highlighting a different level of complexity and the flow-level. In addition, as shown in Figure 2, we also 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: the dynamic region of the CDF is between 1 and 60 nodes (slow curve), corresponding to [0.2–0.3] of the CDF (this is, 10% of the samples have 1 to 60 nodes, which is a relatively small number). Furthermore, with the android malware, we notice that a large majority of the samples (almost 60%) have around 100 nodes in their graph. This characteristic seems to be unique and distinguishing, as shown in Figure 2.

Edges. Figure 3 represents the top 1% of the android and IoT malware samples, 1,707 and 577 edges, respectively, which shows a great difference between them. In particular, this figure shows that differences at the edges count as well. 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 1 to 85 edges correspond to [0.2–0.4] (about 20% of the samples). Additionally, we notice that the smallest 60% of the android samples (with respect to their graph size) have 40 edges whereas the same 60% of the IoT samples have around 95 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.

Density. Figure 4 shows the density of the datasets, where we notice almost 90% of the IoT samples have a density around 0.05 whereas the android samples have a diverse range of density over around 0.15. 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 0.07, for example, we can successfully tell the different types of malware apart with an accuracy exceeding 90%.

Graph Components. Figure 5 shows a boxplot illustration of the number of components in both the IoT and android malware's CFGs. We notice that 3.23% of the IoT malware, corresponding to 93 IoT samples, have more than two components, which indicates that a large percentage of the IoT samples have one component that represents the whole control graphs for the samples. These samples have a range of file sizes from 56,500 – 266,200 Bytes. We notice that 526 (18.3%) of the IoT samples, on the other hand, have only one node, with file sizes in the range of around 2,000 – 350,000 bytes.

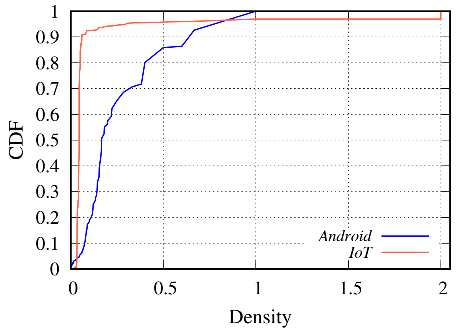

Fig. 4: The distribution of density. Notice that the density is discriminative, where one can tell the two types of malware apart with high accuracy (90%) for a fixed density.

The android malware have a large number of components. We find that 4.47%, or 9 android samples, have only one component, where their size ranges from around 16,900 – 240,900 bytes. On the other hand, 192 samples (95.5%) 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

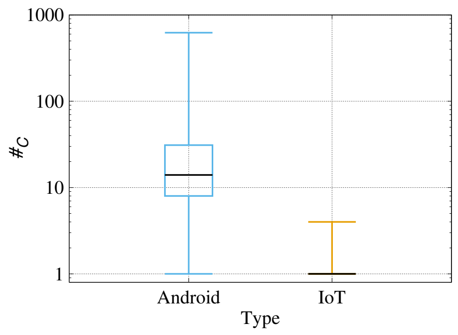

Fig. 5: The distribution of the number of components in CFGs. Notice that $\#_C$ means the number of components. The box represents the distribution from the upper quartile to the lower quartile, and the black bar represents the median value.

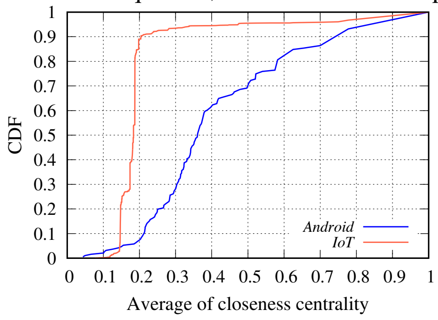

Fig. 6: The average of closeness centrality in the largest component of each sample.

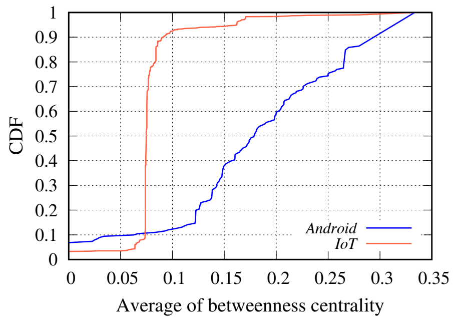

Fig. 7: The average of betweenness centrality in the largest component of each sample.

such, we consider the largest component of these samples for the further CFG-based analysis. However, we notice that 20 android samples have the same node counts in the first and second largest components. Furthermore, we find 14 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 range from 0 – 5, but the file sizes range from around 118,000 – 3,300,000 bytes.

Root Causes of Unreachable Code / Components. Figure 5 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 8 and 18, with median of 14 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 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.

4.2 General Algorithmic Properties and Constructs

Closeness. Figure 6 depicts the CDF for the average closeness centrality for both datasets. To reach this plot, we generalize the definition in 3 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 0.19, highlighting that the closeness alone can be used as a distinguishing feature of the two different types of malware.
Betweenness. Figure 7 shows the average betweenness centrality for both the datasets. The average betweenness is defined by extending 4 in a similar way to extending the closeness definition. Similar to the closeness centrality, 10% of the IoT and android samples have almost 0.07 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 0.08 average betweenness that shows a slight increase when covering a large portion of the IoT samples. This huge gap that we notice in Figure 6 and 7 is quite surprising although explained by correlating the density of the graph to both the betweenness and the closeness: the android samples tend to have a higher density, thus an improved betweenness, which is not the case of the IoT.
Diameter, radius, and average shortest path. Figure 8 shows the diameter of the graphs. Almost 15% of the IoT samples have a diameter of around 12 that can be noticed from the slow growth in the CDF, whereas the android malware have around 0.1. After that, there is a rapid increase in the CDF curve for the diameter in the 80% of both samples, reaching 9 and 17 for the android and IoT, respectively. Similarly, Figure 9 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 6. In addition, 80% of the android samples have around 4 while the IoT have around 8. 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, where we can use the value of 10 for the diameter and 5 for the radius to tell different malware types apart.

Figure 10 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 less than 5.

5 Discussion and Comparison

We conduct an empirical study of the CFGs corresponding to 3,075 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.

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.

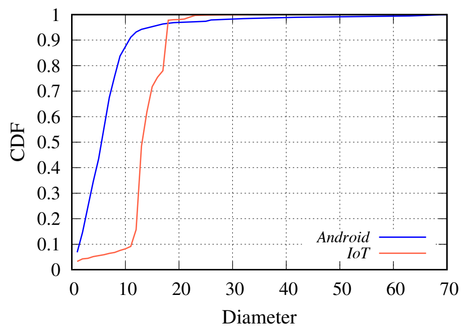

Fig. 8: The distribution of diameter.

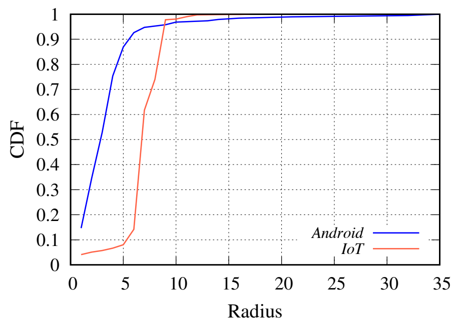

Fig. 9: The distribution of radius.

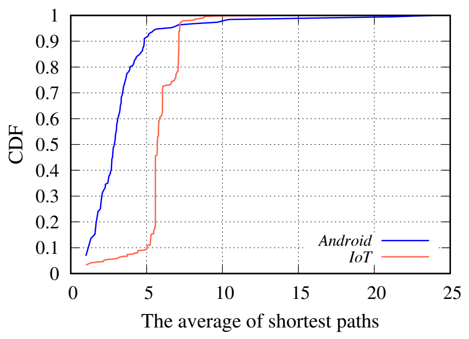

Fig. 10: The average of the lengths of shortest paths.

Even though around 21% of the IoT malware, or 603 samples, have less than two nodes and edges, we notice they have various file sizes ranging from around 2,000 to 350,000 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. We notice these malware samples correspond to only one component except for one malware sample that corresponds to two components.

With the high number of nodes and edges in the android malware, and unlike the IoT samples, we observe that the CFGs of almost 95.5%, or 192 android samples, have more than one component, which shows that the android malware often uses unreachable functions. This is shown when using multiple entry points for the same program, and the multiple components (unreachable code) is a sign of using decoy functions or obfuscation techniques to circumvent the 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 118,500 to 29,000,000 bytes, which is quite large in comparison to the IoT malware (2k-350k, as shown above).

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, and average shortest path for both datasets that can be applied to the detection system and reach an accuracy range around 80% – 90% based on the feature vector being applied. 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, and showing 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 the android malware. As shown in Figure 4, we observe that 75 IoT malware, or almost 2.6%, have a density equal to 2. By examining those samples, we found that they utilize an analysis circumvention technique resulting in infinite loops.

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.

6 Conclusion

In this paper, we conduct an in-depth graph-based analysis of the android and IoT malware to highlight the similarity and differences. Toward this goal, we extract malware 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 IoT to the android malware and tracing size (nodes, edges, and components). We observe decoy functions for circumvention, which correspond to multiple components in the CFG. We further analyze algorithmic features of those graphs, including closeness, betweenness, and density, which all are shown to be discriminative features at the malware type level, and could be used for classification.

Acknowledgement. This work is supported by the NSF grant CNS-1809000, NRF grant 2016K1A1A2912757, Florida Center for Cybersecurity (FC2) seed grant, and support by the Air Force Research Lab. This work would not have been possible without the support of Ernest J. Gemeinhart.

References

1. A. Gerber. (Retrieved, 2017) Connecting all the things in the Internet of Things. Available at [Online]: https://ibm.co/2qMx97a.
2. L. Harrison. (Retrieved, 2015) The Internet of Things (IoT) vision. Available at [Online]: https://blog.equinix.com/blog/2015/03/12/the-internet-of-things-iot-vision/.
3. 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.

4. 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.
5. 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.
6. 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.
7. 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.
8. M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proceedings of the 12th USENIX Security Symposium*, 2003.
9. 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.
10. 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.
11. 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, SP*, 2014, pp. 590–604.
12. 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.
13. 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.
14. 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," *Mathematical Problems in Engineering*, 2015.
15. 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.
16. 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.
17. Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "IoTPOT: A novel honeypot for revealing current IoT threats," *Journal of Information Processing, JIP*, vol. 24, pp. 522–533, 2016.
18. 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.
19. Developers. (Retrieved, 2018) Radare2. Available at [Online]: https://www.radare.org/r/.
20. Developers. (Retrieved, 2018) VirusTotal. Available at [Online]: https://www.virustotal.com.
21. M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVClass: A tool for massive malware labeling," in *Proceedings of the 19th the International Symposium on Research in Attacks, Intrusions and Defenses, RAID*, 2016, pp. 230–253.