

ANALYZING AND DETECTING INTERNET OF THINGS MALWARE USING RESIDUAL
STATIC GRAPH- AND STRING-BASED ARTIFACTS

by

HISHAM ALASMARY
B.S. King Khalid University, KSA, 2009
M.S. The George Washington University, USA, 2016

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2020

Major Professor: David Mohaisen

© 2020 Hisham Alasmary

ABSTRACT

Recently, the Internet of Things (IoT) has become wider and adopted many features from social networks and mainly uses sensing devices technologies, causing a rapid increase in production and adoption. However, security and privacy are serious threats that users usually take precautions to protect their devices and information. Thus, understanding the security shortcomings at first stage will educate IoT users to protect their connected things. Understanding IoT software through analysis, comparison (with other types of malware), and detection (from benign IoT) is an essential problem to mitigate security threats. We focus on two central perspectives, the graph and string representations of the software, typically extracted from the software binaries. First, we look into a comparative study of Android and IoT malware through the lenses of graph measurements. We construct the abstract structures of the malware, using Control Flow Graph (CFG) to represent malware binaries, and use them to conduct an in-depth analysis of malicious graphs. Machine Learning (ML) algorithms are actively used in the process of detecting and classifying malicious software. Toward detection, we use different CFG-based features as mentioned above, and augment them with CFGs of the benign dataset and build a detection system. Furthermore, we classify the IoT malware to their corresponding families. However, adversarial ML attacks on malware detectors are proposed in the literature. For example, Adversarial Examples (AEs) on the CFG can be generated by applying small perturbation to the graph features that force the model to misclassification. Thus, we propose Soteria, a CFG-based AEs detector utilizing deep learning with random walks to construct in-depth features. Moreover, we detect the malicious shell commands by extracting and analyzing the malicious commands of IoT malware. We utilize Natural Language Processing (NLP) for feature generation, followed by a deep learning model to detect malicious commands, hence detecting malware samples.

This dissertation is dedicated to my beloved wife Amani, my kids Taim and Ghaim, my parents,
and my wife's parents for their support and encouragement.

ACKNOWLEDGMENTS

I would like to thank Allah Almighty for giving me the strength, knowledge, ability and opportunity to undertake this research journey. This work would not have been possible without the continued support, advice, and contribution of many people that I would like to thank for their help.

First and foremost, I would like to thank my major advisor, Dr. Aziz Mohaisen, for his contribution to my journey at UCF, including his advice, guidance, and training. I have learned from Aziz all I know about researching computer security, from framing research questions, to technical writing, and technical presentation. His persistence and occasional push to move forward are to be remembered for life!

This work has significantly benefited from the insightful feedback I obtained from my committee members, Dr. Amro Awad, Dr. Clay Posey, and Dr. Wei Zhang, who questioned various assumptions, and helped me position my work in an easily accessible framework/flow. I would like to thank them for taking the time to serve on my committee, as well.

I have been fortunate to collaborate closely with a vibrant research group, the Security Analytics Research Lab (SEAL), who contributed to this work being co-authors, uncredited contributors (through proofreading my work), and critics for various ideas we had in the process of realizing this dissertation. In particular, I would like to thank my collaborators (not in any particular order): Afsah Anwar, Ahmed Abusnaina, Mohammed Abuhamad, RhongHo Jang, Jinchun Choi, Jeman Park, and Muhammad Saad. From the SEAL's family, I would like to thank Ashar Ahmad, Sultan Alshamrani, Ulku Meteriz, Necip Yildiran, and Abdulrahman Alabduljabbar for being nice and supportive colleagues.

This work would not have been possible by the generous sponsorship of King Khalid University in Abha, Saudi Arabia, and the Saudi Arabian Cultural Mission (SACM) in Washington, D.C., USA.

In addition, part of this work was also supported by the National Science Foundation, the National Research Foundation of Korea, NVIDIA, and CyberFlorida.

TABLE OF CONTENTS

LIST OF FIGURES	xii
LIST OF TABLES	xvi
CHAPTER 1: INTRODUCTION	1
1.1 Statement of Research	3
1.2 Need for Work and Scope	5
1.3 Contributions	7
1.4 Dissertation Organization	7
CHAPTER 2: GRAPH-BASED COMPARISON OF IOT AND ANDROID MALWARE	9
2.1 Motivation	9
2.2 Related Work	10
2.3 Dataset	12
2.3.1 Dataset Creation	12
2.4 Methodology	15
2.4.1 Graph Algorithmic Properties	16
2.5 Results	18
2.5.1 General Analysis	18
2.5.2 General Algorithmic Properties and Constructs	23

2.6	Comparison and Discussion	27
2.7	Conclusion	29
CHAPTER 3: ANALYZING AND DETECTING EMERGING INTERNET OF THINGS		
	MALWARE: A GRAPH-BASED APPROACH	30
3.1	Motivation	30
3.2	Related Work	31
3.3	Dataset	32
3.4	IoT Detection	32
	3.4.1 Detection Algorithms	34
	3.4.2 Evaluation Metrics	38
	3.4.3 System Flow	39
	3.4.4 Evaluation	39
3.5	Conclusion	41
CHAPTER 4: SOTERIA: DETECTING ADVERSARIAL EXAMPLES IN CONTROL FLOW		
	GRAPH-BASED MALWARE CLASSIFIERS	43
4.1	Related Work	44
4.2	Background and Motivation	46
	4.2.1 Practical Adversarial Examples	46
	4.2.2 Limitation of Adversarial Learning	48
	4.2.3 Graph Embedding and Augmentation	48

4.2.4	Threat Model	49
4.3	System Design	50
4.3.1	High-Level Architecture	50
4.3.2	Adversarial Examples Detector	52
4.3.2.1	Sample Pre-processing	52
4.3.2.2	Feature Representation	54
4.3.2.3	Building Detection Model	55
4.3.3	Classifier	56
4.3.3.1	CNN Classifiers	57
4.3.3.2	Majority Voting	58
4.4	Dataset and Evaluation	58
4.4.1	Dataset	58
4.4.2	Feature Analysis	60
4.4.3	Evaluation and Analysis	63
4.4.3.1	Adversarial Example Detector	63
4.4.3.2	Classifier	66
4.5	Discussion	69
4.6	Conclusion	71
CHAPTER 5: SHELLCORE: DEFEATING IOT MALWARE THROUGH IN-DEPTH ANALYSIS AND DETECTION OF IOT SHELL CODE		72

5.1	Motivation	72
5.2	Related Work	74
5.3	Background	77
5.4	Dataset and Approach Overview	79
5.4.1	Dataset and Data Processing	79
5.4.1.1	Malicious Dataset Creation	80
5.4.1.2	Assembling a Benign Dataset	82
5.4.2	Methodology: High-Level Overview	85
5.5	Detection Model	87
5.5.1	Featurization	87
5.5.1.1	Traditional NLP-based model	87
5.5.1.2	Customized NLP-based model	88
5.5.2	Feature Representation	88
5.5.3	Feature Reduction	90
5.5.4	Classification Methods	91
5.6	Evaluation and Discussion	92
5.6.1	Traditional NLP-based model	94
5.6.2	Customized NLP-based model	94
5.6.3	Detecting Malicious Commands	95
5.6.4	Malware Detection	97
5.6.5	Discussion	98

5.7 Conclusion 99

CHAPTER 6: CONCLUSION AND FUTURE WORK 101

APPENDIX: COPYRIGHT INFORMATION 103

LIST OF REFERENCES 109

LIST OF FIGURES

2.1	Android and IoT malware detection rate on VirusTotal.	14
2.2	Pipeline of analysis and detection using CFGs. A.I. (Assembly Instructions). .	15
2.3	CFG of a malware highlighting unreachable codes, depicting use of decoy or obfuscation techniques in malware.	16
2.4	The logarithmic scale distribution of the number of nodes.	18
2.5	The logarithmic scale distribution of the number of edges.	19
2.6	The distribution of density.	20
2.7	The distribution of components.	21
2.8	Average of closeness centrality	22
2.9	Average of betweenness centrality	23
2.10	Average of degree centrality.	24
2.11	The distribution of diameter.	25
2.12	The distribution of radius.	26
2.13	Average shortest paths.	27
3.1	Data flow diagram of the CFG-based detection model for the IoT malware. Note that the detection model phase has been deployed at the end of the analyzer phase to build our detector. 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).	31

3.2	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.	37
4.1	GEA adversarial approach. The CFG in figure 4.1(c) is obtained by embedding the CFG of a selected sample in figure 4.1(b) into the CFG of the original sample in figure 4.1(a). It can be done by injecting the code directly. .	49
4.2	The architecture of Soteria. IoT samples are fed to the feature extraction process, where each sample is represented by multiple feature vectors. The feature vectors are forwarded to adversarial example detector. All non-AEs are then forwarded to the classifier to be classified into its corresponding family.	50
4.3	Soteria feature extraction process. IoT samples binaries are disassembled to extract their corresponding CFGs. Then, two nodes labeling techniques are used (Dense-based and level-based), then, several random walks are done over each labeled graph. The trace of the random walk is then used for feature extraction by using n -grams with TF-IDF.	51
4.4	Graph labeling using two approaches, density- and level-based. Each node has a label in $[0, V - 1]$, where $ V $ is the number of nodes in G . Figures 4.4(b) and 4.4(d) show the labeling of the GEA generated CFG over the original graphs in figures 4.4(a) and 4.4(c), respectively.	53

4.5	The proposed AEs detector: The detector consists of five fully connected layers auto-encoder. The input to the auto-encoder is density- and level-based feature vectors, where the output is the reconstructed feature vectors. A validation unit is used to calculate the reconstruction error. A sample is considered as AE if reconstruction error exceeds a threshold.	56
4.6	The structure of Soteria classifiers. The classifiers consist of four convolutional layers with max-pooling and dropout functions. The output of the classifier is the softmax probability of each class.	57
4.7	Soteria classification process. The CNN-based models' inputs are the dense- and level-based feature vectors. The classification decision is the majority vote of the CNN classifiers output probabilities over the feature vectors. . . .	58
4.8	The PCA comparison between the benign and malware families using features used in Alasmary <i>et al.</i> [8].	61
4.9	Soteria: Dense-based labeling feature vector comparison. Figure 4.9(a) shows the PCA distribution of benign and malware samples. Figure 4.9(b) shows the PCA distribution comparison between the normal and GEA generated AEs.	62
4.10	Soteria: Level-base labeling feature vector comparison. Figure 4.10(a) shows the PCA distribution of benign and malware samples. Figure 4.10(b) shows the PCA distribution comparison between the normal and GEA generated AEs.	63
4.11	Soteria: Combined labeling feature vector comparison. Figure 4.11(a) shows the PCA distribution of benign and malware samples. Figure 4.11(b) shows the PCA distribution comparison between the normal and GEA generated AEs.	64

4.12	Reconstruction Error (RE) comparison between normal and the generated AEs. Figure 4.12(a) shows the distribution frequency of the RE among the normal and adversarial samples. Figure 4.12(b) represents the accumulated frequencies of samples and their corresponding RE. The vertical dashed line is the chosen threshold for Soteria AEs detector.	65
4.13	Effect of varying the detector threshold (α) on the detection error. The selected α in Soteria is the intersection between the error rates of normal and adversarial samples.	67
5.1	Retrieving a list of target hosts.	73
5.2	The high-level workflow of ShellCore. The malware contains HTTP requests and commands. We gather the benign dataset by crowd-sourcing and sniffing over an IoT network. Upon extracting the shell commands (<i>i.e.</i> the commands and the HTTP requests), we featurize them as n-grams. We then perform feature reduction, followed by a machine learning-based malware detection system.	79
5.3	Monitoring stations for benign dataset creation. Two network implementations are used: NAT and a home network.	83
5.4	Accuracy variations over the 10 fold cross-validation when detecting malicious commands.	96

LIST OF TABLES

2.1	Dataset: Top 3 Android and IoT families.	14
3.1	Summary of the related works represented in Sections 2.2 and 3.2. Abbreviations: SVM (Support Vector Machine), CNN (Convolutional Neural Network), NB (Naive Bayes), LR (Logistic Regression), DT (Decision Tree-based J48), and RF (Random Forest).	33
3.2	Results of IoT malware detection. Here, FNR, FPR, FDR, FOR, and AR are percentages.	40
3.3	Malware family-level classification of IoT samples. Here, FNR, FPR, FDR, FOR, and AR are percentages.	40
4.1	Current adversarial attacks defenses. The focus of the adversarial attacks defenses is on AEs in the context of image classification. Note: MLP is Multilayer Perceptron, DNN is Deep Neural Network, and RNN is Recurrent Neural Network.	47
4.2	Distribution of IoT samples across benign and malicious families. Gafgyt is the most popular IoT family with 66.18% of the dataset samples, while Tsunami is the least popular with only 262 samples (1.55% of the samples). The dataset is split into the train (80%) and test (20%) subsets.	59
4.3	GEA selected targeted samples. These samples are used to generate the AEs to evaluate Soteria. Three samples from each class are selected of different sizes (number of nodes), <i>i.e.</i> small, medium, and large.	60

4.4	Distribution of dense- and level-based feature vectors extracted by <i>n-grams</i> technique from the random walk traces among the IoT benign and malware classes.	61
4.5	GEA: Detector Performance over adversarial samples. The detector was able to detect an overall percentage of 97.79% of the AEs. DE refers to the detected samples.	66
4.6	GEA: Detector Performance over clean samples. Only 6.16% of the clean samples were misclassified as AEs. All Benign clean samples passed the detector. DE refers to the detected samples (lower is better).	66
4.7	Classification performance of Soteria dense-, level-, and voting-based classification systems in classifying normal (non-adversarial) samples.	68
4.8	Soteria’s classifier predictions over AEs misdetected by the detector. Most of the misdetected samples are generated using GEA with large size selected samples.	68
5.1	Previous works for analysis and detection of the shell commands. Abbreviations: Area Under the Curve (AUC), True Positive Rate (TPR), True Negative Rate (TNR), Accuracy (AC), False-Negative Rate (FNR), False-Positive Rate (FPR), Natural Language Processing (NLP), and Convolutional Neural Networks (CNNs).	75
5.2	Malware dataset by architecture. Percentage is out of the total samples for the given architecture.	81

5.3	Data sources in our dataset. “Sources” is the number of files used to extract commands, while “Commands” is the total number of commands obtained from the source files.	84
5.4	Size characteristics of the different datasets. Net. stands for Network.	92
5.5	Evaluation results: Malicious commands detection.	94
5.6	Evaluation results of malware detection.	96

CHAPTER 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. This paradigm brings about many benefits, including a shift in lifestyles, as featured by home automation applications. IoT also facilitate the communication between smart objects together without the need for humans' participation [42]. The wide-range of IoT-based applications is featured by a rapid increase in production and adoption. Recent reports have shown that the number of IoT devices will be around 64 billion devices by 2025, with a global market between 4 and 11 trillion U.S. dollars in economic value [23]. The increasingly persistent connection between these IoT devices makes their role lie somewhere on the continuum between advantageous and susceptibility. Each of those devices runs multiple pieces of software, or applications, which are increasingly complex, and could have vulnerabilities that could be exploited, hence controlled to launch various attacks, such as Distributed Denial of Service (DDoS) attacks [74, 121, 132, 39]. For example, Dyn, a Domain Name System (DNS) infrastructure company, was targeted by a large amount of DNS lookup requests from millions of IoT devices, causing a significant impact on multiple web services globally [61, 66]. In another incident, baby monitoring systems are reportedly compromised and used for controlling the cameras, and delivering messages through associated speakers, violating the privacy of users, and putting them at risk [21]. Finally, Github experienced a DDoS attack with a peak bandwidth of 1.3 Terabits per second [88].

It is essential to analyze and understand the IoT software stack for malicious behavior detection in order to mitigated possible security threats and vulnerabilities. Existing malicious software analysis and detection mechanisms are impractical, for the resource constraints of the current IoT devices. Thus, it is worthwhile to explore new techniques for securing IoT environments, starting with IoT devices, and utilizing perhaps easy to obtain information from those devices/software.

A major reason for the susceptibility of IoT devices to attacks is the broad software system they utilize, merging components of close and open source software, and utilizing (at times) insecure functions and services. As such, adversaries exploit these vulnerable services (and functions) to deliver malware and to launch orchestrated attacks, such as the ones mentioned earlier. A viable approach towards understanding such IoT software is through program analysis, which is the key tool used in this dissertation.

Program analysis approaches utilized for malware include both static and dynamic analyses. Dynamic analysis requires executing malware for obtaining behavior features that are fed into machine learning algorithms for detection. Although the dynamic features are comprehensive, dynamic analysis techniques are subject to various shortcomings, and most importantly, their complexity and time consumption, resulting in poor scalability. Static analysis, on the other hand, does not require running programs, but relies on programs' contents, obtained from the static binary. A popular static analysis technique is using CFG to build a representative feature modality for malware detection, and is shown to be effective in various studies [65, 131]. Another example is the use of shell commands by malware by infecting the hosts using Command and Control (C2) servers to obtain payloads that include instructions to compromised machines (or bots) synchronizing their actions, including their cycles of activity by attacking targets, propagation by recruiting new bots and acting as a source of propagation, and by a stealthy operation to evade detection. In their operation,

Program analysis provides insight into the constructs of software, but alone does not address the key requirement for our problem scale: given the large number of IoT software samples, including those that are malicious, there is a need to automate decisions concerning whether such software is malicious or benign. Machine Learning (ML) and deep learning algorithms are actively used in the process of classifying and detecting malicious software from the benign ones [83, 85]. Generally, they are widely used in a wide range of applications, such as health-care [9], finance [58],

computer-vision [68], and cybersecurity [108, 29]. For instance, ML theory is leveraged into the process of software graph analysis to build more powerful analysis tools [13]. One such application is exploring IoT malware using both graph analysis and machine learning [7]. These models not only can learn the representative characteristics of the graph, but can also be utilized to build an automatic detection system to predict the label of the unseen software.

ML and deep learning models learn the inherent pattern of the input dataset. Therefore, the rise in the utilization of deep learning models in security-related domains creates incentives for adversaries to manipulate the underlying model to produce their desired outputs. It has been shown that the ML and deep learning networks are prone to vulnerabilities. For example, an adversary can force the model to produce his desired output, *e.g.* misclassification, through crafting the Adversarial Examples (AEs) [93, 87]. The AEs are being crafted by applying a small perturbation to the input dataset. Note that the crafted samples are very similar to the original ones, and are not necessarily outside of the training data manifold. Recently, researchers presented several algorithms for generating adversarial examples, such as the fast gradient sign method [44], DeepFool method [87], the Jacobin-based saliency map method [93].

1.1 Statement of Research

The limited existing literature on IoT malware, and despite malware analysis, classification, and detection being a focal point of analysts and researchers [85, 84, 105, 83], points at the difficulty, compared to other malware types. Thus, understanding IoT software through analysis, abstraction, and classification is an essential problem to mitigate those security issues. More specifically, understanding the similarity and differences of IoT malware compared to another prominent malware type, *i.e.* Android malware, will help analysts understand the differences and use them to build detection systems upon those differences. One such approach is a graph-theoretic analysis where each malware sample is abstracted into a Control Flow Graph (CFG), which could be used

to extract representative static features of the IoT 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.

Moreover, it should be noted that the research works on IoT software analysis and detection have been very limited not only in the size of the analyzed samples, but also the utilized approaches [14, 109]. A promising direction leverages a graph-theoretic approach to analyze and detect IoT malware. As IoT software can be represented using graph-based features from CFG, those features can be utilized to build an automatic detection system to identify whether a given software is malicious or benign. Moreover, the type of malicious software can be identified through malware family-level classification and label extrapolation, a concept widely applied [85]. We develop our next component of this work around the detection technique using ML algorithms over features extracted from abstract graph structures.

While deep learning algorithms bring about a great advantage in the classification and detection of such malware, they are also susceptible to various security vulnerabilities, such as adversarial ML (also known as adversarial examples; or AEs) [94, 2]. Adversarial ML aims to fool and mislead the ML models to generate wrong predictions by manipulating the input data, a process that is done by introducing minimal perturbations [44, 87, 46], resulting in targeted or non-targeted misclassification. Adversarial ML is an active research area, although there is a lack of research on studying the impact of adversarial ML on IoT malware detection, practical implications, and mitigations [3, 46], which we pursue as the next component in this dissertation.

Finally, with the unique characteristics of IoT devices and applications, there is an intrinsic need for developing new analysis and detection techniques that utilize new modalities. There has been some recent work in the past on analyzing shell codes, and using them to understand malicious end-points, although the majority of such work has been focused on other shell interpreters (e.g.,

Powershell and web shell). Motivated by such work, and given the emergence of Linux-based IoT malware that heavily utilize Linux shell, understanding and detecting the malicious use of Linux shell command in context is critical. For instance, IoT devices use a packed version of Linux libraries, called Busybox [125], to achieve Linux capabilities. Such attacks by well-known malware, such as Mirai and Tsunami, misuse the access to the device's shell to infect the host device, to propagate itself, and to launch much powerful attacks. For example, the Carna botnet hacked into 420,000 IoT devices, through default password scans, and was used to conduct an Internet-wide census [98]. Understanding how effective are shell codes obtained from the analysis of IoT malware in detecting them is essential.

1.2 Need for Work and Scope

It is to be noted that this work utilizes various advances in program analysis and machine learning to malware analysis and detection. There has been a large body of work on analyzing and detecting malicious software, *i.e.* Windows, Android, etc., in general. Different techniques were proposed, such as permissions, string analysis, etc.. For instance, Android malware has experienced high growth and spread rapidly in the last decade. Various Android applications that offer multiple services were targeted by different malware to take control of the Android devices. This is done due to the lack of underlying security mechanisms to design Android applications. Different malware analyses have been done toward analyzing Android platforms. One example is the static feature analysis that can be used to detect Android malware. Such analysis is permissions where the applications request frequent permissions from malware applications, unlike the benign ones [72, 126] that allow to access control to use the app that may raise other privacy concerns [38, 122]. These techniques are specific and cannot be extended and applied to the low-end IoT environment, such as a smart lightbulb that has computation constraints.

On the other hand, some other basic analysis techniques are proposed, *i.e.* CFG, to analyze and

detect different malware types. This technique provides the representation of the software during their execution. Some graph-based works have been done on Android, such as CFG [76], function call graphs [40, 55], and Application Programming Interface (API) dependency graphs [133]. Whether such techniques can be extended to IoT malware, how they compare to the state-of-the-art in other malware domains, and whether they can be used for detection is an open direction that we pursue in this dissertation.

In addition, adversarial attacks on malware detectors have recently been conducted by [3, 63, 67]. While Abusnaina *et al.* [3] shows the susceptibility of the CFG-based detectors to the adversarial attacks, the other works append bytes to the binary file. Both of these methods change the files while preserving the practicality and functionality of the clean IoT malware. Given those works, it is essential to understand defenses to those attacks on machine learning algorithms utilized for IoT malware detection using such modality to make the detection practical and sustainable.

Malware use C2 servers to obtain payloads that include instructions to compromised machines (or bots) synchronizing their actions, including their cycles of activity by attacking targets, propagation by recruiting new bots and acting as a source of propagation, and by a stealthy operation to evade detection. Thus, another artifact that can be used for malware analysis and detection is the malware strings. Malware strings are a way to provide hints and clues about the malware functionality, *i.e.* changing the privileges by executing *chmod* command on the shell. Android malware can be identified statically using a string-based approach. This analysis helps to extract, understand, and analyze the content of the string to be able to detect the malware using different features extracted from the strings [36, 123]. This as well constitute the fourth part of this dissertation.

All in all, our primary motivation is how we can generalize the graph-based approach as well as the shell commands to the IoT environment to detect the IoT malware. Then, due to the susceptibility of the ML and deep learning algorithms to the AEs on the CFG-based systems, we build a CFG-based detector against such attacks to detect the AEs. En route, we conduct a comparative analysis

between Linux-based IoT malware and Android counterparts to understand how CFG-based modeling differ in both cases.

1.3 Contributions

In this dissertation we make the following contributions:

1. We start by assembling a dataset of emerging and recent Android and IoT malware samples, and conduct an in-depth analysis of their CFGs.
2. Using various general and algorithmic features extracted from the CFGs, we uncover various findings to distinguish between IoT malware and Android malware.
3. Using the extracted discriminative features from the CFGs, we gathered a dataset of IoT benign samples, and build a CFG-based detection system for IoT malware.
4. Motivated by the recent work on developing AEs on machine learning-based malware detection models, we propose Soteria, a CFG-based model to detect AEs IoT malware.
5. Toward detecting IoT malware using shell command, we extracted the shell command from a dataset of 2,891 recent IoT malware, and a dataset of benign applications, and build a detection model capable of detecting malicious shell commands as well as malicious software.

1.4 Dissertation Organization

This dissertation encompasses material from four papers, three published papers by the author [5, 7, 8], and one another paper under submission [6]. Chapter 2 uses material from reference [7], coauthored with Afsah Anwar, Jeman Park, Jinchun Choi, DaeHun Nyang, and Aziz Mohaisen, which represent the comparative study and analysis between two prominent malware types in the

wild, namely IoT and Android malware, through the lens of the CFG representations of the software. Chapter 3 uses material from reference [8], coauthored with Aminollah Khormali, Afsah Anwar, Jeman Park, Jinchun Choi, Ahmed Abusnaina, Amro Awad, DaeHun Nyang, and Aziz Mohaisen, which represent the CFG-based IoT malware detection system to detect IoT malware. Chapter 4 uses material from reference [5], coauthored with Ahmed Abusnaina, RhongHo Jang, Mohammed Abuhamad, Afsah Anwar, DaeHun Nyang, and David Mohaisen, where we tackle the problem of generating AEs in the malware CFGs by building Soteria, a deep learning-based model that defends the CFG-based classifiers for malware detection against the AEs. Finally, Chapter 5 uses material from reference [6], coauthored with Afsah Anwar, Ahmed Abusnaina, Mohammed Abuhamad, and David Mohaisen, where a deep learning-based detection model is introduced to detect the malicious commands for the IoT malware.

CHAPTER 2: GRAPH-BASED COMPARISON OF IoT AND ANDROID MALWARE ¹

The goal of this chapter 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). Unique to this chapter; however, we look into various algorithmic and structural properties of those graphs to understand code complexity, analysis evasion techniques (decoy functions, obfuscation, etc.).

2.1 Motivation

From the perspective of the software, IoT software is different from well-understood ones on the other platforms, such as Android applications, Windows binaries, and their corresponding malware. Thus, we look into a comparative study of Android and IoT malware through the lenses of graph measures: we construct abstract structures, using the 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 2,962 and 2,891 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

¹This content was reproduced from the following article: Hisham Alasmay, Afsah Anwar, Jeman Park, Jinchun Choi, Daehun Nyang, and Aziz Mohaisen, "Graph-based Comparison of IoT and Android Malware," In Proceedings of the Seventh International Conference on Computational Data and Social Networks, (CSoNet), 2018.

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.

In this chapter, we make the following contributions:

1. 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.
2. Towards analyzing the CFGs, we disassemble a large number of samples. Namely, we use close to 6,000 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.
3. 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 chapter can be utilized to distinguish between IoT malware and Android malware.

2.2 Related Work

While prior works have analyzed the differences between the software or malware in general, there has been a few works on analyzing and detecting IoT malware in particular.

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.* [55] 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 find-

ing 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.* [105] 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 [24] 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.* [18] 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.* [115] 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.* [130] introduced the code property graph, which merges and combines different analyses of the code, such as abstract syntax trees, CFGs, and program dependence graphs in the form of joint data structure to efficiently identify common vulnerabilities. In addition, Caselden *et al.* [20] 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.* [127] 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.* [56] build a tool to classify malware by families based on the features generated from graphs.

Android Malware. Gascon *et al.* [40] 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.* [133] 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.* [50] detected Android malware using the Support Vector Machine (SVM). Milosevic *et al.* [80] 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.

2.3 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.).

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, and IoT malware samples. For our IoT malware dataset, we collected a new and recent IoT malware, up to late January of 2019, using CyberIOCs [30]. For our Android dataset, various recent Android malware samples were obtained from a security analysis vendor [106].

2.3.1 Dataset Creation

Our IoT malware dataset is a set of 2,962 malware samples, randomly selected from CyberIOCs [30]. Additionally, we also obtained a dataset of 2,891 Android malware samples from [106] for contrast. These datasets represent each malware type. We reverse-engineered the malware datasets using *Radare2* [32], a reverse engineering framework that provides various analysis capa-

bilities, 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*. 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* [33] 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. 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, as shown in figure 2.1.

To examine the diversity and representation of malware in our dataset, we label them by their family (class) using *AVClass* [103], 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 [7]. Moreover, we ignore IoT malware families with less than ten samples. We notice the IoT malware belong only to three families, while the Android malware belong to 180 unique families. The IoT malware families and top three Android families, with their share in their corresponding datasets are shown in table 2.1.

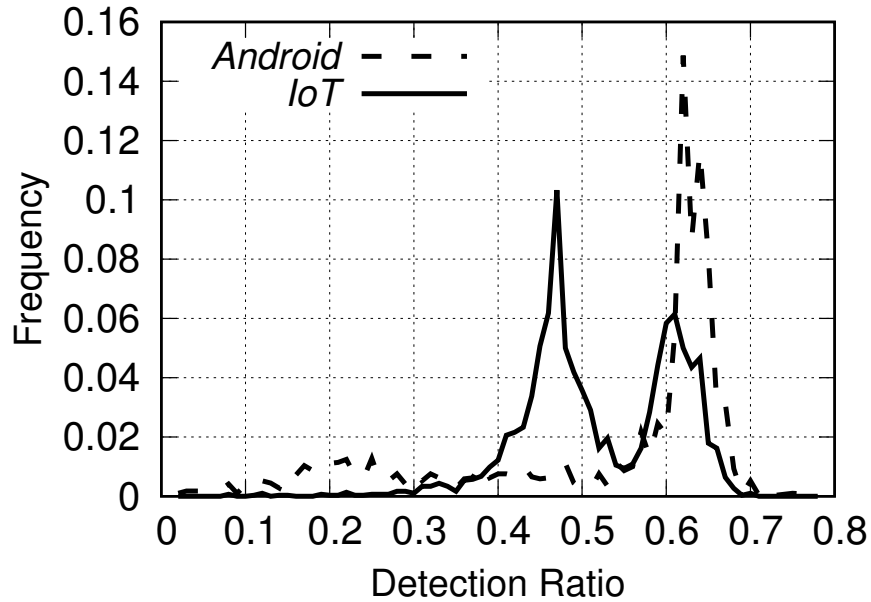


Figure 2.1: Android and IoT malware detection rate on VirusTotal.

Table 2.1: 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

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. . We start by 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

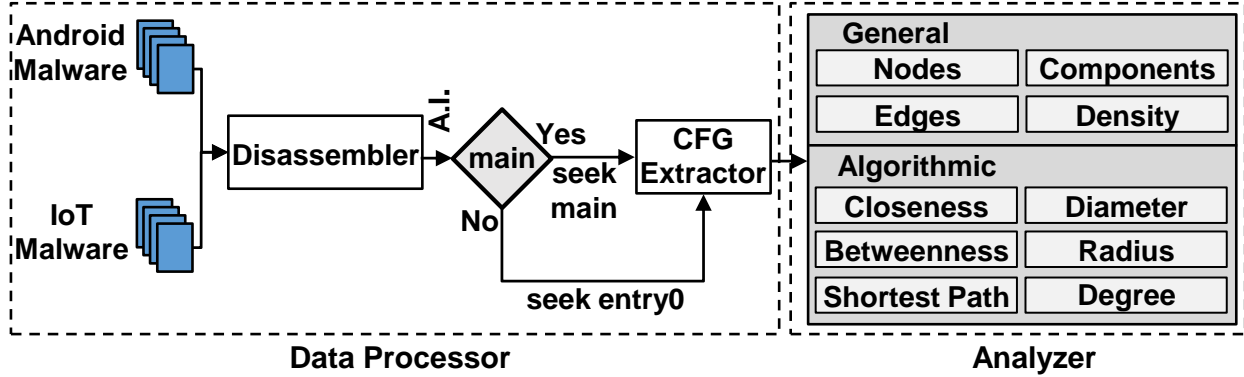


Figure 2.2: Pipeline of analysis and detection using CFGs. A.I. (Assembly Instructions).

without main from the entry point. Then, we use the extracted CFGs for further analysis. The analysis workflow we follow to perform our analysis is shown in figure 2.2.

2.4 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 $V = \{v_1, v_2 \dots, v_n\}$ 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 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.

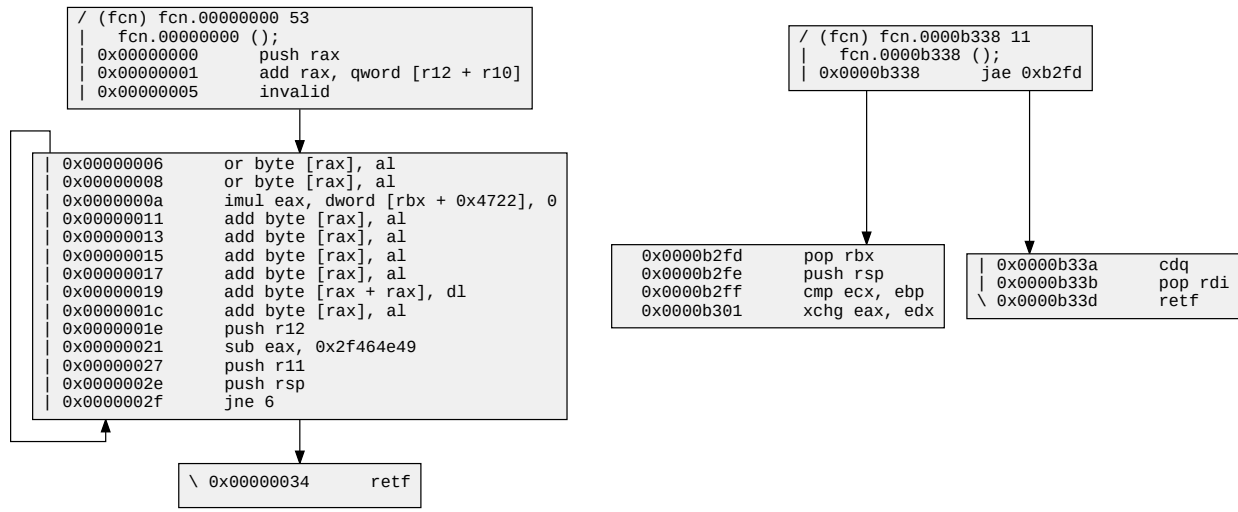


Figure 2.3: CFG of a malware highlighting unreachable codes, depicting use of decoy or obfuscation techniques in malware.

2.4.1 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. 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 later define the various measures used for our analysis. 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. This can be a result of obfuscating the parent node of the branching component, as can be observed in the example of the Android malware sample in figure 2.3. 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.

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 $V = \{v_1, v_2, \dots, v_n\}$.

Definition 3 (Shortest Path) For a graph $G = (V_i, E_i)$, the shortest path is defined as: $v_i^x, v_i^{x_1}, v_i^{x_2}, v_i^{x_3}, \dots, v_i^y$ 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 $c_c = \sum_{\forall v_j \in V \wedge v_i} d(v_i, v_j) / n - 1$.

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 $\Delta(\cdot)$ be the total number of shortest paths between such nodes. The betweenness centrality is defined as $\Delta(v_i) / \Delta(\cdot)$.

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.

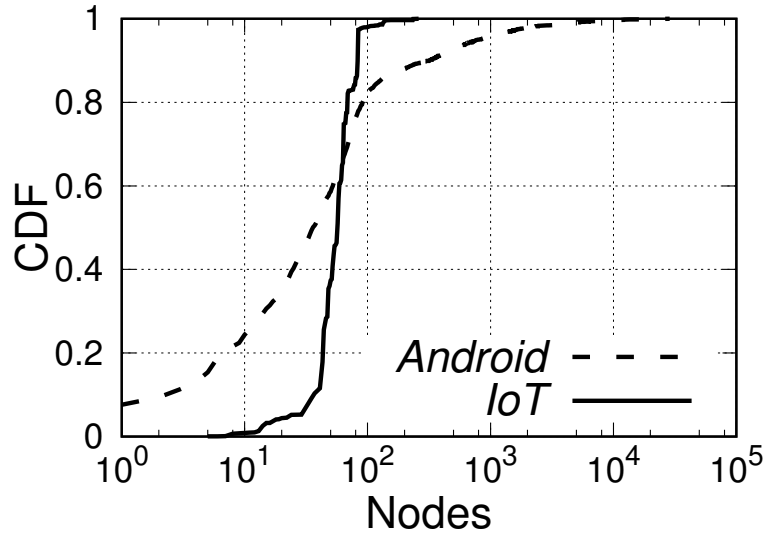


Figure 2.4: The logarithmic scale distribution of the number of nodes.

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 $\max_{v_i \neq v_j} d(v_i, v_j)$ while the radius is $\min_{v_i \neq v_j} d(v_i, v_j)$.*

In this chapter, 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.

2.5 Results

2.5.1 General Analysis

The logarithmic scale that 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, is depicted in figure 2.4 and figure 2.5.

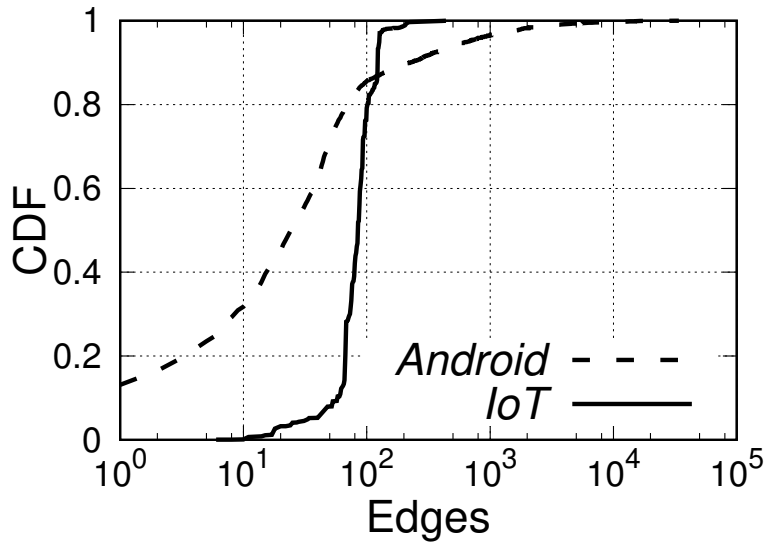


Figure 2.5: The logarithmic scale distribution of the number of edges.

Size Analysis: Nodes. The Android and IoT malware samples have at least 28,691 and 260 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, 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 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. The CDF logarithmic scale for the number of nodes in both malware datasets that highlights the percent change towards large node values of the Android samples is shown in figure 2.4.

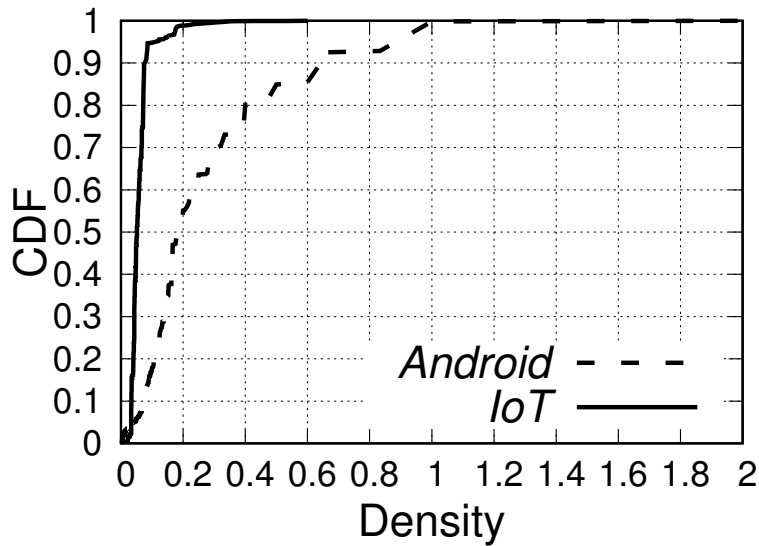


Figure 2.6: The distribution of density.

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. 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. The CDF logarithmic scale of the edges count for both malware datasets is represented in figure 2.5.

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.

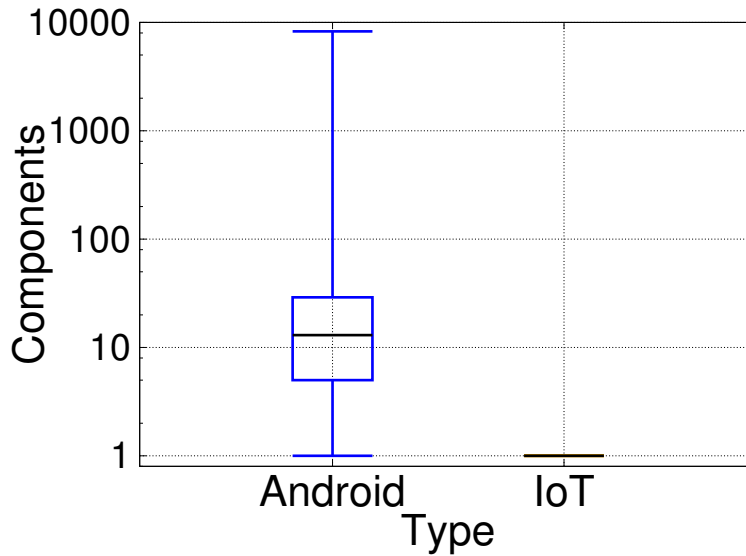


Figure 2.7: The distribution of components.

Graph Density Analysis. 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%. The density of the datasets is shown in figure 2.6.

Graph Components Analysis. 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

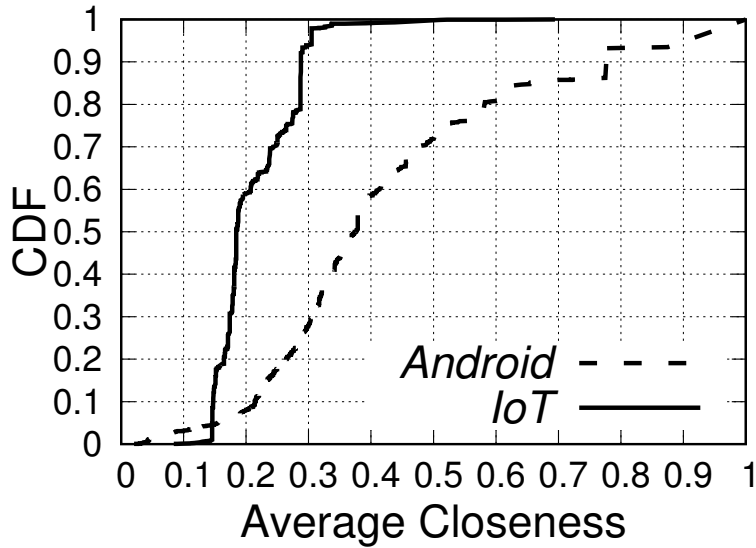


Figure 2.8: Average of closeness centrality

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 nodes 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. The illustration of the number of components in both the IoT and Android malware’s CFGs is depicted in figure 2.7.

Root Causes of Unreachable Code / Components. 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 the majority of the IoT platforms are constrained, limiting the number of functions (software-based). 2) The Android Operating System (OS) is advanced and can

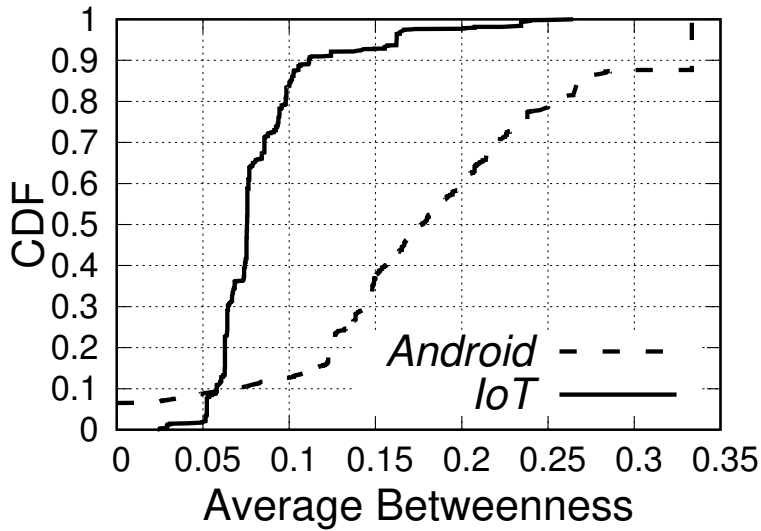


Figure 2.9: Average of betweenness centrality

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. The boxplot of the number of components that captures the median and 1st and 3rd quartile as well as the outliers for both the Android and IoT malware is shown in figure 2.7.

2.5.2 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 graph constructs. We elaborate on further analysis using those features in the following.

Graph Closeness Centrality Analysis. 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

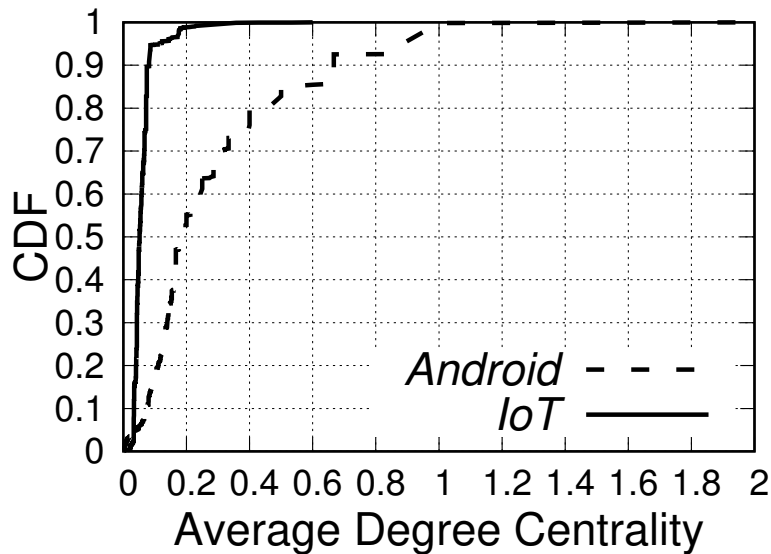


Figure 2.10: Average of degree centrality.

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%. The CDF for the average closeness centrality for both datasets is depicted in figure 2.8.

Graph Betweenness Centrality Analysis. 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. The CDF for the average betweenness centrality for both datasets is shown in figure 2.9.

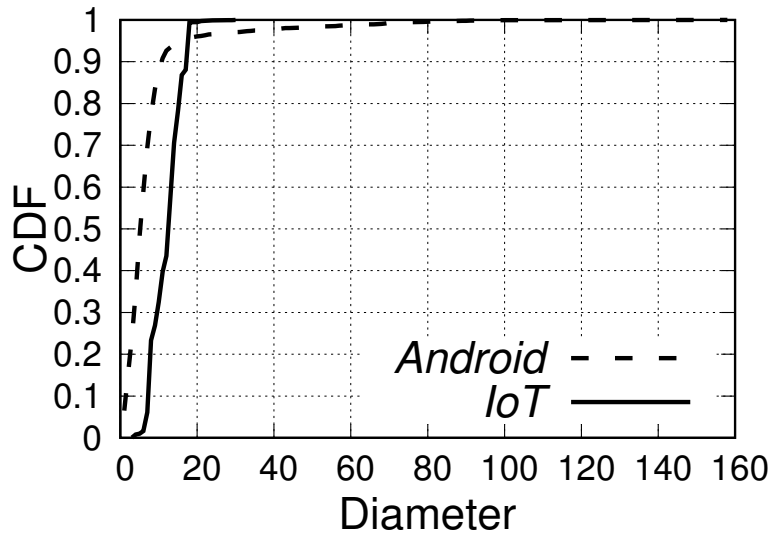


Figure 2.11: The distribution of diameter.

This huge gap 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 as shown in figure 2.8 and figure 2.9.

Graph Degree Centrality Analysis. 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. The average of degree centrality in the largest components is shown in figure 2.10.

Diameter, Radius, and Shortest Paths Analysis. 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. The CDF for the diameter of the graphs is depicted in figure 2.11.

After that, there is a rapid increase in the CDF curve for the diameter in the 80% of both samples,

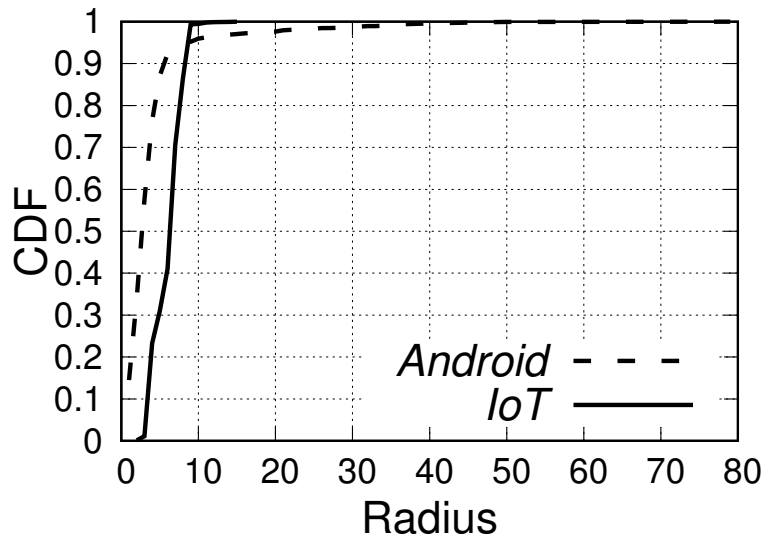


Figure 2.12: The distribution of radius.

reaching around 10 and 18 for the Android and IoT, respectively. 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 samples have around 7. This shows a 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. Similarly, the CDF of the radius of the graphs is shown in figure 2.12.

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. The CDF for the average shortest path of the graphs are represented in figure 2.13.

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.

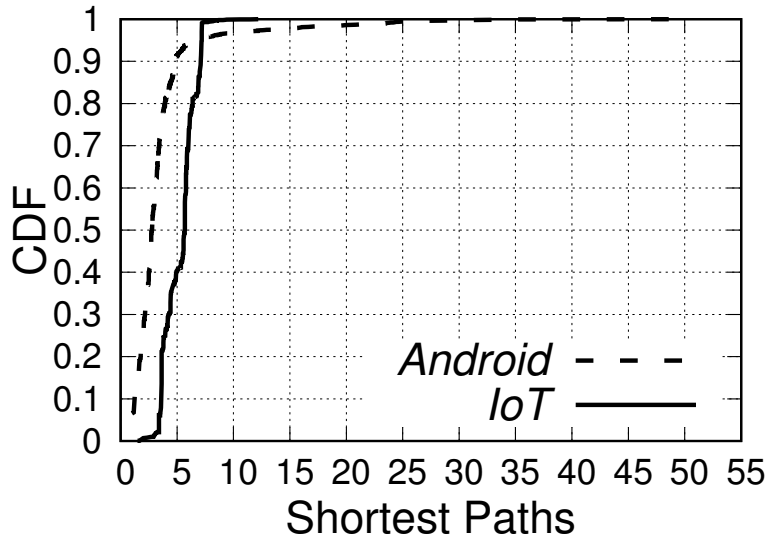


Figure 2.13: Average shortest paths.

2.6 Comparison and Discussion

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 an 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 the high number of nodes and edges in the Android malware, and unlike the IoT samples, we observe that the CFGs of almost 86.16%, or 2,491 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 perhaps 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 12,000 to 114,000,000 bytes, which is quite large in comparison to the IoT malware 1,100 - 1,000,000 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 drastic 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. 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, as shown in figure 2.6.

Moreover, we found 32 Android samples 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, as shown in figure 2.10.

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 huge, indicative of obfuscation.

2.7 Conclusion

We conduct an in-depth analysis of the general and algorithmic features of the CFGs of the Android and IoT malware datasets, including the number of nodes and edges, closeness, betweenness, and density, etc.. We highlight the shift in the graph representation from the IoT to the Android. We observe different tracing size of nodes, edges, and components. We also observe decoy functions for circumvention, which correspond to multiple components in the CFG. The general and algorithmic features of the graphs are shown to be discriminative features at the malware type level, so they will be used for classification and detection systems.

CHAPTER 3: ANALYZING AND DETECTING EMERGING INTERNET OF THINGS MALWARE: A GRAPH-BASED APPROACH ¹

In chapter 2, we provide a CFG-based analysis of the two prominent malware types, IoT and Android, to uncover the similarities and differences between them from the graph perspective. In this chapter, we extend the work by designing a deep learning-based detection system to detect IoT malware by utilizing various CFG features. The analysis workflow as well as the IoT detection flow system are shown in figure 3.1.

3.1 Motivation

The goal of this chapter is to use the aforementioned graph characteristics, the algorithmic and structural properties of the graphs, to build an IoT detection system to distinguish the malware from the benign binaries. 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). We make the following contributions:

1. Towards detecting the IoT malware based on the CFGs, we use the analyzed IoT malware dataset of 2,962 samples used in Section 2.3. Additionally, we assemble a dataset of 2,999 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.

¹This content was reproduced from the following article: Hisham Alasmay, Aminollah Khormali, Afsah Anwar, Jeman Park, Jinchun Choi, Ahmed Abusnaina, Amro Awad, Daehun Nyang, and Aziz Mohaisen, "Analyzing and Detecting Emerging Internet of Things Malware: A Graph-based Approach," In IEEE Internet of Things Journal, (IoT-J), 2019.

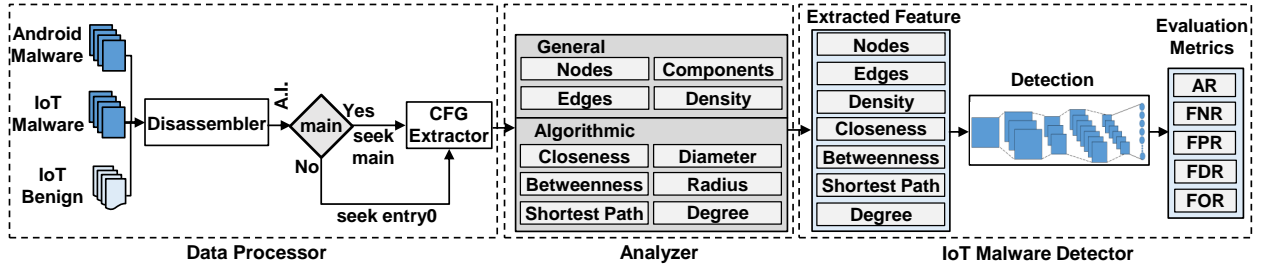


Figure 3.1: Data flow diagram of the CFG-based detection model for the IoT malware. Note that the detection model phase has been deployed at the end of the analyzer phase to build our detector. Abbreviations: A.I. (Assembly Instructions), AR (Accuracy Rate), FNR (False Negative Rate), FPR (False Positive Rate), FDR (False Discovery Rate), and FOR (False Omission Rate).

2. Using the different features as described in Section 2.4.1, 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 $\approx 99.32\%$.

3.2 Related Work

IoT Malware Detection. Pa *et al.* [91] proposed IoTPOT, an IoT honeypot and sandbox to analyze and capture IoT telnet-based attacks targeting IoT environment that run on multiple CPU architectures. Su *et al.* [112] 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 [124] 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.* [53] proposed an IoT forensic system, named Probe-IoT, that investigates IoT malicious behaviors using distributed digital ledger. Shen *et al.* [107] 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 example, Antonakakis *et al.* [12] analyzed Mirai botnets that launch DDoS attacks using IoT devices. Kolias *et al.* [61] 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.* [37] 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. The literature works (represented in Sections 2.2 and 3.2) on analyzing and detecting different malware types on different operating systems platforms using various approaches are highlighted in table 3.1.

3.3 Dataset

To accomplish the end goal of this chapter, we created a dataset of binaries and clustered them under two different categories: IoT malware samples and benign IoT samples. For the IoT malware samples, we used the aforementioned dataset of 2,962 IoT malware samples that we analyzed in section 2.3. Finally, to test our proposed IoT malware detector, we manually assembled a dataset of 2,999 benign samples from source files on GitHub [31]. For our analysis and detection, we augment the datasets by reversing the samples to address various analysis issues. Using an off-the-shelf tool, Radare2, we then disassemble the benign 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.

3.4 IoT Detection

This section is devoted to the detection of the IoT malware based on the CFGs features, as mentioned earlier (Section 2.4.1). 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 sam-

Table 3.1: Summary of the related works represented in Sections 2.2 and 3.2. Abbreviations: SVM (Support Vector Machine), CNN (Convolutional Neural Network), NB (Naive Bayes), LR (Logistic Regression), DT (Decision Tree-based J48), and RF (Random Forest).

Work	Platform	Dataset	Sample size	Task	Approach
[55]	x86	malware	102,391	Analysis	Function Call Graph
[105]	x86	malware, benign	51	Analysis	Function Call Graph
[24]	x86	malware, benign	14	Detection	Control Flow Graph
[20]	x86	benign programs	2	Analysis	Information Flow Graph, Control Flow Graph
[127]	x86	malware, benign	7,501	Detection	Quantitative Data Flow Graph
[115]	x86	malware, benign	43,353,581	Detection	File-Relation Graph
[56]	x86	malware, benign	3,768	Classification	System Call Graph
[18]	Linux	malware, benign	572	Analysis	Control Flow Graph
[130]	Linux	vulnerabilities	88	Analysis	Code Property Graph
[40]	Android	malware, benign	147,950	Detection	Function Call Graph / Machine Learning (SVM)
[133]	Android	malware, benign	15,700	Detection	Weighted Contextual API Dependency Graphs
[50]	Android	malware, benign	28	Detection	Machine Learning (SVM)
[80]	Android	malware, benign	2,199	Detection	Classifier (NB, LR, DT)
[91]	IoT	malware	106	Collection	IoT Honeypot
[112]	IoT	malware, benign	865	Detection	Deep Learning (CNN)
[124]	IoT	malware, benign	554	Detection	Algorithm
[53]	IoT	N/A	N/A	Forensic	Digital ledger (Blockchain)
[107]	IoT	malware	N/A	Detection	Theoretical analysis
[12]	IoT	malware	1,028	Analysis	Static analysis
[61]	IoT	malware	N/A	Analysis	Analyze Mirai source code
[37]	IoT	malware	N/A	Analysis	Analyze Mirai source code
Chapter 2	IoT, Android	malware	5,853	Analysis	Control Flow Graph
This Chapter	IoT	malware, benign	5,961	Detection	Control Flow Graph/ Deep Learning (CNN)

ples 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.

3.4.1 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:

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

where, x_1, x_2, \dots, x_n are the variables and $\beta_1, \beta_2, \dots, \beta_n$ 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 [52] 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:

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

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 hyperplane with the largest margin between the two classes.

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 [120]. 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 [17]. 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 the probability of 0.25 are applied, which results in a 2D tensor S_{b1} of size 10×46 .

$$C_{b1i}' = X \otimes F_{b1i}', \quad i = 1 : 46$$

$$C_{b1i}'' = C_{b1i}' \otimes F_{b1i}'', \quad i = 1 : 46$$

$$M_{b1i} = \text{maxpool}(C_{b1i}'', 2, 2), \quad i = 1 : 46$$

$$S_{b1i} = \text{dropout}(M_{b1i}, 0.25), \quad i = 1 : 46$$

- **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 .

$$C_{b2i}' = S_{b1} \otimes F_{b2i}', \quad i = 1 : 92$$

$$C_{b2i}'' = C_{b2i}' \otimes F_{b2i}'', \quad i = 1 : 92$$

$$M_{b2i} = \text{maxpool}(C_{b2i}'', 2, 2), \quad i = 1 : 92$$

$$S_{b2i} = \text{dropout}(M_{b2i}, 0.25), \quad i = 1 : 92$$

- **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

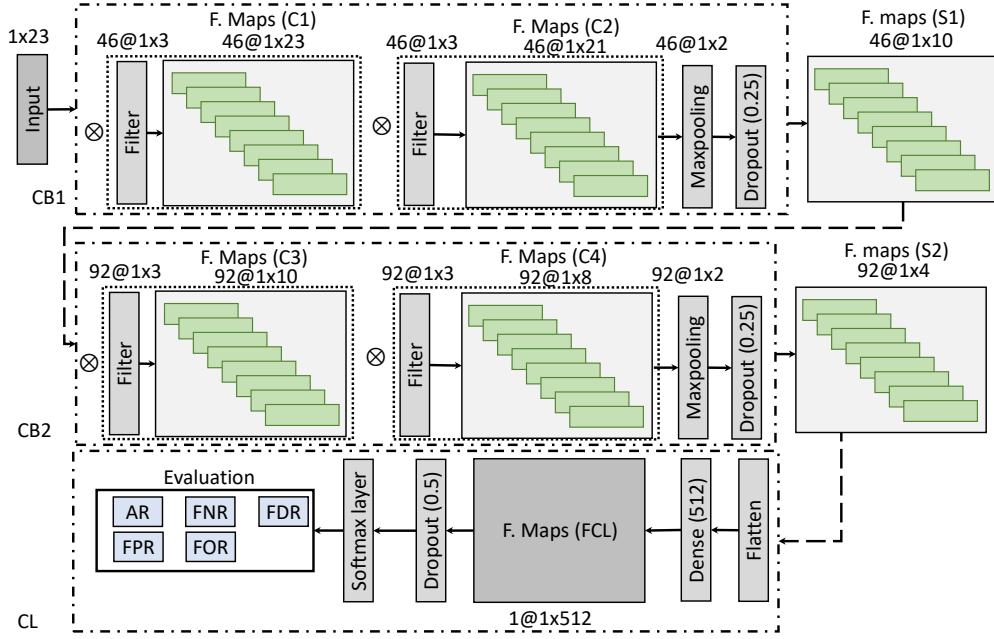


Figure 3.2: The internal design of the architecture with a 1D convolutional neural network of multiple layers followed by a softmax classifier and used for our the detection task in this work. Notice that 46@1x3, for example, stands for “applying 46 filters, each of size 1x3 on the input data.

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.

$$FCL = \text{dense}(\text{Flatten}(S_{b2}), 512))$$

$$S_{FC} = \text{dropout}(FCL, 0.5)$$

$$\text{output} = \text{softmax}(S_{FC})$$

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 lay-

ers use a Rectified Linear Units (ReLU) activation function. In addition, we used dropout to prevent model over-fitting. We refer the interested reader to [69] for more details on CNN internals. The architecture of the CNN design is shown in figure 3.2.

3.4.2 Evaluation Metrics

In order to investigate the generalization of the classifier, the K-fold cross validation method [60] is used. Although K is an unfixed parameter, K=10 is commonly used in the literature [16, 1, 71]. 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 the classifiers, which are shown in tables 3.2 and 3.3. 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:

$$AR = [(TP + TN)/(TP + FP + FN + TN)] \times 100$$

$$FDR = (FP/(FP + TP)) \times 100$$

$$FPR = (FP/(FP + TN)) \times 100$$

$$FOR = (FP/(FN + TP)) \times 100$$

$$FNR = (FN/(FN + TN)) \times 100$$

3.4.3 System Flow

For IoT samples, we extract 23 different features from the general and algorithmic characteristics of the CFGs, and categorize them into seven groups. 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's performance. The analysis workflow we follow to perform our analysis, as well as the IoT detection flow system is depicted in figure 3.1.

3.4.4 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.* [113] uses component-based CFGs to detect code reuse in Android application with a detection rate of 96.60% for malware variants. Bruschi *et al.* [18] proposes a strategy to detect malicious metamorphic 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) classi-

Table 3.2: 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 3.3: 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

fication 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 section 3.4.2).

For our evaluation, we use the 10-fold cross-validation method to generalize our results. In this method, we partition the dataset into ten 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.

We observed that all 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 clas-

sification 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%. The detailed results of our malware detection and classification models are listed in table 3.2 and table 3.3, respectively.

Feature reduction. Although, there has been substantial work on feature reduction based on features' discriminative power [134, 62], 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.

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 [3]. 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 then be 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.

3.5 Conclusion

In this chapter, 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

two different datasets, namely, IoT malware and IoT benign samples 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 then 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 $\approx 99.66\%$ 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 $\approx 99.32\%$ with 2.93% FNR and 0.45% FPR.

CHAPTER 4: SOTERIA: DETECTING ADVERSARIAL EXAMPLES IN CONTROL FLOW GRAPH-BASED MALWARE CLASSIFIERS ¹

Given that ML models' output depends on the input patterns, ML models can be prone to targeted attacks on their inputs. Particularly, an adversary may fool the models by applying perturbations to the input to generate Adversarial Examples (AEs) [3, 44, 54, 87]. Nevertheless, there have been several attempts to defend against the adversarial attacks on ML models by including the AEs in the training process [79]. Although prior works have shown the inefficiency of the malware detection models when subjected to adversarial examples, to the best of our knowledge, there is no work on defending such models from adversarial attacks. Identifying the research gap, with this work, we inch closer towards bridging the gap.

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

¹This content was reproduced from the following article: Hisham Alasmay, Ahmed Abusnaina, Rhongho Jang, Mohammed Abuhamad, Afsah Anwar, DaeHun Nyang, and David Mohaisen, "Soteria: Detecting Adversarial Examples in Control Flow Graph-based Malware Classifiers," In International Conference on Distributed Computing Systems, (ICDCS), 2020.

appropriate family label to the malware with an accuracy of 99.91%.

Contributions. In this chapter, we make two contributions:

1. Motivated by the recent body of work on developing adversarial examples on machine learning-based malware detection algorithms, we propose the design and implementation of Soteria, a system for detecting IoT malware. Similar to other efforts in this space, Soteria utilizes CFG based feature representations. Unique to Soteria, we use both density-based and level-based labels for CFG labeling, a random walk-based traversal approach for feature extraction, and n -gram based module for feature representation. End-to-end, Soteria's representation ensures a simple yet powerful randomization property of the used classification features, making it difficult even for a powerful adversary to launch a successful attack. Soteria also employs a deep learning approach, consisting of an auto-encoder for detecting AEs, and eliminating them from the classification process, and a CNN architecture for detecting and classifying malware samples.
2. We evaluate the performance of Soteria, using a large dataset consisting of 16,814 IoT samples, and demonstrate its superiority in comparison with state-of-the-art approaches. Soteria yields an accuracy rate of 97.79% for detecting AEs, and 99.91% overall accuracy for classification malware families.

4.1 Related Work

Machine and deep learning algorithms widely leveraged towards securing software against adversaries in general and detecting malware in particular. For instance, Alasmay *et al.* [8] analyzed two prominent malware, IoT and Android, based on the CFG-graph representation of the malicious software. Moreover, Alam *et al.* [4] analyzed the malware and proposed a malware detection system to detect malware with even small CFGs and then to address the changes that occurred in

the frequencies of opcodes. Bruschi *et al.* [18] proposed a malware detection method that uses two CFG techniques to compare and detect malware based on two CFGs of malware code and other known malware.

Several research works have been proposed to defend against adversarial machine learning. Most of these approaches are image-based methods. For example, Goodfellow *et al.* [44] proposed to train the model with a set of AEs to minimize the test error between the real and AEs of the model's result. Papernot *et al.* [92] designed a network distillation model to defend against adversarial attacks such as fast gradient sign method [44] and L-BFGS attack [114]. Cui *et al.* [27] introduced a malware detection method for malicious codes using deep learning by transferring the malicious code into grayscale images. Ni *et al.* [89] proposed a malware family classification system that converts malicious codes of nine different malware families into grayscale images. Metzen *et al.* [79] proposed a detection method for adversarial perturbation over trained AEs. Moreover, Rozsa *et al.* [99] proposed a machine learning model that tested the adversarial examples. They correlate their robustness of the three adversarial attacks to the accuracy of eight deep network classifiers. In addition, Miyato *et al.* [82] proposed a detection method on the text-domain. They trained the model over adversarial examples that apply small perturbation to the word that is embedded in RNN.

Several methods have been proposed to generate adversarial examples that can manipulate the desired output to fool the classifiers [3, 44, 54, 87]. Adversaries can make small modifications to the malware to misclassify them as benign, yet they remain malware files [29, 45]. Other methods apply and add small noise or perturbation to optimize the images to generate the adversarial examples [70, 44, 19]. For example, Carlini and Wagner [19] proposed three adversarial attacks against distilled neural networks that break many defenses models. Moreover, Moosavi-Dezfooli *et al.* [87] proposed a DeepFool method that generates minimal perturbation to change the classification labels based on iterative linearization of the classifiers. Recently, Abusnaina *et al.* [3] proposed

adversarial attacks over the CFGs of malware binaries through designing two adversarial attacks to craft the IoT detector.

4.2 Background and Motivation

Adversarial examples (AEs) can be generated by slightly manipulating a sample to fool the classifier, and done in the context of malware on either the binary or the code level. ❶ **Binary-level AEs** the generation of such AEs entails manipulating the bytes of the malware sample upon compilation, without any regard to the function and purpose of such bytes, as has been done in several works [10, 129, 64]. Another method for binary-level AEs generation would entail injecting a benign block of bytes into an unreachable part of the malware binary, e.g., by adding a new section or appending the benign bytes to the end of malicious code, thus altering the feature representation introduced by the AE. ❷ **Code-level AEs** the generation of those AEs entails applying perturbation over the original code by either modifying the structure of the code or inserting an external code into it. For instance, augmenting or splitting functions results in a structure modification, thus altering the resulting feature space representation of the sample (e.g., CFG-based).

4.2.1 Practical Adversarial Examples

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

Impractical Adversarial Example. An AE is impractical if the injected code is compiled as an unused function during the compilation process. In the binary level, a sample that manipulated

Table 4.1: Current adversarial attacks defenses. The focus of the adversarial attacks defenses is on AEs in the context of image classification. Note: MLP is Multilayer Perceptron, DNN is Deep Neural Network, and RNN is Recurrent Neural Network.

Paper	Model	Dataset	Application
Goodfellow <i>et al.</i> [43]	MLP	130,000+	Image
Xu <i>et al.</i> [128]	DNN	6,000	Image
Meng <i>et al.</i> [78]	DNN	121,000	Image
Liao <i>et al.</i> [73]	DNN	280,000	Image
Dhillon <i>et al.</i> [35]	DNN	60,000	Image
Papernot <i>et al.</i> [95]	DNN	130,000	Image
Samangouei <i>et al.</i> [101]	DNN	70,000	Image
Miyato <i>et al.</i> [82]	RNN	805,753	Text

by any form of byte injection (*e.g.*, adding a new section or appending at the end of file) is not considered as the practical adversarial example.

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

Both code- and binary-level approaches can be used for generating practical AEs, although binary-level approaches are difficult to apply for fine-grained perturbations. A recent study by Abusnaina *et al.* [3] showed that adding external code to the original one leads to a high misclassification rate of the model’s outputs while preserving the functionality of the original code. Such behavior can be critical as it results in changing the source code, execution flow, signature, and binaries, which reduces the performance of state-of-the-art classifiers. In this study, we focus on the injection of external code as a capability for creating AEs, since such an approach affects various representations of the original samples (see subsection 4.2.4).

4.2.2 Limitation of Adversarial Learning

Adversarial training is a defense to increase the learning model's robustness by training over clean and adversarial datasets. For example, this technique is used for enhancing the robustness of image classifiers by perturbing the training data, as listed in table 4.1.

Drawbacks. A large number of studies on adversarial learning were implemented to generate AEs by perturbing the feature space, typically an image. Training a model over AEs generated by one method may not increase its robustness against other methods. This highlights the problem of adversarial learning, training against a set of methods does not guarantee the robustness against different attacks. This problem becomes critical with the existence of code-level manipulation. Where an adversary can change the outcome of the attack by changing the embedded code, or slightly changing the attack method *et al.* [3]; e.g., an adversary can decide which portion of the code he wants to execute. Therefore, adversarial benign and malicious samples can co-exist within the same feature space, calling for methods to detect AEs, by only relying on the structure of the clean sample.

4.2.3 Graph Embedding and Augmentation

Graph Embedding and Augmentation (GEA) is an approach [3] that is shown to produce AEs that are executable, while allowing targeted adversarial attacks, addressing limitations of the literature. GEA inserts benign code into a target malware sample to generate an AE, with different feature representations. GEA applies direct modifications to the CFG of the sample, affecting the extracted features and resulting in a misclassification, both targeted and non-targeted.

Generating AEs with GEA. The process of generating adversarial examples is done by merging the code of the original sample with the code of a targeted sample. The targeted sample belongs to the class, which the adversary desires to misclassify to. The generated CFGs of each sample are

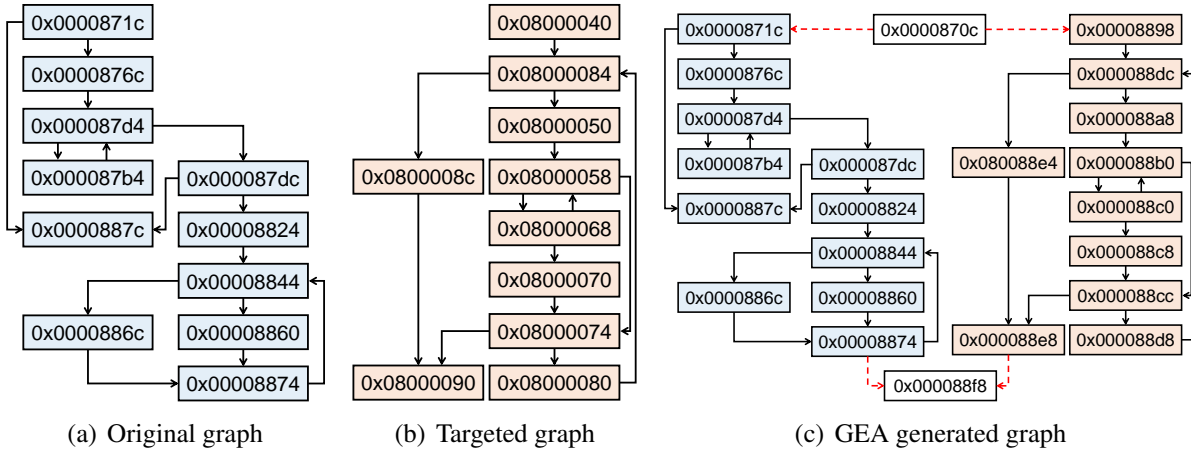


Figure 4.1: GEA adversarial approach. The CFG in figure 4.1(c) is obtained by embedding the CFG of a selected sample in figure 4.1(b) into the CFG of the original sample in figure 4.1(a). It can be done by injecting the code directly.

shown in figure 4.1. The generated CFG of the original sample code is depicted in figure 4.1(a), whereas, the CFG of the embedded code is represented in figure 4.1(b). Combining both the original and external code results in figure 4.1(c). The combination is done by creating a shared entry and exit blocks, where only one branch is executed. In this case, the left branch that belongs to the original sample will be executed. Note that changing in the structure of the code will change the extracted features, resulting in misclassification.

4.2.4 Threat Model

The adversary’s goal is to fool the classifier by misclassifying malicious samples as benign and vice versa while preserving the functionality and practicality of the original sample. Our threat model focuses on AEs generated based on code-level manipulation using GEA (see section 4.2.3).

We assume that the adversary has full access to the source code of benign and malicious samples. Moreover, the adversary can edit, compile, and merge samples, and knows the model’s design and its internal architecture. The adversary’s goal is to conduct targeted and non-targeted misclassifi-

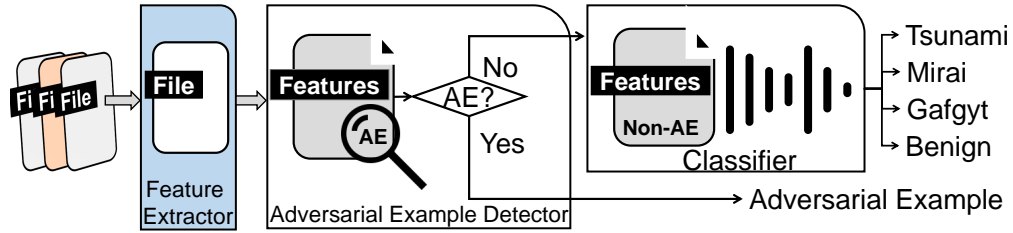


Figure 4.2: The architecture of Soteria. IoT samples are fed to the feature extraction process, where each sample is represented by multiple feature vectors. The feature vectors are forwarded to adversarial example detector. All non-AEs are then forwarded to the classifier to be classified into its corresponding family.

cation. The objective of Soteria is to provide a robust and accurate classification in the presence of this model.

4.3 System Design

4.3.1 High-Level Architecture

To address the impracticability of modification-based adversarial examples, we propose Soteria, a malware classification framework that incorporates two modules: adversarial sample detection and malware classification. Soteria manifests the following advantage. It eliminates the cost of extracting new features, meaning that it can re-use the features generated during the detection of the AEs to classify a sample as benign or malicious. Alternatively, the user has the flexibility over the choice of classifier, meaning that the user can make use of different set of features, classifier parameters, or another classifier altogether. The high-level architecture of Soteria, comprising of three major components, feature extractor, AEs detector, and malware classifier is shown in figure 4.2.

Feature Extractor. Soteria utilizes the features from the graphical representation of a program’s flow execution, i.e., CFG. For a graph G , such that, $G = (V, E)$, and nodes (V) and edges (E)

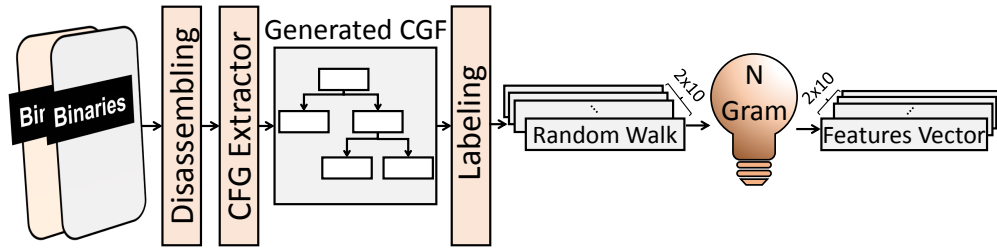


Figure 4.3: Soteria feature extraction process. IoT samples binaries are disassembled to extract their corresponding CFGs. Then, two nodes labeling techniques are used (Dense-based and level-based), then, several random walks are done over each labeled graph. The trace of the random walk is then used for feature extraction by using n -grams with TF-IDF.

represent the basic blocks and the traversed paths, respectively. A critical advantage of the CFG is that it summarizes the control flow by connecting the entry block with reachable blocks directly or indirectly. Particularly, if a block of code is appended to an existing program, with an intention to fool the classifiers, knowing that the appended blocks are unreachable, our feature extraction methodology ignores such blocks, in contrast to binary- and image-base classifiers. The features driven from the CFG ignore the non-executable part of samples, eliminating the effect from noise injection and unused functions in the sample.

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

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

4.3.2 Adversarial Examples Detector

The purpose of the detector is to distinguish normal samples from adversarial ones, regardless of whether the sample is malicious or not. The flow of the feature extraction, including sample pre-processing with CFG extraction and labeling, followed by feature extraction using n -gram of the obtained random walks on the labeled CFG is shown in figure 4.3.

4.3.2.1 Sample Pre-processing

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

❶ **Density-based Labeling (DBL).** The density of a node is defined as the summation of in- and out-edges over the total number of edges in the graph. DBL sorts all nodes according to their density, where the densest node is labeled as 0 and the least dense node is labeled as $|V| - 1$, and the centrality factor of a node is used to rank nodes with tied density CF_{v_i} ². If two or more nodes still have the same centrality factor, we assign labels based on their levels, considering the main or entry block function as the entry node. We notice some cases where two nodes with equal values are at the same level (symmetric nodes), and label them in ascending order since switching their labels will not affect the consistency of labeling.

Nodes 0 and 1 are the densest nodes because they are connected to four blocks, and node 0 has a higher centrality factor value. The labeling ends by assigning label 10 to the entry block as the least dense node with the lowest centrality factor, as shown in the result of the density-based labeling in figure 4.4(a).

²Centrality factor of a node is the sum of node's betweenness and closeness centrality values, $CF_{v_i} = B_{v_i} + C_{v_i}$. The betweenness centrality (B_{v_i}) of a node v_i is defined as $\Delta(v_i)/\Delta(m)$, where $\Delta(v_i)$ is the count of shortest paths travel through v_i and connecting nodes v_j and v_t , for all j and t where $i \neq j \neq t$, and $\Delta(m)$ is the total number of shortest paths between such nodes. The closeness centrality (C_{v_i}) of a node is defined as the average shortest path between node v_i and all other nodes in the graph G .

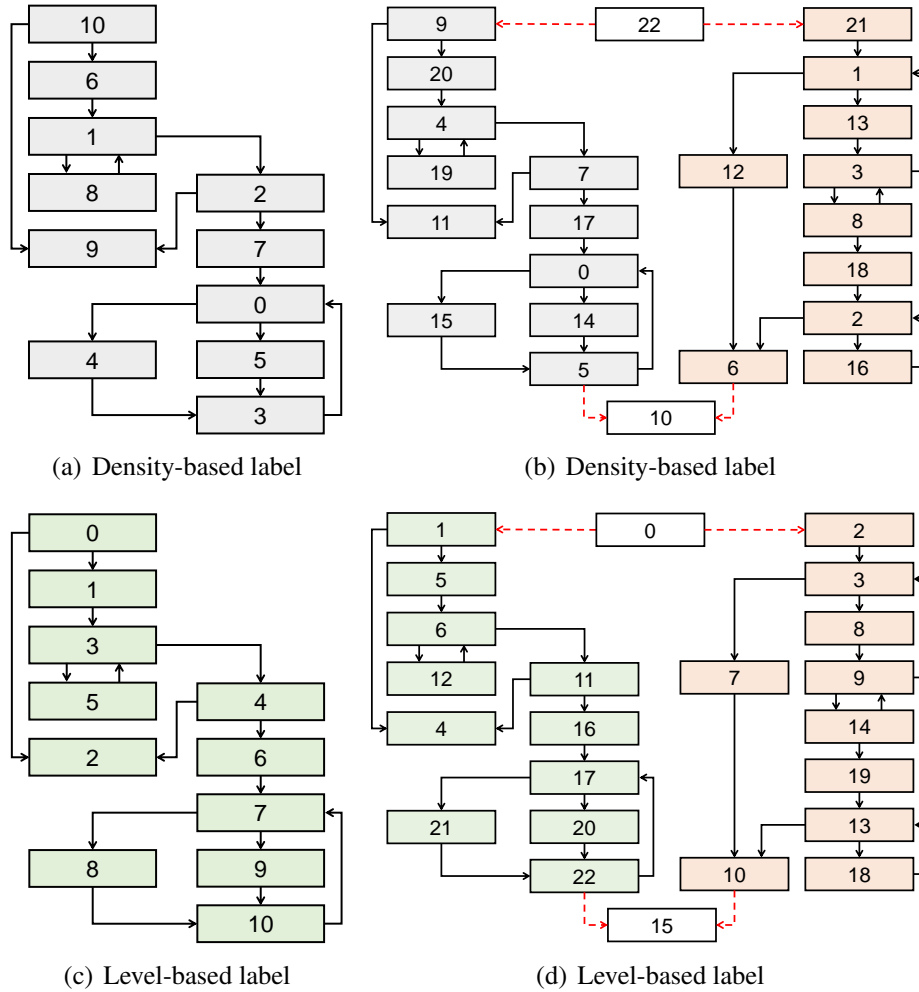


Figure 4.4: Graph labeling using two approaches, density- and level-based. Each node has a label in $[0, |V| - 1]$, where $|V|$ is the number of nodes in G . Figures 4.4(b) and 4.4(d) show the labeling of the GEA generated CFG over the original graphs in figures 4.4(a) and 4.4(c), respectively.

II Level-based Labeling (LBL). The level of a node v_i is defined by the smallest number of steps S_{v_i} from the entry node to reach v_i , where the level of a node is equal to $1 + S_{v_i}$. In LBL, we consider the main or entry block function in the CFG as the first level layer, and follow (in breadth-first search manner) other levels for labeling them. For nodes at the same level, we follow the same labeling mechanism in DBL.

The entry block is assigned with label 0. In the second level, there are two nodes with the same density values, and the centrality factor values are used. The process ends by labeling the last level nodes. Note that the entry block will always have the label 0 when using the LBL method, as shown as an example for the LBL labeling result in figure 4.4(c),

Both density- and level-based labeling follow the strict predefined rules to guarantee consistency of representation and ensures that any modification applied to the graph will be reflected in the labels' assignment. It is worth noting that the labels' assignment varies for each graph, even when they share a sub-graph. Labels' assignment over GEA results in changing the labels, and the feature extraction process, hence affecting the detector's behavior. The labeling result of the generated graphs using GEA is shown in figure 4.4(b) and figure 4.4(d).

4.3.2.2 Feature Representation

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

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

✦ **n -grams:** The n -gram technique can be used in different models for feature representation of text, documents, graphs, etc. Unique terms or n -gram are extracted from the entire corpus before counting the frequencies in individual samples. Inspired by node2vec [47], we use n -gram representation of the graphs from the sequences of nodes obtained by the random walk. From the derived random walks with the lengths specified above, we extract n -grams of lengths 2, 3, and 4 as a feature representation of the CFG. Given that, the number of n -grams is large even for small graphs. We select and use the top 500 discriminative features for each LBL and DBL (thus, 1,000 features in total). The selection of the top discriminative feature is based on the frequency of W .

4.3.2.3 Building Detection Model

The auto-encoder reconstructs the given input at the output layer, and consists of four main blocks; an input layer, an output layer, hidden layers, and a validation unit, which are described below. The core of the detection model is an auto-encoder that consists of five fully connected dense layers, as shown in figure 4.5.

- **Input Layer.** This layer is a one-dimensional vector of size 1×1000 fed by the density- and level-based features vectors.
- **Hidden Layers.** These layers consist of three fully connected dense layers, and extract a deep representation of the features. The design is based on decoding the features from 1×1000 to 1×2000 and 1×3000 . Afterward, a third layer encodes the features presentation to 1×2000 . This structure eliminates the features dependencies in the reconstruction process, as the extracted features are mutually independent.
- **Output Layer.** This layer is fully connected to the third hidden layer. With a shape of 1×1000 , the output layer reconstructs the features seen at the input as its output, which is then returned as a density- and level-based vector.

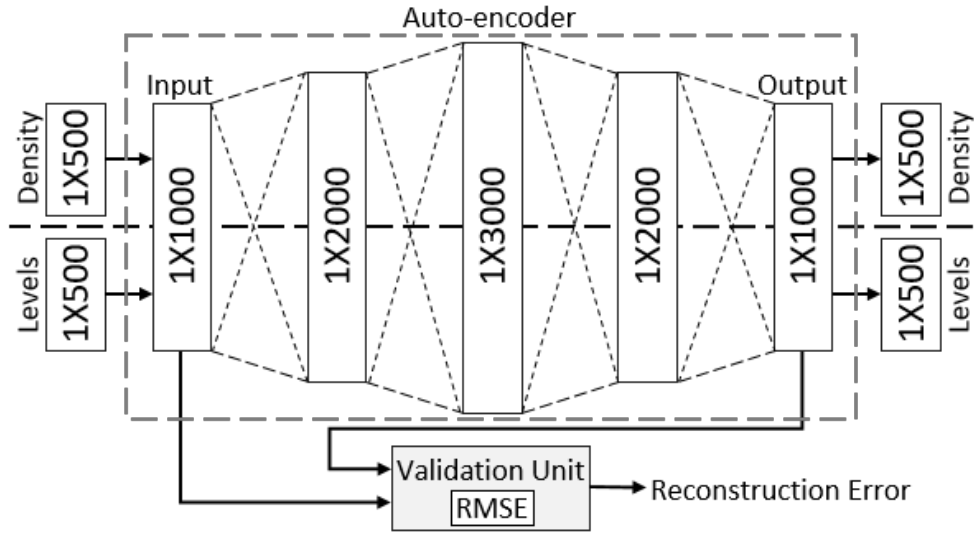


Figure 4.5: The proposed AEs detector: The detector consists of five fully connected layers auto-encoder. The input to the auto-encoder is density- and level-based feature vectors, where the output is the reconstructed feature vectors. A validation unit is used to calculate the reconstruction error. A sample is considered as AE if reconstruction error exceeds a threshold.

- **Validation Unit.** The validation unit computes the Reconstruction Error (RE) by calculating the Root Mean Square Error (RMSE) between the original input x and the reconstructed output \hat{x} . If the RMSE exceeds the threshold, set to be 50%, the sample x is labeled as AE.

4.3.3 Classifier

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

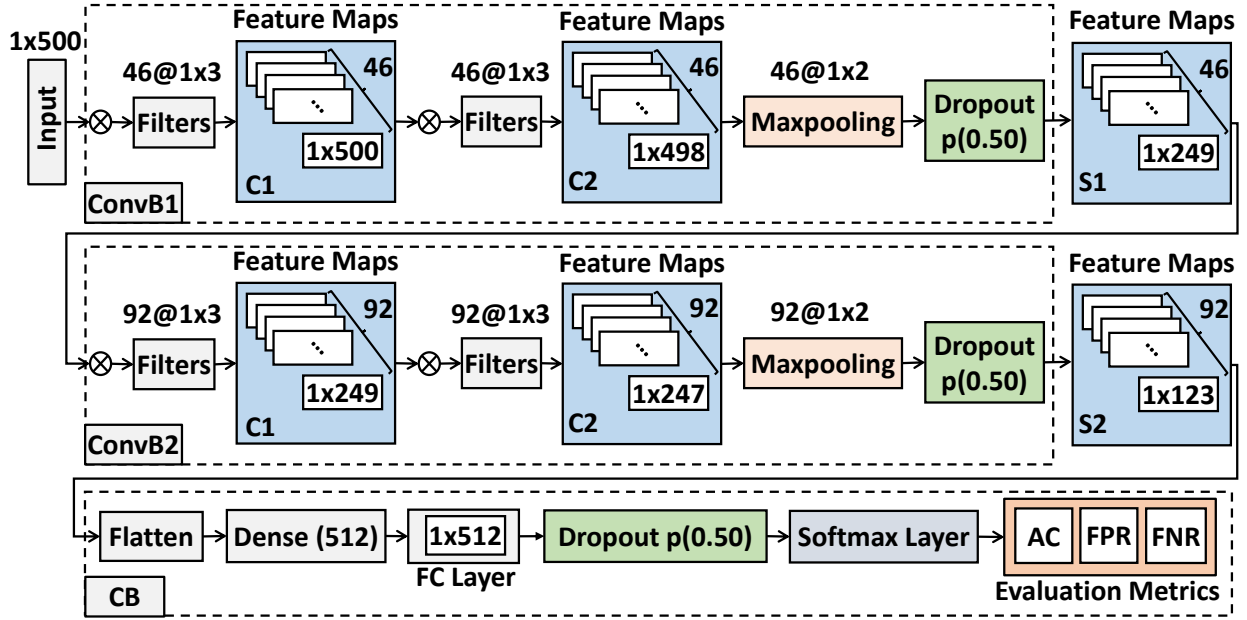


Figure 4.6: The structure of Soteria classifiers. The classifiers consist of four convolutional layers with max-pooling and dropout functions. The output of the classifier is the softmax probability of each class.

4.3.3.1 CNN Classifiers

The input to the classifier in Soteria is a one dimensional (1D) vector of size 1×500 representing the density- or level-based extracted features. All layers use the Rectified Linear Units (ReLU) activation function, and dropout regularization to prevent model over-fitting. Below, we describe the CNN structure in the following using the notation p as the dropout probability, s as the stride, m as the max-pooling size. The structure of the classifier, which consists of three blocks: convolutional blocks (ConvB) 1 and 2 and a classification block (CB), is shown in figure 4.6.

- **ConvB1.** ConvB1's input is the extracted features, and consists of two consecutive convolutional layers with 46 filters of size 1×3 , that operate convolutions with $s = 1$ with no padding to generate feature maps of size 46×498 . Each convolutional layer is followed by a max-pooling with $s = m = 2$ and a dropout with $p = 0.25$.

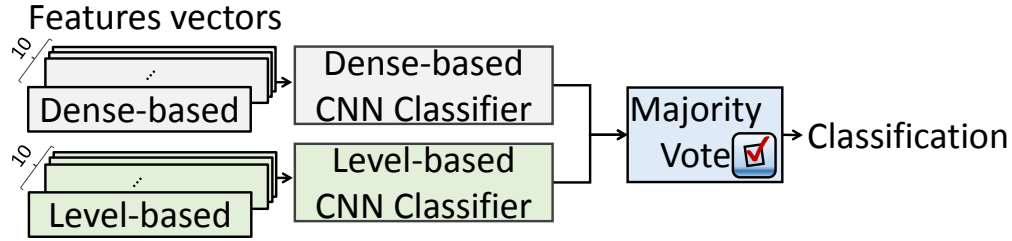


Figure 4.7: Soteria classification process. The CNN-based models’ inputs are the dense- and level-based feature vectors. The classification decision is the majority vote of the CNN classifiers output probabilities over the feature vectors.

- **ConvB2.** Similar to ConvB1, except for the number of filters. ConvB2 consists of two convolutional layers with 92 filters of size 1×3 , followed by max-pooling and dropout.
- **CB.** CB’s input is the flattened feature maps of ConvB2, fed to a fully connected layer of size 512 with a dropout $p = 0.5$. The output of the fully connected layer is fed to a softmax layer for the classification.

4.3.3.2 Majority Voting

For each sample, we perform ten random walks and generate 20 feature vectors (from both DBL and LBL). These feature vectors are forwarded to their corresponding CNN classifiers. The final output is based on the majority voting unit, where the class with the highest vote is used as the sample’s label, as shown in figure 4.7.

4.4 Dataset and Evaluation

4.4.1 Dataset

To evaluate Soteria, we assembled a dataset of IoT benign samples and IoT malware. We collected 13,798 malware samples, randomly selected from CyberIOCs [30] during the period of January

Table 4.2: Distribution of IoT samples across benign and malicious families. Gafgyt is the most popular IoT family with 66.18% of the dataset samples, while Tsunami is the least popular with only 262 samples (1.55% of the samples). The dataset is split into the train (80%) and test (20%) subsets.

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

2018 to late February of 2019. For the benign samples, we manually assembled a dataset of 3,016 samples from source-code projects available on GitHub [31]. Next, we used Radare2 [32] to obtain the CFGs of the samples. Throughout the study, wherever required, we use 80% of our dataset for training and validation and 20% for evaluation.

Malware Family (Class). To determine the family label of the malware, we inspect the malware samples through *VirusTotal* [33]. The scan results from the *VirusTotal* are then passed through *AVClass* [104] to label them with their family class. *VirusTotal* scans include scan results from multiple anti-virus software, each of which assigns a family name to the malware. *AVClass* further uses the majority vote to determine the family label. Soteria classifies the samples into different classes, i.e., family labels and benign. We divided our dataset into 80% for training and validation, and 20% for evaluation. The distribution of IoT samples across classes is shown in table 4.2.

Adversarial Dataset. Recall that we utilize the GEA to generate the AEs to evaluate Soteria’s robustness. These AEs are generated from the test dataset (20% of the samples per class). Towards this, we start by selecting three samples from each class, i.e., one from each size: small, medium, and large. We define small, medium, and large by the minimum, median, and maximum number of nodes in the dataset. Taking a sample from a class for each size, as the targeted sample, we

Table 4.3: GEA selected targeted samples. These samples are used to generate the AEs to evaluate Soteria. Three samples from each class are selected of different sizes (number of nodes), *i.e.* small, medium, and large.

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

generated adversarial examples by applying GEA over every sample in the test dataset of all the classes except for the targeted sample class. For example, if we select a sample of size *Small* from the benign dataset, we then apply GEA over this sample and each of the samples in the test dataset of Gafgyt, Mirai, and Tsunami, giving us a total of 2,742 AEs (Gafgyt, Mirai, and Tsunami have 2,217, 473, and 52 samples, respectively, aggregating to 2,742 AEs). The number of generated AEs of each class is reported in table 4.3.

4.4.2 Feature Analysis

We extract features from 200 random samples from each class. Recall that we extract density- and level-based features from each sample. We use both of these feature vectors together to create a combined feature vector of size 1×1000 . We used Principal Component Analysis (PCA) [81] with a dimension of two. PCA converts a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components.

Table 4.4: Distribution of dense- and level-based feature vectors extracted by n -grams technique from the random walk traces among the IoT benign and malware classes.

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

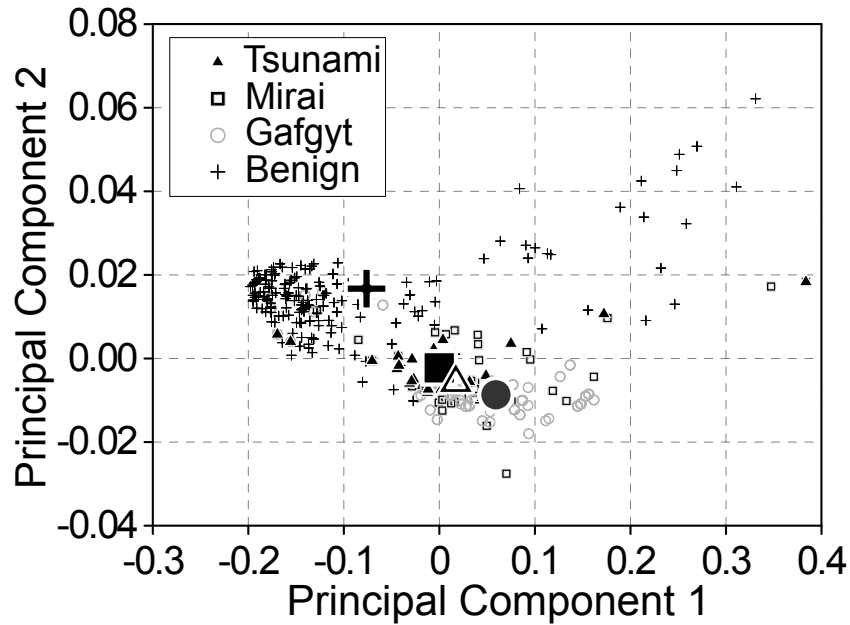
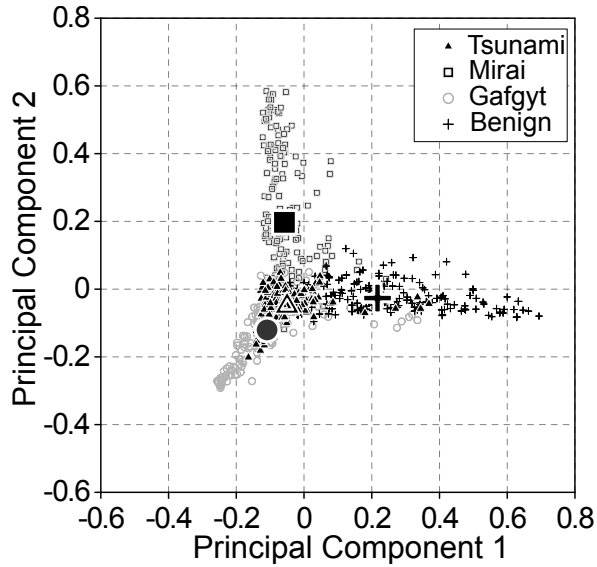
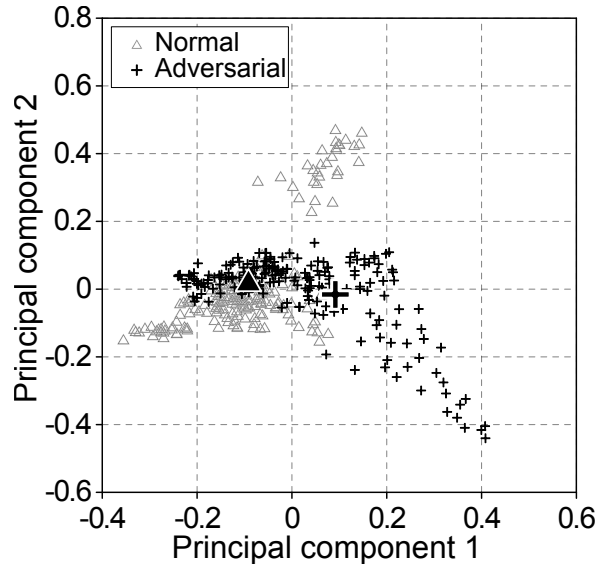


Figure 4.8: The PCA comparison between the benign and malware families using features used in Alasmery *et al.* [8].

Baseline Comparison. Prior works, like, Alasmery *et al.* [8] and Abusnaina *et al.* [3] use graph-theoretic features extracted from the general structure of the CFG. With the comparative analysis of such features with our feature considerations, we exhibit our feature sets to be more discriminative. We notice that our feature representation is more discriminative of the classes. Additionally,



(a) Benign and malware



(b) Normal and adversarial

Figure 4.9: Soteria: Dense-based labeling feature vector comparison. Figure 4.9(a) shows the PCA distribution of benign and malware samples. Figure 4.9(b) shows the PCA distribution comparison between the normal and GEA generated AEs.

we notice that the malicious classes in the figures are indistinguishable using the graph theoretic features. The PCA visualizations of the feature vectors between the classes of features considered in the prior works and our features design are shown in figure 4.8, figure 4.9(a), figure 4.10(a), and figure 4.11(a), respectively.

Moreover, the distribution of the discriminative features over the four classes with 51 and 129 density-based and level-based features, respectively, shared between classes is reported in table 4.4.

AE vs. Clean Features. To detect AEs, i.e., distinguish the AEs from the clean samples, understanding the differences in feature representation between clean and AEs is important. To examine this, we applied PCA on the clean and adversarial feature vectors. We notice that the clean and AEs are distinguishable, particularly when using the combined feature vectors. The results of which are shown in figure 4.9(b), figure 4.10(b), and figure 4.11(b).

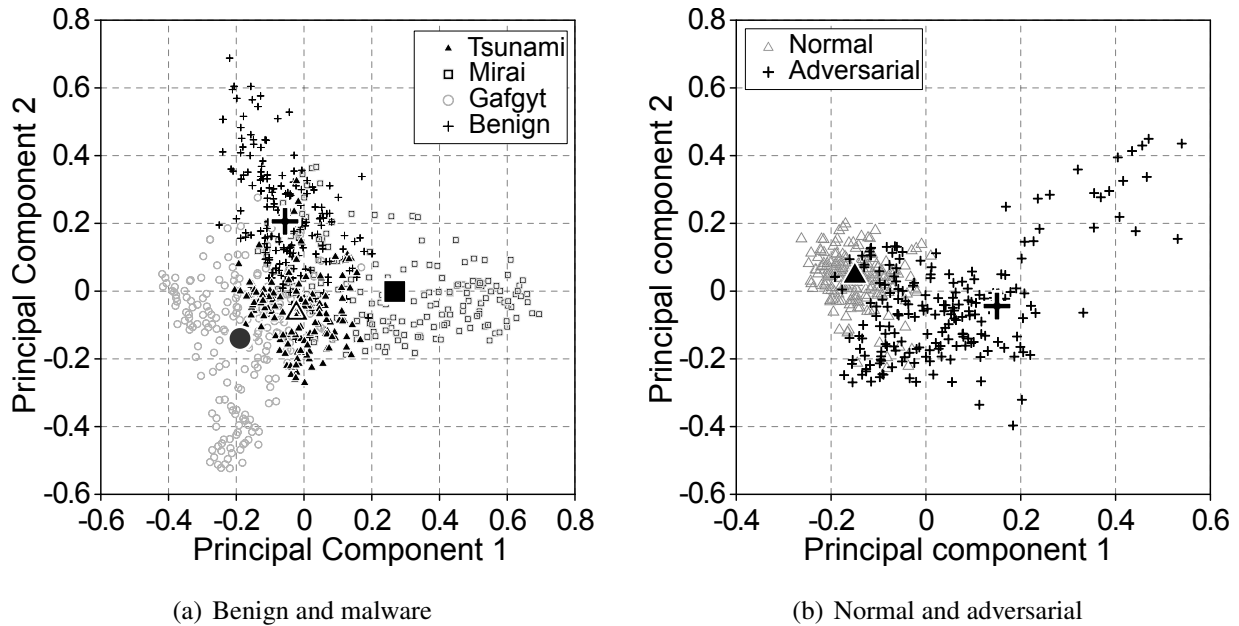


Figure 4.10: Soteria: Level-base labeling feature vector comparison. Figure 4.10(a) shows the PCA distribution of benign and malware samples. Figure 4.10(b) shows the PCA distribution comparison between the normal and GEA generated AEs.

4.4.3 Evaluation and Analysis

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

4.4.3.1 Adversarial Example Detector

We evaluated AE detector of Soteria by its ability to detect AEs and distinguish them from the clean samples, regardless of their class. The spatial differences between clean and adversarial samples are shown in figure 4.9(b), figure 4.10(b), and figure 4.11(b).

Training Parameters. The reconstruction error (RE) is the RMSE between the original and reconstructed samples; we set the number of epochs to 100 with a batch size of 128. We trained

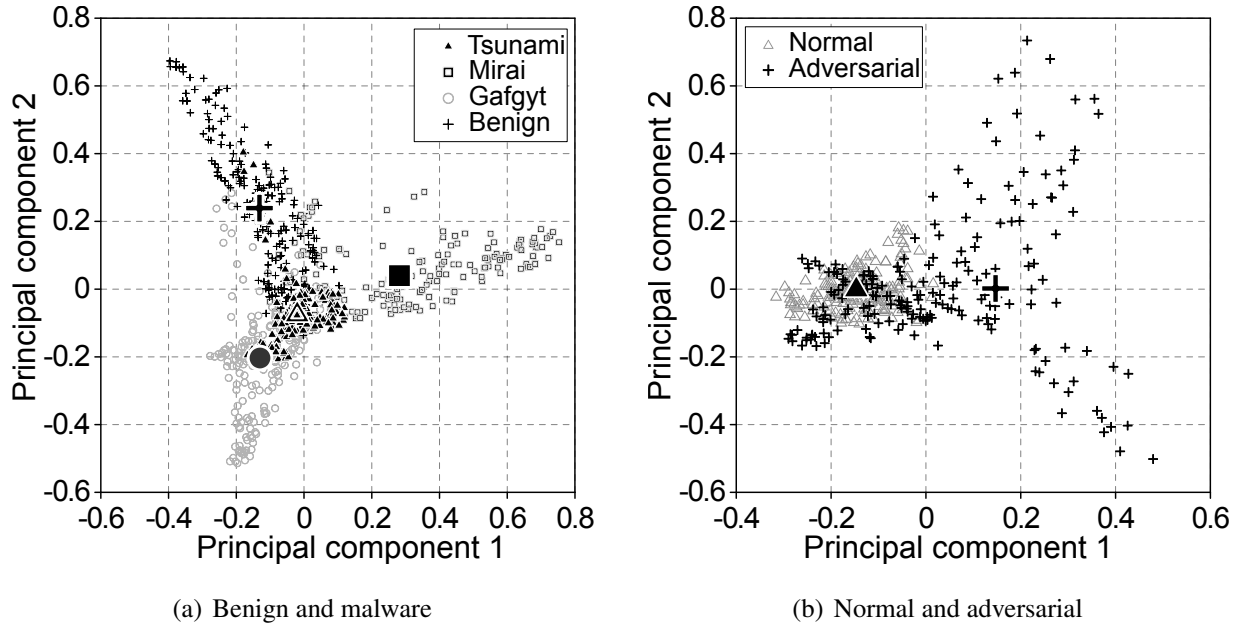


Figure 4.11: Soteria: Combined labeling feature vector comparison. Figure 4.11(a) shows the PCA distribution of benign and malware samples. Figure 4.11(b) shows the PCA distribution comparison between the normal and GEA generated AEs.

Soteria on reconstructing the training data in table 4.2.

Testing. Given the trade-off between adversarial detection sensitivity (false negatives) and the clean samples misdetection (false positive), setting a proper RE threshold is essential. We calculate the RE and set the threshold (T_h) as $T_h = \mu(\vec{R}_E) + \sigma(\vec{R}_E)$, where \vec{R}_E is a vector of all RE values of the training samples, and μ and σ are the mean and standard deviation of the training samples RE, respectively. To consider a sample as adversarial, half of its feature vectors should have a RE higher than the threshold. The RE distribution over the clean and adversarial features vectors is depicted in figure 4.12.

Performance. Overall, Soteria detector detects 97.79% of the AEs. In most cases (9 out of 12), the detector was able to detect AEs with an accuracy greater than 99%. The performance results of Soteria against AEs are reported in table 4.5.

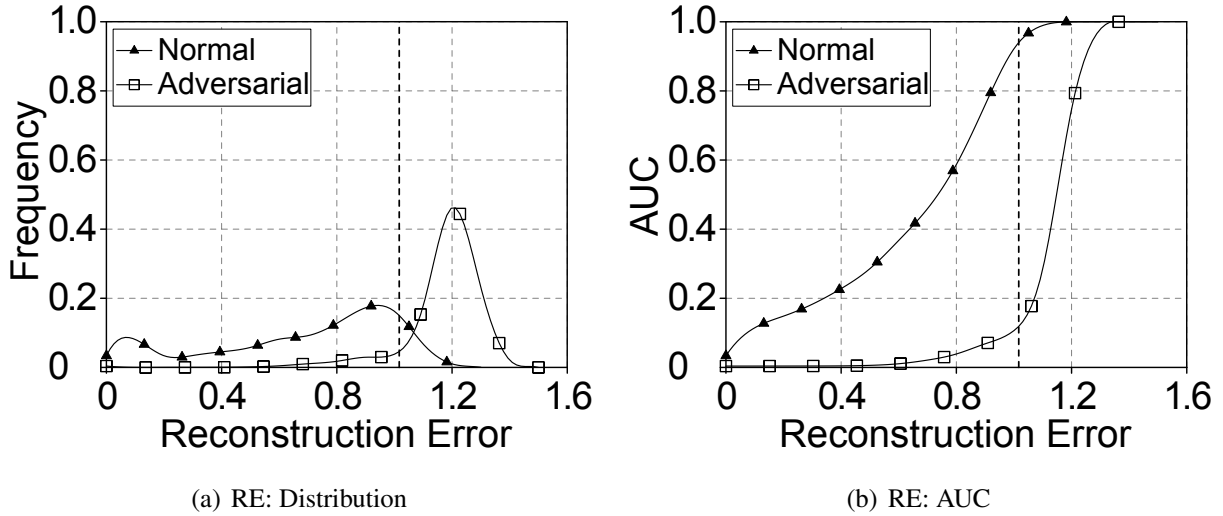


Figure 4.12: Reconstruction Error (RE) comparison between normal and the generated AEs. Figure 4.12(a) shows the distribution frequency of the RE among the normal and adversarial samples. Figure 4.12(b) represents the accumulated frequencies of samples and their corresponding RE. The vertical dashed line is the chosen threshold for Soteria AEs detector.

Furthermore, We notice that only samples from the Gafgyt family were misdetrcted as AEs, mainly because of the high number of discriminative features associated with this family. The detection performance against clean samples is reported in table 4.6.

In conclusion, we detected AEs and distinguish them from the clean samples with high accuracy. The detected samples are labeled as adversarial and will not be forwarded to the classifier.

Analysis. To show the importance of setting the right threshold, we re-implement the threshold as $T_h = Mean(\vec{R}_E) + \alpha \times SDV(\vec{R}_E)$, where α is an arbitrary value. We test the detector performance against the clean and adversarial samples by varying α from 0 to 2.0. With $\alpha = 0$, all AEs were detected, although more than 60% of the clean samples were classified as AEs. With $\alpha = 2.0$, all clean samples were correctly detected, and no AEs were detected by Soteria. Note that our selected threshold was chosen without access to the test dataset. The effect of α on the detection error is depicted in figure 4.13.

Table 4.5: GEA: Detector Performance over adversarial samples. The detector was able to detect an overall percentage of 97.79% of the AEs. DE refers to the detected samples.

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

Table 4.6: GEA: Detector Performance over clean samples. Only 6.16% of the clean samples were misclassified as AEs. All Benign clean samples passed the detector. DE refers to the detected samples (lower is better).

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

4.4.3.2 Classifier

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

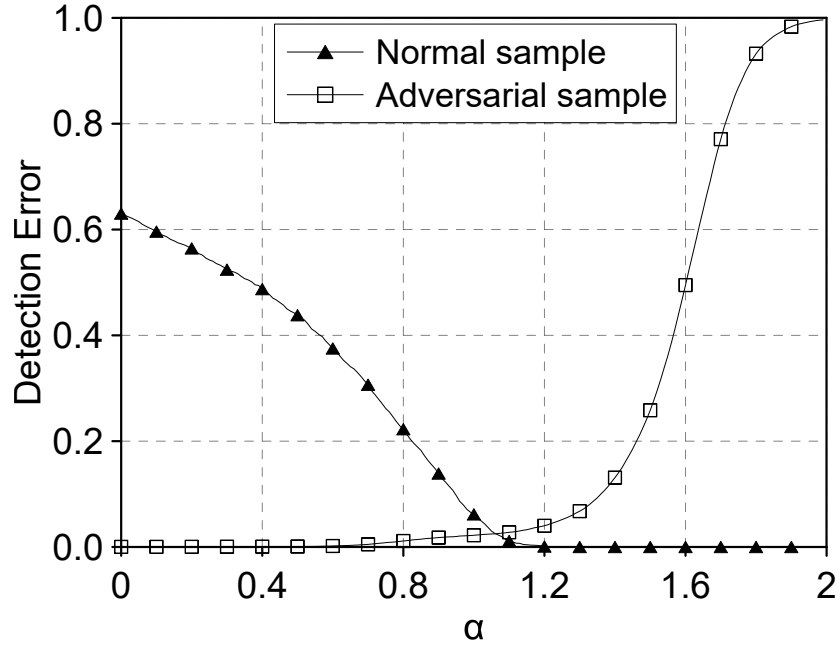


Figure 4.13: Effect of varying the detector threshold (α) on the detection error. The selected α in Soteria is the intersection between the error rates of normal and adversarial samples.

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

Performance. We evaluated Soteria’s classifier’s performance against two existing models: ① Graph-based: Alasmay *et al.* [8] propose a malware detector based on features extracted from the general structure of the CFG. , and ② Image-based: Cui *et al.* [28] uses an image-based design where each sample is represented as an image of a fixed size to detect malware. We implemented the above two systems. The model accuracy over each class is measured by the number of samples correctly classified over the total number of samples that belong to that class. For the image-based model, we implemented the four models mentioned in their design. The evaluation of 96×96 and 192×192 based models shows poor performance, with an overall accuracy rate of 66.37%. Therefore, we did not include it in the comparison. Our evaluation shows that Soteria outperforms

Table 4.7: Classification performance of Soteria dense-, level-, and voting-based classification systems in classifying normal (non-adversarial) samples.

Class	Model Accuracy					
	DBL	Soteria		[3]	[28]	
		LBL	Voting		24 × 24	48 × 48
Benign	99.45	99.70	100	99.00	99.00	99.50
Gafgyt	99.70	97.00	100	98.55	98.87	99.14
Mirai	99.49	98.73	99.36	97.67	92.81	92.81
Tsunami	100	100	100	84.61	32.69	59.61
Overall	99.63	97.77	99.91	98.29	97.01	97.70

Table 4.8: Soteria’s classifier predictions over AEs misdetected by the detector. Most of the mis-detected samples are generated using GEA with large size selected samples.

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

the existing systems, as particularly shown in the Tsunami classification and overall accuracy rate.

The performance results of the models for the different classes are reported in table 4.7.

Analysis. Recall that the accuracy of our AE detector was 97.79%, meaning that 2.21% of the AEs were not detected by Soteria, and were forwarded to the classifier. Given its application, it

is important to understand the classifier’s behavior against those samples. The classifier detects them as benign or Gafgyt, with a large percentage (76.1%) of the samples classified as benign. It is worth noting that there is a clear difference in the feature representation between clean and AEs, as shown in figure 4.9(b), figure 4.10(b), and figure 4.11(b) .

However, due to the variety in the benign samples’ features distribution, the adversarial examples that pass the detector are likely to be classified as benign. This can be critical, even with a detection rate of as high as 97.79%, given the application domain. The classifier’s behavior over these samples is shown in table 4.8.

4.5 Discussion

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

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

Adversarial Capabilities. In section 4.2.4, we discuss the threat model and adversarial capabilities. We assumed that the adversary can access the source code of the samples, and can modify

and merge them. Moreover, he has prior knowledge of the design and its internal architecture. Soteria's success implies that the adversary cannot generate practical AEs. What the adversary does not have in Soteria is the ability to know in advance what features are being used for the classifier, since those features are randomized for every run of the system. For instance, inserting a single block with a low density near the exit block will not highly affect the labeling of the sample, and will not be detected as an AE by Soteria. However, Soteria can classify the sample to its original class since the labels are intact. Moreover, the adversary needs to ensure that the labels change in such a way the classification decision will be toggled without being detected by the AE detector, which happened in our evaluation in 2.21% of the generated AEs. Finally, and due to the change in the labeling, the adversary cannot force the classifier into a targeted misclassification.

Alternative Features for Classifier. In Soteria, we built a classifier that is based on the utilized features from the detector design process. However, the classifier can be replaced, with some caveats. The detector decision is based on the extracted CFG. Appended binaries at the end of the file will not affect the detector decision. Clean samples with adversarial binaries appended to them will not be detected as AEs by Soteria. While this is an advantage of Soteria classifier, it is equally a serious shortcoming with other approaches, such as image-based malware classifiers [75]. Ideally, the classifier should be at least as good as the classifier proposed in this chapter, meaning that it should only consider the executable binaries in the classification process. Moreover, the discriminative features are highly distinguishable among classes, and the feature extraction process is immune to feature space manipulation.

Limitations. Our work has two major limitations. ❶ **CFG-based Features:** CFG-based features are effective compared to other feature designs. However, CFG does not necessarily reflect the actual code. Editing the code without even changing the functionality (by creating an equivalence) would affect the structure of the CFG, which might be exploited by the adversary to evade detection in the first place. For example, an adversary may inject a sample of code that would not result

in a new branching, but would still affect the structure of the CFG. While such an event is well within the scope of our adversary model, and would not affect the classification results, it would only affect the feature space, requiring us to retrain Soteria to capture the new feature space. ②

Binary Obfuscation: Obtaining a representative CFG would not be possible under obfuscation, typically done using string obfuscation, resulting in hiding parts of the code, or function obfuscation, resulting in an incomplete CFG. An incomplete CFG may result in an incomplete feature representation of the sample, and thus a misclassification. Obfuscation is a shortcoming of our work, and deobfuscation is an active research area in its own right, where developed tools can be used as the basis for our work to obtain representative CFGs.

4.6 Conclusion

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

CHAPTER 5: SHELLCORE: DEFEATING IOT MALWARE THROUGH IN-DEPTH ANALYSIS AND DETECTION OF IOT SHELL CODE

Malware-infected hosts use Command and Control (C2) servers to obtain payloads that include instructions to compromised machines (or bots) synchronizing their actions, including their cycles of activity by attacking targets, propagation by recruiting new bots and acting as a source of propagation, and by stealthy operation to evade detection. To achieve those goals, the Linux shell comes to their rescue. For example, to change privileges, it uses the shell to execute *chmod* command, to infect other hosts it uses the shell to launch dictionary brute-force attack, to propagate (bot) it connects to C2 servers to download instructions using the HTTP protocols via Linux shell, to launch an attack it uses the shell to flood the HTTP of the victim, to remove the traces of execution it executes *rm* command on the shell, etc. [12].

IoT devices utilize a packed version of Linux libraries, called Busybox [125], to implement Linux capabilities. Linux shell is susceptible to brute-force, privilege escalation, Shellshock, and other vulnerabilities (e.g., CVE-2018-9310, CVE-2019-1656, CVE-2018-0183, CVE-2017-6707) etc. [90, 34, 22, 119]. Having access to the shell on a device would allow the adversary a full control to the device. With the emergence of Linux-based IoT devices and the fact that the majority of work in the literature has been focused on other shell interpreters (e.g., power and web), analyzing and detecting the malicious use of the Linux shell in IoT devices is of prime importance, which we address in this chapter.

5.1 Motivation

Shell Commands and Automation. Most IoT malware uses stealthy operations to evade detection. In their operation, IoT botnets heavily rely on Unix-based shell commands. For example,

```
1 wget \%s -q -O DNS.txt
2 || busybox wget \%s -O DNS.txt
3 || /bin/busybox wget \%s -O DNS.txt
4 || /usr/busybox wget \%s -O DNS.txt
```

Figure 5.1: Retrieving a list of target hosts.

bots use the shell to execute *chmod* command to change privileges. Further, bots use the shell to launch a dictionary brute-force attack and to propagate by connecting to the C2 server to download instructions using the HTTP protocols. To launch an attack a bot typically obtains a set of targets from a dropzone by invoking a set of commands that uses the shell to flood the HTTP of the victim and to remove the traces of execution by executing the *rm* command [12], as shown in figure 5.1 .

Adversaries today leverage the results from search engines for Internet-connected devices, such as Shodan [77]. For example, a simple “default password” search on Shodan returned 72,763 results. Moreover, malware authors can arm themselves by exploiting a set of vulnerabilities, such as those present in services run on the device (*e.g.* CVE-2018-0183) and/or in outdated firmware (*e.g.* CVE-2016-1560), to gain access to the shell. On top of all, zero-day vulnerabilities can be abused to access the shell to achieve short-term attacks. Therefore, detecting malicious shell commands to harden the security of a device is of paramount importance. Prior works have studied the malicious use of Windows PowerShell. However, the malicious utilization of the Unix shell is not fully-investigated, especially when used by IoT devices. This chapter aims to study and detect the malicious shell commands used in common IoT malware.

In this chapter, we design, implement, and evaluate ShellCore, a system for detecting malicious shell commands used in IoT malware. To this end, we collect a dataset of residual shell commands used by IoT malware samples from IoTPOT [91].

Our analysis of such samples shows that shellcodes can be found embedded in the disassembled code of the malware binaries. Therefore, we utilize static analysis to search through the disas-

sembled code of malware to extract shell commands used in malware samples. For the benign shell commands, we collect a dataset from benign applications and users. In particular, we use the traffic generated from applications in a real-time environment. For analyzing and detecting malicious commands, ShellCore utilizes a Natural Language Processing (NLP) approach for feature generation, followed by a deep learning-based model to detect malicious commands.

Contributions. This chapter aims to utilize static analysis to detect the malicious use of shell commands in IoT binaries, and to use those shell commands as a modality for IoT malware detection. As such, we make the following two contributions::

- Using shell commands, extracted from 2,891 recent IoT malware samples along with commands from a benign dataset, we design an accurate detection system that detects malicious shell commands with more than 95% accuracy. Compared to the state-of-the-art approaches, our approach is more efficient and accurate. Using token- and character-level features, the feature space on the shell commands is easy to explain and interpret. Features contributing to malicious behavior can be easily identified and restricted to legitimate use.
- We extend our command-level detection approach and design a detection model for malicious files (malware samples), which often include multiple commands. Extending the results of detecting individual commands, we group the commands by file and detect the malicious files with more than 99% accuracy. Our detection approach can be applied to files compiled for any IoT hardware architecture (*e.g.* ARM, MIPS, Power PC, etc.) as long as the shell commands are extracted (which can be done statically and efficiently).

5.2 Related Work

Shell Commands. Hendler *et al.* [51] detect malicious PowerShell commands using a combi-

Work	Shell Type	Dataset	Capability	Performance	Method
[51]	PowerShell	66,388	Detection	AUC [98.5–99.0%] TPR [0.24–0.99%]	NLP Methods CNN Architecture
[97]	PowerShell/.NET	6	Analysis	—	Static Analysis Dynamic Analysis
[119]	Linux shell	13,257	Analysis	—	Diversification
[110]	Web shell	481	Analysis	—	Static Analysis Dynamic Analysis
[116]	Web shell	7,681	Detection	Precision [98.6%] Recall [98.6%], F1-score [98.6%]	CNN Architecture
[117]	Web shell	39,596	Detection	AUC [82.9-100%] TPR [74.5-100%] TNR [88.9-100%]	Malware Signature Malware Functions Longest Word (header)
[100]	PowerShell	4, 079	Detection	AC [85%]	Deep Learning
Ours	Linux shell	1,637,658	Detection	AC [92.9-99.9%] FNR [0-31.7%] FPR [0-8.5%]	NLP Methods Deep Learning Support Vector Machine

Table 5.1: Previous works for analysis and detection of the shell commands. Abbreviations: Area Under the Curve (AUC), True Positive Rate (TPR), True Negative Rate (TNR), Accuracy (AC), False-Negative Rate (FNR), False-Positive Rate (FPR), Natural Language Processing (NLP), and Convolutional Neural Networks (CNNs).

nation of Machine Learning (ML) approaches, Natural Language Processing (NLP), and Convolutional Neural Network (CNN). Pontiroli and Martinez [97] analyze PowerShell and .NET malware by analyzing the code execution. We note that both works are focused on shell commands that can only run on Microsoft Windows, for a single architecture, and it is unclear if the same insight can be applied to IoT software and command artifacts. On the other hand, Uitto *et al.* [119] proposed a diversification technique to Linux shell commands by modifying and extending commands to protect against injection attacks.

Web Shell. The web shell is a script that allows the adversary to run commands on a web server to control a targeted web server remotely as an administrator. There have been some works on detecting malicious usage of the web shell. Starov *et al.* [110] statically and dynamically analyzed a set

of web shells to uncover visible and hidden features of malicious Hypertext Preprocessor (PHP) shells. Leveraging VirusTotal, they achieved an accuracy of 90% and 88.5% for the obfuscated and de-obfuscated shells, respectively, that are detected by at least one antivirus engine. Tian *et al.* [116] proposed a system to detect malicious web shell commands using CNN and word2vec-based approaches with precision, recall, and F1-score performance around 98.6%. Additionally, Tu *et al.* [117] proposed a detection system to identify the web shells by calculating the threshold score values calculated from Malware Signature (MS), Malware Functions (MF), and the Longest Word (LW) in the header of the files by comparing the values with a database that have MS and MF values, and achieved an accuracy of 82.9%, 93.7%, and 100% for MS, MF, and LW, respectively. Moreover, Rusak *et al.* [100] proposed a deep learning approach to classify malicious PowerShell by families based on two features from the abstract syntax trees representation of the PowerShell codes, and achieved an accuracy of 85% with 3-fold cross-validation. The literature works related to the analysis and detection of the different malware types using shell commands are highlighted in table 5.1.

IoT Malware Detection. A few works have been done on IoT malware detection. Kirat *et al.* [57] proposed an elegant sequencing-based system for detecting obfuscation and evasion techniques. Pa *et al.* [91] proposed IoTPOT, a detection system that analyzed and detected Telnet-based attacks on IoT devices. The system used a sandbox that supports different malware architectures. Bertino and Islam [15] stated that IoT devices are vulnerable to botnet attacks, and proposed a behavior-based approach that combines behavioral artifacts and external threat indicators for malware detection. This approach, however, relies on external online threat intelligence feeds (e.g., VirusTotal) and cannot be generalized to other than home network environments (due to offloading).

Hossain *et al.* [53] proposed Probe-IoT, a forensic system that investigates IoT-related malicious activities. Montella *et al.* [86] proposed a cloud-based data transfer protocol for IoT devices

to secure the sensitive data transferred among different applications, although not addressing the insecurity of the IoT software itself. Angrishi [11] outlined the anatomy of IoT botnets and recommended several security measures to address them. Dahl *et al.* [29] classified a large number of malware samples by minimizing their representation (features) using the Principal Component Analysis (PCA), allowing a large number of algorithms that could otherwise be impractical to be applied to the problem domain. Pascanu *et al.* [96] detected malware using language-level instructions and a standard classifier. Cozzi *et al.* [26] analyzed a large scale Linux malware through studying and investigating the malicious behavior of malware, and discussed obfuscation techniques that malware authors use. We note that in all of those studies, IoT malware is addressed indirectly by addressing generic malware that is not particularly tailored for IoT devices.

5.3 Background

This work studies the malicious shell commands extracted by analyzing IoT malware samples. The embedded IoT devices use wrapped utilities and libraries of the Unix systems, such that it has the Unix capabilities without much overhead to the device. Thus, many malware authors design their malware to abuse Unix capabilities, *e.g.* to gain access to the devices and to further propagate and launch attacks. Gaining insights into IoT malware through shell access analysis and understanding, by distinguishing between shell commands used by malware and benign software, form the basis of this work. We utilize static analysis to extract the commands from the malware samples. This section elaborates on the use of shell commands as a weapon, and the static analysis approach in detail.

The Use of Shell as a Weapon.

Shell is a command-line interpreter that provides a command-line interface for operating systems by executing a particular command from the terminal to perform specific tasks by calling the ap-

appropriate OS command. It abstracts the details of the communication between the kernel and the operating system by managing the technical operation. However, adversaries use the shell commands to gain access to host devices to launch attacks. This can be facilitated by the use of default credentials by the owners, vulnerabilities in the services such as SSH and device firmware. The vulnerabilities in the firmware could be due to the usage of outdated firmware or due to delayed upgrading of firmware or services. For example, in 2014, Shellshock bash attacks caused a vulnerability in Apache systems through HTTP requests, and using the *wget* command to download a file from a remote host and save it to the *tmp* directory to cause infection [59]. Additionally, a recent vulnerability (CVE-2019-1656), which results from the improper input validation in Linux operating system and can be exploited by the adversaries by sending crafted commands to gain access to targeted devices, was reported [90].

By abusing the shell, adversaries can utilize the shell to brute-force the credentials of users to gain access to the device by launching a dictionary attack. Additionally, they can use the shell to connect to C2 servers to download instructions; *e.g.* infecting the device, propagating itself, or launching a series of directed flooding attacks. Moreover, malware can use bash to *find* command to look for uninfected files in the host device and use the *tmp* directory to download and run malware. The use of Unix shell by a malware to abuse the device by executing shell commands, where the use of *or* operator is to brute-force different commands, is depicted in figure 5.1.

Static analysis. To defend against malicious utilization of the Unix shell, it is important to have a baseline of the shell commands injected by the malware. Static analysis approaches employ various techniques to reveal indicators that determine whether a software is malicious or benign. One such indicator is command executions, which can be utilized to observe and analyze the shell commands used by the malware, thereby hinting at possible execution patterns of the malware. This goal can be achieved by observing the strings, functions, and disassembled code of the program etc.. While executing programs can show commands being used in a real-time, static analysis can provide the

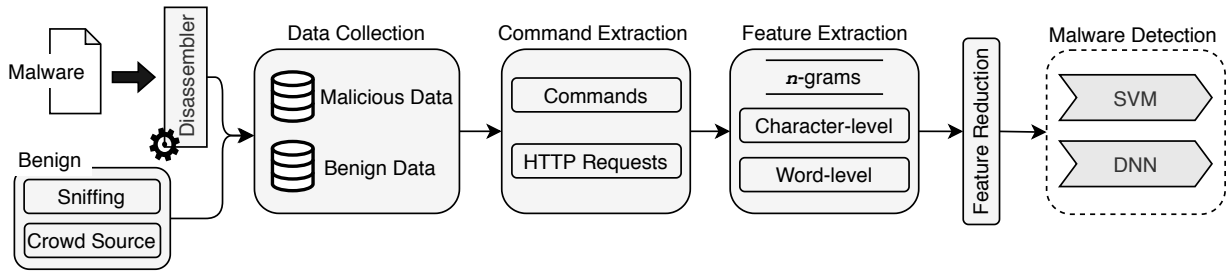


Figure 5.2: The high-level workflow of ShellCore. The malware contains HTTP requests and commands. We gather the benign dataset by crowd-sourcing and sniffing over an IoT network. Upon extracting the shell commands (*i.e.* the commands and the HTTP requests), we featurize them as n -grams. We then perform feature reduction, followed by a machine learning-based malware detection system.

commands more efficiently.

We employ static analysis on our malware dataset to create our final dataset of malicious commands. To do so, we disassemble every malware sample in our dataset and extract the strings from them. We then utilize the strings to (i) create a dataset of shell commands and label them as malicious commands and (ii) group the commands by file (malware sample) and label each as a malicious sample.

5.4 Dataset and Approach Overview

As we aim to detect malicious commands and files, we begin by disassembling the malware samples to extract strings from their code base. For our detection modules, we implement ML-based algorithms to detect malicious commands.

5.4.1 Dataset and Data Processing

We obtain a dataset of 2,891 randomly selected malware samples from the IoTPOT project [91], a honeypot emulating IoT devices. To test our detector, ShellCore, we build a dataset of be-

nign samples gathered from real-time networks and voluntary submission of shell history. Our approach, split into three modules: initial discovery, command extraction, and detection. In the initial discovery module, we disassembled the malware binaries. To create a set of rules that we can automatically apply to samples for obtaining the relevant commands, we manually examined all shell commands extracted from the strings of 18 malware samples and established patterns of those commands. Those common patterns are used to automate the extraction of shell commands from the rest of the samples.

The second component (module) in our workflow, namely the command extraction module, takes the command patterns obtained in the initial discovery phase and applies those patterns on the text obtained from each sample. More precisely, in the command extraction module, we extract the shell commands from the malicious and benign binary samples, by concentrating on the *strings*, and label them as malicious or benign. More details on the patterns are below. The system flow for ShellCore is shown in figure 5.2.

5.4.1.1 Malicious Dataset Creation

Using an off-the-shelf tool, *Radare2*, we disassemble each malware sample (among the 2,891 samples in our initial dataset) and extract the strings from the disassembled code. We then utilize the strings appearing in each sample to magnify the commands assimilated in them, then add them to our dataset of malicious commands. For coverage of the shell commands appearing in malware, we gather the *strings* from the disassembled code. For faster extraction of the shell commands, we calculate the offset where the strings reside, *i.e.*, the memory address where the string is referenced in the disassembled code, and then conduct the disassembly from that offset. We pull the instruction set at the offset and extract the desired command. Before automating the process of command extraction, we manually analyzed many samples to observe patterns that could uniquely identify the shell commands. This manual analysis included 18 malware samples

Table 5.2: Malware dataset by architecture. Percentage is out of the total samples for the given architecture.

Architecture	# of Samples	Percentage
ARM	668	23.11%
MIPS	600	20.75%
Intel 80368	449	15.53%
Power PC	270	9.34%
X86-64	242	8.37%
Renesas SH	233	8.06%
Motorola m68k	217	7.51%
SPARC	212	7.33%
Total	2,891	100%

and their associated strings. In total, we identified 1,273 patterns that we used later to extract shell commands from other samples in our dataset. Such patterns include keywords, *e.g.* kill, wait, disown, suspend, fc, history, break, etc.—we utilize online resources to build a dataset of keywords of shell commands to augment our automation process. For example, strings beginning with shell command keywords such as, "*cd* ", between *if* and *fi*, among other similar command structures, are extracted. Such examples of the shell commands are shown in table 5.3.

Based on the determined patterns to identify the shell commands, we use regular expressions to search for the specific patterns within the *strings* obtained from the malware to automate the process for all the malware samples. Although the commands contained in the strings may not be syntactically correct, *e.g.* the spaces are masked with special characters or spaces, they, however, hint to the location of shell command references. With an intent to find the actual commands, we go to the address where a particular string is quoted and disassemble at that offset. We then store the command gathered from the offset and label it as malicious. The steps used to extract the strings from the malware samples, from which the shell commands are then obtained, is shown in algorithm 1. We follow this process on all the malware samples, obtaining a total of 2,008 unique

Algorithm 1: Command extractor

Input : directory to malware

Output: commands in the malware

```
1 malwareDirectory ← directory to malware
2 for malware in malwareDirectory do
3   | Extract the strings in the program.
4   | strings ← split strings by new line
5   | for i ← 0 to len(strings) by 1 do
6   |   | if pattern in strings[i] then
7   |   |   | offset ← offset of strings[i]
8   |   |   | Traverse to the offset
9   |   |   | instructionSet ← Disassemble at offset
10  |   |   | command ← command in the instructionSet
11  |   | else
12  |   |   | continue
13  |   | end
14  | end
15 end
```

commands.

We also analyzed our malware samples to find the architecture for which they are compiled. To do so, we use the Linux *File* command that determines a file's format, target architecture, etc.. The malware distribution according to their architectures and their percentage in our dataset is depicted in table 5.2.

5.4.1.2 Assembling a Benign Dataset

To evaluate our proposed detection system, we need a dataset of benign commands along with the malicious command dataset. Compared to the malicious dataset, assembling the command line usage by a benign application is a challenging task. For example, while Linux-based applications are ubiquitous, and one can easily collect a set of Linux binaries, extracting the corresponding shell commands, and use them as a baseline for our benign dataset. Note that such binaries are not necessarily intended for embedded devices. Another approach to collect benign shell commands

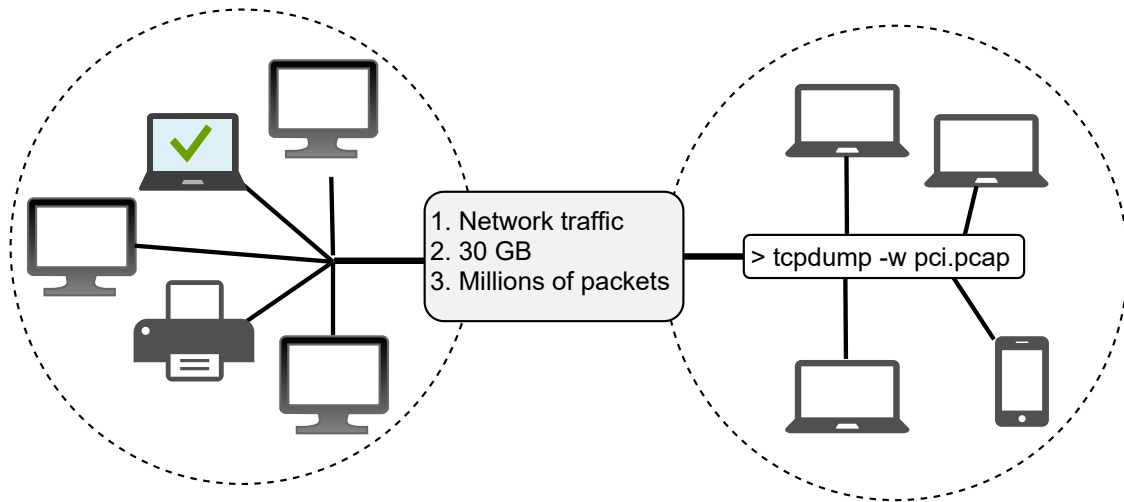


Figure 5.3: Monitoring stations for benign dataset creation. Two network implementations are used: NAT and a home network.

is by observing shell access and usage by benign users. Collecting a large-scale dataset requires monitoring network traffic to collect shell commands by benign users. Since the majority of traffic nowadays carried over HTTPS, the encrypted traffic represents a major hurdle in extracting such benign shell commands in the wild.

Therefore, we build our monitoring network setup to collect a set of benign commands. In particular, we look for commands coming from various Linux-based tools, frameworks, and software inject. Since an entry point for many malware families is the abuse of many application-layer protocols, such as HTTP, FTP, TFTP, etc., with the intent to distribute malicious payloads and scripts, we attempt to monitor those protocols in benign use setup for similar but benign data collection. As such, we build our benign command collection framework with two separate networks, as highlighted in figure 5.3.

The first network is hidden behind a NAT and consists of five stations, while the second network is a home network with 11 open ports: 21, 22, 80, 443, 12174, 1900, 3282, 3306, 3971, 5900, and 9040. The main purpose of this setup is to capture the incoming and outgoing packets from

Table 5.3: Data sources in our dataset. “Sources” is the number of files used to extract commands, while “Commands” is the total number of commands obtained from the source files.

Data Sources	#Sources	#Commands	An example
PCAP Network 1	5	1,625,143	<i>GET/update - delta/hfnkpinlhghieaddgfemjhofmfblmnib/5092/5091/193cb84a0e51a5f0ca68712ad3c7fddd65bb2d6a60619d89575bb263fc5dec26.crxHTTP/1.1nrnnHost : storage.googleapis.comnrnnConnection : keep - alivenrnnUser - Agent : Mozilla/5.0(X11; Linux;x86_64)AppleWebKit/537.36(KHTML, likeGecko)Chrome/72.0.3626.121Safari/537.36nrnnAccept-Encoding : gzip, deflatenrnnnrnn</i>
PCAP Network 2	5	4,735	<i>GET/favicon.icoHTTP/1.1nrnnConnection : closenrnnUser - Agent : Mozilla/5.0(compatible; NmapScriptingEngine; https : //nmap.org/book/nse.html)nrnnHost : 192.168.2.1nrnnnrnn</i>
Bash Commands	9	5,772	<i>sudogethttps : //download.oracle.com/otn - pub/java/jdk/8u201 - b09/42970487e3af4f5aa5bca3f542482c60/jdk - 8u201 - linux - x64.tar.gz</i>
Malware	2,891	2,008	<i>GET/cdn - cgi/l/chk_aptcha?id = %s&g - recaptcha - response = %sHTTP/1.1User - Agent : %sHost : %sAccept : */Referer : http : //%/Upgrade - Insecure - Requests : 1Connection : keep - alivePragma : no - cacheCache - Control : no - cache</i>

the home network. Our home network in this experimental setup consisted of two 64-bit Linux devices, one Amazon Alexa, one iPhone device, one Mac device with voice assistant, Siri, being continuously used, and a router. These two networks that represent the high-level illustration of our benign data collection system are depicted in figure 5.3.

In the first network, we have five devices that are used in a lab setting under “normal execution”, *i.e.* for everyday use. The network is monitored over a period of 24 hours, where all network traffic is captured (as shown in the left network in figure 5.3).

The second network is a home network designed by selecting a variety of devices, also operating under “normal execution” with the exception that the configured voice assistants in the second network are actively queried during the monitoring time. To establish a baseline, the network is monitored without the devices and continuously as the devices are added to the network. For the voice assistants, we iterate over a set of questions requiring access to the Internet and actively

monitor the traffic at the router for seven hours. Using these settings, we gathered a dataset of approximately 34 GB from the first network and approximately 1 GB from the second network.

In addition to the network traffic, we also gathered bash history data from nine volunteers. To protect the privacy of the users, we anonymize users' identity by manually observing the commands and removing every clearly identifying information, including usernames, domain names, IP addresses, etc.. Overall, we collected a dataset of approximately 143 MB of data, consisting of 5,772 commands. The collected commands correspond to services such as ssh, git, apt, Makefile, curl, etc., and generic Linux commands, such as cd, rm, chmod, cp, find, etc..

The traffic gathered from the five volunteers in Network 1 resulted in a total of 28,578,754 individual payloads, where the majority of them are encrypted and cannot be used for our data collection. On the other hand, the same dataset had 1,625,143 un-encrypted payloads, which we utilize for our data collection. Similarly, the five sources in Network 2 generated 4,735 un-encrypted payloads, while the bash data source consisted of a total of 5,772 commands from nine volunteers. A sample of the payloads from the four data sources described earlier: three benign and one malicious, is shown in table 5.3.

5.4.2 Methodology: High-Level Overview

The shell is a single point of entry for malware to launch attacks. Accordingly, detecting malicious commands before they are executed on the host will help secure the host. Even though the malware aims to exploit a vulnerability in the device to access its shell, detecting malicious commands will help mitigate such exploits. Our analysis highlights the use of shell commands for infection, propagation, and attack by malware authors. The Unix capabilities of embedded IoT devices give adversaries the required avenue to abuse the shell.

For example, the malware attempts to download a file named *DNS.txt* from a dropzone, the address

of which is stored in variable s . This variable contains a list of target devices for malware propagation. As is evident from the figure, the malware first attempts to download *DNS.txt* at home directory (default entry point upon gaining access). In case of failure, the malware attempts to initiate BusyBox by attempting three different locations, the success of which is dependent on where the BusyBox is installed on the device, followed by downloading the *DNS.txt* file, as shown in the usage of the shell for propagation in figure 5.1.

We exploit this insight to build ShellCore. In the following, we explain ShellCore and its different building blocks. The goal of ShellCore is to realize a design with an effective detection system that detects malicious files based on their usage of the shell. To this end, we break down the problem into two parts – (i) detecting malicious commands and (ii) detecting malicious files. We use real-world IoT malware samples and disassemble them to extract shell commands. We then extract features from the commands, by taking advantage of the bag of words technique that is commonly used in NLP applications. In particular, we create a corpus of words from individual commands, considering all commands in our dataset, and establish a feature space based on their occurrences in the corpus to represent each command as a feature vector. Along with the words, we also use the n-grams to represent the commands as feature vectors. Moreover, we augment the feature representations of commands in our dataset using PCA.

Upon representing the malicious and benign commands as feature vectors, ShellCore aims to detect malicious commands. To do so, ShellCore employs machine learning algorithms to classify commands. To evaluate ShellCore, we train and test the detection model using our dataset. We use cross-validation to limit the bias and measure the generalization capabilities of the model. Using the same model architecture, we extend the detection system to detect malicious files. For detecting malicious files, we group the commands by each malware sample and benign application in one single record that is represented as one feature vector to be analyzed. This is well abstracted in the command extraction block in figure 5.2.

5.5 Detection Model

We utilize the capabilities of deep learning techniques to detect malicious commands using NLP-based feature generation. To help better learn the specifics of shell commands, we tune the default NLP algorithms to enrich the feature representation of the commands. We represent the commands as a feature vector using the bag of words method, reduce the features using PCA, and use ML-based algorithms for the malicious command and sample detection.

5.5.1 Featurization

Feature representation is a method to present the attributes of samples. It is the process of cleansing and linking the data such that it is transformed in a format that is understood by the employed algorithms for detection. In this section, we discuss selecting features that better represent the characteristics of the samples in the dataset. There are many methods of featurization depending upon the nature of the data. Considering the textual nature of our samples, we focus on text-based featurization methods. Towards this, we leverage the traditional NLP-based approach by considering words in the samples as features. Additionally, since such an approach misses on very crucial attributes, we then employ a customized NLP approach to meet our goals.

5.5.1.1 Traditional NLP-based model

We leverage NLP for feature generation, by considering independent words as features and occurrence of space and/or characters as tokenizers, while words with a length greater than two are considered in the bag of words for feature vector creation. We adopt the bag of words approach along with n -grams. Let I_1 be the words in a command, and n is the total number of words in the command. Therefore, each word in the command can be represented as I_{1i} , where $i \in [1, n]$, such that $I_1 = I_{11}, I_{12}, I_{13}, \dots, I_{1n}$.

5.5.1.2 Customized NLP-based model

The traditional NLP-based approach does not take into consideration the operational symbols in a command, which undermines many discriminating and dominant characteristics of the shell command, thereby not representing the commands accurately. The presence of many shell commands utilizing keywords $l \leq 2$ called for building a more accommodating feature generation mechanism. To do so, we change the boundaries of the definition of a word by considering every space, special characters, alphabets, and numbers as words, along with n-grams and command statistics. This augments our vocabulary with more granular features to capture the attributes precisely. Let I_2 be a representation of each character, alphabet, number, etc. constituting a command, and n is the total number of such constituents in the command. Therefore, every such constituent in the command can be represented as I_{2j} , where $j \in [1, n]$, such that $I_2 = I_{21}, I_{22}, I_{23}, \dots, I_{2n}$.

5.5.2 Feature Representation

To represent every element in the dataset from a defined reference point, we represent every element with respect to axes in a space. In particular, every command/sample in the dataset is represented as a feature vector in the defined feature space. To train our detection model, each command is represented as a feature vector, where every element represents a distinct feature of the input. In this regard, we begin by finding the feature space to determine the dimensionality of the vectors. Particularly, the commands are augmented such that every feature of the commands in the dataset has a representation in the feature space. Every command in the dataset is then represented in a space of n axes, where n is the size of feature space. To do so, we devise multiple representations of commands, such as by including the words in the commands and by splitting the commands by spaces and every special character. We also form a feature vector by considering each and every letter and special character as features combined with the special characters. We implement the bag of words method to define our feature space. The rest of this section explains

our feature representation mechanism using the bag of words technique.

Bag of Words as Command Embedding. We realize a representation of commands/samples using the bag of words technique. Depending upon the splitting pattern of the samples, we create a central vector that stores all the words in the samples. Each sample in the dataset is then mapped to an index in the sparse vector representation, *i.e.* feature vector for every element in a dataset, where the vector has an index for every word in the vocabulary—every word, in n different words in the central vector, is converted into n bits, among which the bits are left zero, except for their occurrence (multi-hot encoding). Translating this into perspective, to generate the vector space, we add every word to an array. For every sample, we initialize its feature vector with a size equal to the bag of words. For every occurrence of a word in a sample, its index location is incremented. Therefore, every feature vector of a sample represents the frequency of the corresponding word in the dictionary.

Encoding Syntax. An important characteristic of the commands is their syntax. This syntax is dependent on the structure of the command. Therefore, in addition to the standard features gathered from the commands, we also augment the feature space with feature proximity, to capture the structure of the commands. To do so, we also include the features of n -grams. Every n contiguous words in a sample's shell command are considered a feature. When using n -grams as features, every n contiguous words occurring in a sample are added to the bag of words corresponding to them in the feature space.

For each of the two models as aforementioned, we create separate bag of words, such that, the bag contains all the words I_{1i}^k , where $i \in [1, n]$ and $k \in [1, m]$, such that n is the total number of words in a command and k is the total number of commands in the dataset. along with the n -grams. Therefore, the words in all the commands as per the traditional NLP model, can be combined as $I_{11}^1, I_{11}^2, I_{11}^3, \dots, I_{12}^1, I_{13}^1, \dots, I_{1n}^m$. Let B be the bag of word for the dataset, such that $B = B_1, B_2, B_3, \dots, B_t$, where $t \leq m * n$ and B_p , such that $p \in [1, t]$, is unique in B . Moving

forward, each command I_i , where $i \in [1, n]$, can be represented as a feature vector (F) with respect to the bag of words B , such that the t^{th} index can be represented as the frequency of occurrence, of the t^{th} word in the bag, in the command. $F = f_{B_1}, f_{B_2}, f_{B_3}, \dots, f_{B_t}$, such that f_{B_p} , where $p \in [1, t]$, depicts the frequency of the word, appearing at index p in the bag B , in the command I_i .

5.5.3 Feature Reduction

We capture as many features as possible to achieve accurate results. However, beyond a certain point, the performance of the model becomes inversely proportional to the number of features. The usage of a wide variety of features to represent samples leads to a very high dimensional feature vector which leads to (i) high cost to perform learning and (ii) overfitting, *i.e.* the model may perform very well on the training dataset, but poorly on the test dataset. Dimensionality reduction or feature reduction is applied with an aim to address the two problems. We implement the principal component analysis (PCA) for feature reduction to improve the performance and the quality of our classifier of ShellCore. PCA features (components) are extracted from the raw features. It is a statistical technique used to extract features from multiple raw features, where raw features are of n -grams and statistical measurements. PCA creates new variables, named Principal Components (PCs). PCs are linear combinations of the original variables, where a possible number of correlated variables are transformed into a low dimension of uncorrelated PCs. It normalizes the dataset by transforming them into a normal distribution with the same standard deviation [49]. This generates a standard representation of variables in order to identify a subset of them that can best characterize the underlying data [118]. We reduce the d -dimensional vector representation of commands to q number of principal components onto which the retained variance under projection is maximal.

5.5.4 Classification Methods

After representing each sample as a feature vector, we classify these samples into classes/targets – malicious or benign. Considering the text-based non-linear features and high dimensionality of the feature vectors, we utilize two well-known machine learning algorithms for non-linear classification tasks: Deep Neural Networks (DNN) and Support Vector Machine (SVM).

1) Deep Neural Networks (DNN). DNN is a type of connected and feed-forward neural networks with multiple hidden layers between the input and output layers. The hidden layers consist of a number of parallel neurons, connected with a certain weight to all nodes in the following layers to generate a single output for the next layer.

Given a feature vector X of length q and target y , the DNN-based classifier learns a function $f(.) : R^q \rightarrow R^o$, where q is the input's dimension and o is the output's dimension. With multiple hidden layers, the dimension of the output of every hidden layer decreases with transformation. Each neuron in the hidden layer transforms the values of the preceding layer using linearly weighted summation, $w_1 + w_2 + w_3 + \dots w_q$, which passes through a non-linear activation function such that, $g(.) : R \rightarrow R$. The output of the hidden layers is then fed to the output layer, and passed to an activation function f , outputting the prediction of the classifier.

2) Support Vector Machine (SVM). SVM classifies the data by finding the best hyperplane that separates the data from the two classes. 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 hyperplane with the largest margin between the two classes [111]. We use SVM due to its effectiveness in high dimensional spaces, its effectiveness when the dimension is greater than number of samples, and it being memory efficient. To achieve the goal, we utilize the following decision function [48, 25] $sgn(\sum_{i=1}^n y_i \alpha_i K(x_i, x) + \rho)$ where $x_i, i \in [1, q]$, is the

Table 5.4: Size characteristics of the different datasets. Net. stands for Network.

Data	Commands	Max length	Min length	Avg. length	Median length	Standard deviation
Net. 1	1,625,143	1,564	52	184.68	185	4.88
Net. 2	4,755	1,536	8	209.01	167	146.26
Bash	5,772	356	2	23.00	14	27.71
Malware	2,008	984	5	293.91	384	168.03

training feature vector of sample, ρ is the hyperplane margin, y_i are the output labels, and the kernel function $K(x_i, x)$ is defined as, $K(x_i, x) = \phi(x_i)^T \phi(x_j)$

5.6 Evaluation and Discussion

In this section, we evaluate ShellCore’s performance and discuss the results. We start by classifying individual commands using the NLP-based approach. In all evaluations, our model exhibits high accuracy. We divide our evaluation into two parts. First, we build a detection system to detect malicious commands by considering every individual command in the dataset. Second, this detection system is then extended for detecting malicious files, where the above commands corresponding to an application are combined together when representing a single file as a feature vector of multiple commands.

In the following, we provide further details of the datasets and their characteristics, and the utilized evaluation metric. We then describe the traditional and customized NLP-based models. Finally, we describe how these two models are leveraged for detecting individual commands and malicious files.

Dataset. We notice that the low deviation of length in commands in Net. 1 indicates that the commands have a similar length. Moreover, we notice that Net. 2 (corresponding to the IoT devices setting) and Malware datasets have closest lengths overall, per the average and standard

deviation characteristics of their distributions. The number of commands as well as the length statistics (maximum, minimum, average, median, and standard deviation) are shown in table 5.4.

Parameters Tuning. For a better representation of commands' features, we utilize n -grams. Particularly, we use 1- to 5-grams. For the DNN-based classifier, we also try multiple combinations of parameters to tune the classifier for better performance. We achieve the best performance when using five hidden layers.

K-Fold Cross-Validation. The evaluation of a machine learning algorithm depends on the training and testing data. To generalize the evaluation, cross-validation is used. For K-fold cross-validation, the data are sampled into K subsets. Then, the model is trained on one of the K subsets and tested on the other K-1 subsets. The process is then repeated allowing each subset to be the testing data while the remaining nine are used for training the model. The performance results are then taken as the average of all runs. In this work, We used 10-fold cross-validation to minimize bias towards a certain data. For 10-fold cross-validation, the model is trained for one sample and it is tested on the other nine samples, for ten iterations.

Evaluation Metric. We evaluate the results of the classification task in terms of accuracy, false-negative rate, and false-positive rate. Accuracy (AC) is defined as the sum of true positive and true negative divided by the sum of true positive, false positive, false negative, and true negative. For a class C_i , (where $i \in \{1, 2, 3\}$), False Positive (FP), False Negative (FN), True Positive (TP), and True Negative (TN) are defined as:

1. TP of C_i is all C_i instances classified correctly
2. TN of C_i is all non- C_i not classified as C_i
3. FP of C_i is all non- C_i instances classified as C_i
4. FN of C_i is all C_i instances not classified as C_i .

Table 5.5: Evaluation results: Malicious commands detection.

	Traditional NLP		Customized NLP
	SVM	DNN	DNN
AC	0.9290	0.9340	0.9530
FNR	0.0317	0.0210	0.0271
FPR	0	0.0085	0.0853

Mathematically, Accuracy (AC) is defined as the sum of true positive and true negative divided by the sum of true positive, false positive, false negative, and true negative. The False-Negative Rate (FNR) is the sum of false negative divided by the sum of the true positive and false negative. The False-Positive Rate (FPR) is the sum of false positive divided by the sum of false positive and true negative. We report the metrics as mean accuracy, mean FNR, and mean FPR for the ten folds.

5.6.1 Traditional NLP-based model

The traditional NLP-based learning model uses words as features, with spaces and other special characters as tokenizers. Additionally, it does not consider words less than three characters long. Moreover, to better represent the locality of the words, the model utilizes n -grams. Particularly, it uses 1- to 5-grams. With 10-fold cross-validation, the model achieves the results for both SVM and the DNN-based classifiers, with the DNN-based classifier yielding better results, as shown in table 5.5.

5.6.2 Customized NLP-based model

We note that the traditional approach only considers words, and neglects the characters, spaces, and words that have a length of less than three. This, in turn, presents a major shortcoming, since a large number of command keywords have a length less than three, including *cd*, *ls*, *etc.*,

or consist of special characters, such as | |, &&, etc.. To address the shortcoming, we create the feature generation step such that it considers these important domain-specific characteristics that would otherwise be ignored. To do so, we change the way in which a word is defined by carefully declaring the tokenizers such that no character is ignored. Subsequently, the changed bag of words considers the character-level, and contain every letter, number, and character represented as an individual feature. Moreover, to capture the placement of the letters, characters, and spaces, we also consider combinations of these elements in the form of n -grams (up to 5-grams) into a vector space. Finally, for feature reduction, we use PCA such that it covers for 99% of the variance in the training dataset.

5.6.3 *Detecting Malicious Commands*

We use ShellCore for detecting individual malicious commands. We first present the results of the traditional model followed by the customized NLP model.

When used with DNN, the traditional model provides an accuracy of 93.4% along with an FNR of only 2.1% and FPR of 0.85%. While the SVM shows an accuracy of 92.9%. We observe that the accuracy rate, although greater than that of SVM, is relatively low. We, therefore, work towards improving the accuracy and reducing the false positives and the false negatives by considering other important features of the samples by evaluating the customized model. Moreover, we observe improved performance with DNN classifier. Therefore, we select the DNN-based model as the classifier for ShellCore, as can be observed in table 5.5.

We test the veracity of the customized NLP-based model for detecting individual malicious commands over the same dataset. The approach improves the performance of the model, where the improvement in the accuracy over the traditional model is about 2%, *i.e.* 95.3%. The results of the approach is shown in table 5.5

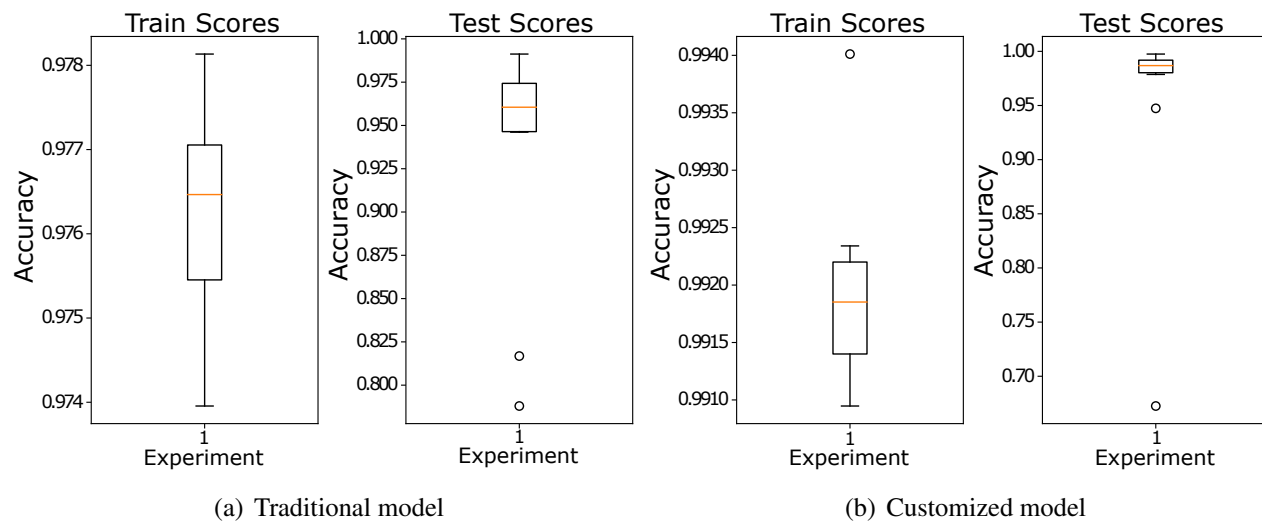


Figure 5.4: Accuracy variations over the 10 fold cross-validation when detecting malicious commands.

Table 5.6: Evaluation results of malware detection.

Metric	Traditional NLP	Customized NLP
AC	0.996	0.998
FNR	0.001	0.002
FPR	0.006	0.001

The difference in the evaluation results of the two models is expected and is mainly because of the difference in the features of the two models, and hint on the importance of special characters and letters with length less than three. Moreover, the accuracy is in the range of 97.4% to 97.8% in every fold when using the traditional model for the training phase, while the accuracy for the testing phase is in the range of 78.8% to 99.1%. For the customized model, however, the testing accuracy range is between 67.3% to 99.7, with only once going less than 95%. Overall, the 95.3% accuracy in detecting malicious commands shows the effectiveness of the latter model, as shown in the accuracy for each fold in the 10-fold cross-validation in figure 5.4.

5.6.4 Malware Detection

The next natural step is to generalize from the shell command detection to binaries (malware) detection, which we pursue in the following. In particular, in this section, we classify files as malicious or benign using vectors of feature per file that combine the feature values of the shell commands associated with each file.

Dataset. We use the same malware samples and the collected benign dataset as in the other experiments and analyses. Particularly, we cluster the commands by their source. For the malicious dataset, we group the commands by malware sample. However, we do not have designated source information for the benign commands dataset. To generate a benign dataset, we cluster the commands in the benign dataset under separate file labels. Keeping in mind the susceptibility of clustering to human and algorithmic biases, we cluster the commands such that the probability distribution of the benign dataset across files remains intact. We observe that the probability distribution of the number of commands in every malicious software and sample the distribution such that every sample represents the number of commands in a benign software. With the number of commands in every file, we then randomly select those many commands from the pool of benign commands.

We use ShellCore to train and test the model over the file specific dataset. Particularly, commands corresponding to a file are represented as a feature vector of that file. Similar to individual commands detection, we try both the traditional and customized NLP-based approaches. The performance results of ShellCore for malware detection is represented in figure 5.4.

Also, it is demonstrated that ShellCore can detect malicious commands with high accuracy and very low error (false positive and false negative). Moreover, the results show that the accuracy of ShellCore is improved when using the customized NLP-based model, as shown in table 5.6.

5.6.5 Discussion

This chapter studies the usage of shell commands, with the aim to detect malicious shell commands and malware utilizing them. To do so, the proposed system, ShellCore, leverages the power of machine learning- and deep learning-based algorithms. The system is then evaluated, and the results are presented. Our results show that, however, ShellCore does not detect individual commands with very high accuracy, although performing very well in detecting malware that uses those commands. We elaborate on those results.

Detecting individual shell commands. Although researchers have looked into the malicious usage of Windows PowerShell, and except for analyzing the vulnerabilities in Linux shell (*e.g.*, shellshock), the malicious usage of shell commands has not been analyzed in the past. Prior works have analyzed and detected the use of shell commands to propagate attacks *e.g.*, sending malicious bots [41], and installing ELF executables on Android systems [102]. Given the larger ecosystem of connected embedded devices with Linux capabilities, and sensing the urgency, we analyze the usage of shell commands used by malware. This work presents a system to detect the malicious commands with 95.3% accuracy.

Malware Detection. IoT malware has been on the rise. Given the difficulty of obtaining samples, very few works have been done on detecting IoT malware, and even less using residual strings in the binaries either. Section 5.2 discusses the methods that work on detecting IoT malware. In this work, we use the commands in the malware samples for detecting them. Our detection model achieves an accuracy of 99.8% with FNR and FPR of 0.2% and 0.1%, respectively. As malware abuse the shell of the host device, detecting them at the shell will safeguard the device from becoming infected. Additionally, malware access a device by breaking into the host device by launching a dictionary attack, typically a single shell command execution. Alternatively, a host device can also be infected by a zero-day vulnerability or an outdated device with an existing

exploitable vulnerability, among others, which are all also executed by individual shell commands. For a successful event, where the adversary breaks into a host, it will then abuse the shell to infect the host, followed by propagating the malware, and creating a network of botnets to launch attacks. As such, having a detector of such a high accuracy, at both the individual command level and malware sample level, with low FPR and FNR, will help stop the host device from being used as an intermediary target for launching attacks, despite the presence of vulnerabilities or the host. This makes this work very timely and necessary.

Dataset. One of the biggest challenge in developing and studying systems for detecting malicious commands is the absence of a benign dataset. We propose a way to assemble such a dataset. Additionally, to reduce the bias, we collect the dataset from two different networks with a diverse set of devices and behaviors. Moreover, we assemble the benign usage of shell by end-users, based on several volunteers' device usage. For generating benign dataset for malware detection, we use mathematical models that aggregate files into groups while respecting the distribution of the original sample size in both benign and malicious files. As such, and to remove the possible human bias, we cluster the benign commands by files, such that the probability distribution of the number of commands in both malicious and benign dataset is the same. While the shell commands of both the benign and malicious datasets used in this study will be made public, we leave exploring additional methods for obtaining benign datasets and contrasting them to the datasets used in this study as a future work.

5.7 Conclusion

In this chapter, we propose a system, called ShellCore, for detecting malicious commands and malware in IoT devices. We analyzed malicious shell commands from a dataset of 2,891 IoT malware samples, along with a dataset of benign shell commands assembled corresponding to benign applications. ShellCore leverages deep learning-based algorithms to detect malicious commands and

files, and an NLP-based approaches for feature creation. ShellCore detects individual malicious commands with an accuracy of more than 95%, and an accuracy of around 99.8%, with low FPR and FNR, when detecting malware. The results reflect that despite a comparatively low detection rate for individual commands, the proposed model is able to detect their source with high accuracy.

CHAPTER 6: CONCLUSION AND FUTURE WORK

At the beginning of this research, an in-depth analysis of the general and algorithmic features of the malware CFGs was conducted between two prominent malware datasets, IoT and Android malware. We constructed the CFGs, as an abstract representation of the software binaries, and conducted a deep comparative study between the two malware types. We highlighted the CFG-based shift between them by analyzing both general characteristics, such as the number of nodes and edges, and graph algorithmic features, such as average shortest path, betweenness centrality, closeness centrality, density, etc.. To understand the difference, we contrasted those features to uncover the similarities and differences between IoT and Android malware. We observed various interesting findings, such as the different sizes of nodes and edges. We found a larger number of nodes in Android malware, compared to the IoT malware, highlighting its higher order of complexity. We also observed the prevalence of unreachable code in Android malware, noted by the multiple components, which is a sign of decoy functions for circumventing malware analysts.

Then, using the aforementioned general and algorithmic features of the CFGs, as shown to be discriminative, we built an IoT classification and detection system. Toward this, we assembled a dataset of IoT benign samples for detection. Specifically, we build a CFG-based detection model to detect IoT malware by utilizing various CFG features of IoT benign and malware samples. We evaluated the detection model by leveraging four different classifiers (LR, SVM, RF, and CNN) and achieved an accuracy rate of $\approx 99.66\%$ with 0.33% FNR and 0.33% FPR using CNN. Moreover, we classified the IoT malware based on their families and achieved an accuracy of $\approx 99.32\%$ with 2.93% FNR and 0.45% FPR.

Further, to tackle the need to detect adversarial ML attacks that generates AEs, we proposed Soteria, an adversarial ML detector for IoT malware. More precisely, Soteria defends the CFG-based classifiers for malware detection against the AEs. Soteria consists of two main components: an

AEs detector and a family-based classifier. The AE detector is a CFG-based model capable of detecting AEs without training the model over adversarial samples. The model computes the reconstruction error between the input data and the reconstructed output of the auto-encoder, and uses a threshold to detect the AEs with an overall accuracy of 97.79%. Moreover, the CFG-family-based classifier achieved an accuracy of 99.91% on the clean samples. To increase the robustness of Soteria, the two models are independent.

Finally, we proposed ShellCore, an IoT malware detection system to detect malicious commands in the Linux-based IoT devices. We assembled a dataset of IoT malware and benign samples and analyzed the malicious shell commands of the IoT malware. Then, we used NLP-based approaches for feature generation, and neural network-based algorithms to detect malicious commands and files. We achieved an accuracy of more than 95% to detect individual malicious commands, and $\approx 99.8\%$, with low FPR and FNR, to detecting malware files. Despite the low detection rate for individual commands, the model detects their files with a high accuracy rate.

As future work, The obfuscated binary will result in hiding some parts of the code or functions, resulting in an incomplete CFG representation of the software, hence, affecting the feature representation of the malware. Moreover, CFG does not necessarily reflect the actual code representation. So, the code can be edited without changing its functionality; thus, it would affect the structure of the CFG, which might be exploited by the adversary to evade detection models. Additionally, we will extend our static analysis works to be dynamic by executing the IoT malware on real-time IoT devices to understand the behaviors of the malware, including execution pattern, memory traces, API calls, etc..

APPENDIX: COPYRIGHT INFORMATION

Consent to Publish

Lecture Notes in Computer Science

Title of the Book or Conference Name: International Conference on Computational Data & Social Science
Volume Editor(s) Name(s): Xuemin.Chen, Arun.Sen, Wei.Li and My.T..Thai
Title of the Contribution: Graph-based Comparison of IoT and Android Malware
Author(s) Full Name(s): Hisham.Alasmary*, Afsah.Anwar*, Jeman.Park*, Jinchun.Choi*
Corresponding Author's Name, Affiliation Address, and Email: DaeHun Nang**, and Aziz Mohaisen*
University of Central Florida*, Inha University**

When Author is more than one person the expression "Author" as used in this agreement will apply collectively unless otherwise indicated.

The Publisher intends to publish the Work under the imprint **Springer**. The Work may be published in the book series **Lecture Notes in Computer Science (LNCS, LNAI or LNBI)**.

§ 1 Rights Granted

Author hereby grants and assigns to **Springer Nature Switzerland AG, Gewerbstrasse 11, 6330 Cham, Switzerland** (hereinafter called **Publisher**) the exclusive, sole, permanent, world-wide, transferable, sub-licensable and unlimited right to reproduce, publish, distribute, transmit, make available or otherwise communicate to the public, translate, publicly perform, archive, store, lease or lend and sell the Contribution or parts thereof individually or together with other works in any language, in all revisions and versions (including soft cover, book club and collected editions, anthologies, advance printing, reprints or print to order, microfilm editions, audiograms and videograms), in all forms and media of expression including in electronic form (including offline and online use, push or pull technologies, use in databases and data networks (e.g. the Internet) for display, print and storing on any and all stationary or portable end-user devices, e.g. text readers, audio, video or interactive devices, and for use in multimedia or interactive versions as well as for the display or transmission of the Contribution or parts thereof in data networks or search engines, and posting the Contribution on social media accounts closely related to the Work), in whole, in part or in abridged form, in each case as now known or developed in the future, including the right to grant further time-limited or permanent rights. Publisher especially has the right to permit others to use individual illustrations, tables or text quotations and may use the Contribution for advertising purposes. For the purposes of use in electronic forms, Publisher may adjust the Contribution to the respective form of use and include links (e.g. frames or inline-links) or otherwise combine it with other works and/or remove links or combinations with other works provided in the Contribution. For the avoidance of doubt, all provisions of this contract apply regardless of whether the Contribution and/or the Work itself constitutes a database under applicable copyright laws or not.

The copyright in the Contribution shall be vested in the name of Publisher. Author has asserted his/her right(s) to be identified as the originator of this Contribution in all editions and versions of the Work and parts thereof, published in all forms and media. Publisher may take, either in its own name or in that of Author, any necessary steps to protect the rights granted under this Agreement against infringement by third parties. It will have a copyright notice inserted into all editions of the Work according to the provisions of the Universal Copyright Convention (UCC).

The parties acknowledge that there may be no basis for claim of copyright in the United States to a Contribution prepared by an officer or employee of the United States government as part of that person's official duties. If the Contribution was performed under a United States government contract, but Author is not a United States government employee, Publisher grants the United States government royalty-free permission to reproduce all or part of the Contribution and to authorize others to do so for United States government purposes. If the Contribution was prepared or published by or under the direction or control of the Crown (i.e., the constitutional monarch of the Commonwealth realm) or any Crown government department, the copyright in the Contribution shall, subject to any agreement with Author, belong to the Crown. If Author is an officer or employee of the United States government or of the Crown, reference will be made to this status on the signature page.

16.01.2018 10:38

§ 2 Rights Retained by Author

Author retains, in addition to uses permitted by law, the right to communicate the content of the Contribution to other research colleagues, to share the Contribution with them in manuscript form, to perform or present the Contribution or to use the content for non-commercial internal and educational purposes, provided the original source of publication is cited according to the current citation standards in any printed or electronic materials. Author retains the right to republish the Contribution in any collection consisting solely of Author's own works without charge, subject to ensuring that the publication of the Publisher is properly credited and that the relevant copyright notice is repeated verbatim. Author may self-archive an author-created version of his/her Contribution on his/her own website and/or the repository of Author's department or faculty. Author may also deposit this version on his/her funder's or funder's designated repository at the funder's request or as a result of a legal obligation. He/she may not use the Publisher's PDF version, which is posted on the Publisher's platforms, for the purpose of self-archiving or deposit. Furthermore, Author may only post his/her own version, provided acknowledgment is given to the original source of publication and a link is inserted to the published article on the Publisher's website. The link must be provided by inserting the DOI number of the article in the following sentence: "The final authenticated version is available online at [https://doi.org/\[insert DOI\]](https://doi.org/[insert DOI])." The DOI (Digital Object Identifier) can be found at the bottom of the first page of the published paper.

Prior versions of the Contribution published on non-commercial pre-print servers like ArXiv/CoRR and HAL can remain on these servers and/or can be updated with Author's accepted version. The final published version (in pdf or html/xml format) cannot be used for this purpose. Acknowledgment needs to be given to the final publication and a link must be inserted to the published Contribution on the Publisher's website, by inserting the DOI number of the article in the following sentence: "The final authenticated publication is available online at [https://doi.org/\[insert DOI\]](https://doi.org/[insert DOI])".

Author retains the right to use his/her Contribution for his/her further scientific career by including the final published paper in his/her dissertation or doctoral thesis provided acknowledgment is given to the original source of publication. Author also retains the right to use, without having to pay a fee and without having to inform the Publisher, parts of the Contribution (e.g. illustrations) for inclusion in future work. Authors may publish an extended version of their proceedings paper as a journal article provided the following principles are adhered to: a) the extended version includes at least 30% new material, b) the original publication is cited, and c) it includes an explicit statement about the increment (e.g., new results, better description of materials, etc.).

§ 3 Warranties

Author agrees, at the request of Publisher, to execute all documents and do all things reasonably required by Publisher in order to confer to Publisher all rights intended to be granted under this Agreement. Author warrants that the Contribution is original except for such excerpts from copyrighted works (including illustrations, tables, animations and text quotations) as may be included with the permission of the copyright holder thereof, in which case(s) Author is required to obtain written permission to the extent necessary and to indicate the precise sources of the excerpts in the manuscript. Author is also requested to store the signed permission forms and to make them available to Publisher if required.

Author warrants that Author is entitled to grant the rights in accordance with Clause 1 "Rights Granted", that Author has not assigned such rights to third parties, that the Contribution has not heretofore been published in whole or in part, that the Contribution contains no libellous or defamatory statements and does not infringe on any copyright, trademark, patent, statutory right or proprietary right of others, including rights obtained through licences; and that Author will indemnify Publisher against any costs, expenses or damages for which Publisher may become liable as a result of any claim which, if true, would constitute a breach by Author of any of Author's representations or warranties in this Agreement.

Author agrees to amend the Contribution to remove any potential obscenity, defamation, libel, malicious falsehood or otherwise unlawful part(s) identified at any time. Any such removal or alteration shall not affect the warranty and indemnity given by Author in this Agreement.

§ 4 Delivery of Contribution and Publication

Author agrees to deliver to the responsible Volume Editor (for conferences, usually one of the Program Chairs), on a date to be agreed upon, the manuscript created according to the Publisher's Instructions for Authors. Publisher will undertake the reproduction and distribution of the Contribution at its own expense and risk. After submission of the Consent to Publish form signed by the Corresponding Author, changes of authorship, or in the order of the authors listed, will not be accepted by the Publisher.

§ 5 Author's Discount for Books

Author is entitled to purchase for his/her personal use (if ordered directly from Publisher) the Work or other books published by Publisher at a discount of 40% off the list price for as long as there is a contractual arrangement between Author and Publisher and subject to applicable book price regulation. Resale of such copies is not permitted.


§ 6 Governing Law and Jurisdiction

If any difference shall arise between Author and Publisher concerning the meaning of this Agreement or the rights and liabilities of the parties, the parties shall engage in good faith discussions to attempt to seek a mutually satisfactory resolution of the dispute. This agreement shall be governed by, and shall be construed in accordance with, the laws of Switzerland. The courts of Zug, Switzerland shall have the exclusive jurisdiction.

Corresponding Author signs for and accepts responsibility for releasing this material on behalf of any and all Co-Authors.

Signature of Corresponding Author:

Date:

.....  10/2/2018

- I'm an employee of the US Government and transfer the rights to the extent transferable (Title 17 §105 U.S.C. applies)
- I'm an employee of the Crown and copyright on the Contribution belongs to the Crown

For internal use only:
Legal Entity Number: 1128 Springer Nature Switzerland AG
Springer-C-CTP-01/2018

IEEE COPYRIGHT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

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

Alasmay, Hisham; Khormali, Aminollah; Anwar, Afsah; Park, Jeman; Choi, Jinchun; Abusnaina, Ahmed; Awad, Amro; Nyang, DaeHun; Mohaisen, Aziz

IEEE Internet of Things Journal

COPYRIGHT TRANSFER

The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

GENERAL TERMS

1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. The undersigned agrees that publication with IEEE is subject to the policies and procedures of the [IEEE PSPB Operations Manual](#).
4. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing copyright transfer shall be null and void. In this case, IEEE will retain a copy of the manuscript for internal administrative/record-keeping purposes.
5. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.
6. The author hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the author has obtained any necessary permissions. Where necessary, the author has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE

BY TYPING IN YOUR FULL NAME BELOW AND CLICKING THE SUBMIT BUTTON, YOU CERTIFY THAT SUCH ACTION CONSTITUTES YOUR ELECTRONIC SIGNATURE TO THIS FORM IN ACCORDANCE WITH UNITED STATES LAW, WHICH AUTHORIZES ELECTRONIC SIGNATURE BY AUTHENTICATED REQUEST FROM A USER OVER THE INTERNET AS A VALID SUBSTITUTE FOR A WRITTEN SIGNATURE.

Aziz Mohaisen

Signature

27-06-2019

Date (dd-mm-yyyy)

Information for Authors

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the

requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/authorrightsresponsibilities.html Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

- Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
- Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
- Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use. The IEEE Intellectual Property Rights office must handle all such third-party requests.
- Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

AUTHOR ONLINE USE

- **Personal Servers.** Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
- **Classroom or Internal Training Use.** An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
- **Electronic Preprints.** Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the IEEE work with a Digital Object Identifier (DOI) or link to the article abstract in IEEE Xplore, or (2) the accepted version only (not the IEEE-published version), including the IEEE copyright notice and full citation, with a link to the final, published article in IEEE Xplore.

Questions about the submission of the form or manuscript must be sent to the publication's editor.

Please direct all questions about IEEE copyright policy to:

IEEE Intellectual Property Rights Office, copyrights@ieee.org, +1-732-562-3966

LIST OF REFERENCES

- [1] 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*, pages 101–114, 2018.
- [2] A. Abusnaina, H. Alasmary, M. Abuhamad, S. Salem, D. Nyang, and A. Mohaisen. Subgraph-based adversarial examples against graph-based iot malware detection systems. In *8th International Conference on Computational Data and Social Networks, CSoNet*, pages 268–281, 2019.
- [3] 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.
- [4] S. Alam, R. N. Horspool, I. Traoré, and I. Sogukpinar. A framework for metamorphic malware analysis and real-time detection. *Computers & Security*, 48:212–233, 2015.
- [5] H. Alasmary, A. Abusnaina, R. Jang, M. Abuhamad, A. Anwar, D. Nyang, and D. Mohaisen. Soteria: Detecting adversarial examples in control flow graph-based malware classifiers. In *the 40th IEEE International Conference on Distributed Computing Systems, ICDCS*, 2020.
- [6] H. Alasmary, A. Anwar, A. Abusnaina, M. Abuhamad, and D. Mohaisen. ShellCore: Defeating IoT malware through in-depth analysis and detection of IoT shell code. In *submission to the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec*, 2020.

- [7] 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*, pages 259–272, 2018.
- [8] H. Alasmary, A. Khormali, A. Anwar, J. Park, J. Choi, A. Abusnaina, A. Awad, D. Nyang, and A. Mohaisen. Analyzing and Detecting Emerging Internet of Things Malware: A Graph-based Approach. *IEEE Internet of Things Journal*, 2019.
- [9] B. Alipanahi, A. Delong, M. T. Weirauch, and B. J. Frey. Predicting the sequence specificities of DNA-and RNA-binding proteins by deep learning. *Nature biotechnology*, 33(8):831, 2015.
- [10] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth. Evading machine learning malware detection. *Black Hat*, 2017.
- [11] K. Angrishi. Turning Internet of Things IoT into Internet of Vulnerabilities IoV : IoT Botnets. *Computing Research Repository CoRR*, abs/1702.03681, 2017.
- [12] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the Mirai Botnet. In *Proceedings of the 26th USENIX Security Symposium*, pages 1093–1110, 2017.
- [13] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon. From throw-away traffic to bots: Detecting the rise of DGA-based malware. In *USENIX Security*, pages 491–506, 2012.
- [14] A. Azmoodeh, A. Dehghantanha, and K.-K. R. Choo. Robust malware detection for Internet Of (Battlefield) Things devices using deep eigenspace learning. *IEEE Transactions on Sustainable Computing*.

- [15] E. Bertino and N. Islam. Botnets and Internet of Things security. *IEEE Computer*, 50(2):76–79, 2017.
- [16] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi. EXPOSURE: finding malicious domains using passive DNS analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2011.
- [17] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- [18] 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*, pages 129–143, 2006.
- [19] N. Carlini and D. A. Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy, SP*, pages 39–57, 2017.
- [20] 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*, pages 164–181, 2013.
- [21] CBSNews. Baby monitor hacker delivers creepy message to child. Available at [Online] : <https://tinyurl.com/y9g9948c>, 2015.
- [22] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *Proceedings of the Asia Pacific Workshop on Systems, APSys*, page 5, 2011.
- [23] Christo Petrov. Internet of Things Statistics 2019 [The Rise Of IoT]. Available at [Online]: <https://tinyurl.com/rasheg6>, 2019.
- [24] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

- [25] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.
- [26] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti. Understanding Linux malware. In *IEEE Symposium on Security & Privacy*, 2018.
- [27] Z. Cui, F. Xue, X. Cai, Y. Cao, G. Wang, and J. Chen. Detection of malicious code variants based on deep learning. *Trans. Industrial Informatics*, 14(7):3187–3196, 2018.
- [28] Z. Cui, F. Xue, X. Cai, Y. Cao, G.-g. Wang, and J. Chen. Detection of malicious code variants based on deep learning. *IEEE Transactions on Industrial Informatics*, 14(7):3187–3196, 2018.
- [29] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu. Large-scale malware classification using random projections and neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP*, pages 3422–3426, 2013.
- [30] Developers. Cyberiocs. Available at [Online]: <https://freeiocs.cyberiocs.pro/>, 2019.
- [31] Developers. Github. Available at [Online]: <https://github.com/>, 2019.
- [32] Developers. Radare2. Available at [Online]: <http://www.radare.org/r/>, 2019.
- [33] Developers. VirusTotal. Available at [Online]: <https://www.virustotal.com>, 2019.
- [34] Developers. Cve-2010-4258: Turning denial-of-service into privilege escalation. Available at [Online]: <https://tinyurl.com/y8ex6ltj>, Retrieved, 2010.
- [35] G. S. Dhillon, K. Azizzadenesheli, Z. C. Lipton, J. Bernstein, J. Kossaifi, A. Khanna, and A. Anandkumar. Stochastic activation pruning for robust adversarial defense. In *the 6th International Conference on Learning Representations, ICLR*, 2018.

- [36] L. Dolberg, Q. Jérôme, J. François, R. State, and T. Engel. RAMSES: revealing android malware through string extraction and selection. In *Proceedings of the 10th International Conference on Security and Privacy in Communication Networks, SecureComm, Revised Selected Papers, Part I*, pages 498–506, 2014.
- [37] M. D. Donno, N. Dragoni, A. Giaretta, and A. Spognardi. DDoS-capable IoT malwares: Comparative analysis and Mirai investigation. *Security and Communication Networks*, 2018:7178164:1–7178164:30, 2018.
- [38] W. Enck, M. Ongtang, and P. D. McDaniel. Understanding android security. *IEEE Security & Privacy*, 7(1):50–57, 2009.
- [39] M. Frustaci, P. Pace, G. Aloï, and G. Fortino. Evaluating critical security issues of the IoT world: Present and future challenges. *IEEE Internet of Things Journal*, 5(4):2483–2495, 2018.
- [40] 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*, pages 45–54, 2013.
- [41] D. Geer. Malicious bots threaten network security. *IEEE Computer*, 38(1):18–20, 2005.
- [42] A. Gerber. Connecting all the things in the Internet of Things. Available at [Online]: <https://ibm.co/2qMx97a>, 2018.
- [43] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems, NIPS*, pages 2672–2680, 2014.
- [44] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *the 3rd International Conference on Learning Representations, ICLR*, 2015.

- [45] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel. Adversarial perturbations against deep neural networks for malware classification. *abs/1606.04435*, 2016.
- [46] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel. Adversarial examples for malware detection. In *22nd European Symposium on Research Computer Security*, pages 62–79, 2017.
- [47] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *the 22nd ACM International Conference on Knowledge Discovery and Data Mining, KDD*, pages 855–864, 2016.
- [48] I. Guyon, B. E. Boser, and V. Vapnik. Automatic capacity tuning of very large vc-dimension classifiers. In *Proceedings of the Advances in Neural Information Processing Systems, NIPS*, pages 147–155, 1992.
- [49] L. H. Chiang, E. L. Russell, and R. Braatz. Fault detection and diagnosis in industrial systems. 12, 2001.
- [50] H. Ham, H. Kim, M. Kim, and M. Choi. Linear SVM-Based android malware detection for reliable IoT services. *Journal of Applied Mathematics*, 2014:594501:1–594501:10, 2014.
- [51] D. Hendler, S. Kels, and A. Rubin. Detecting malicious PowerShell commands using deep neural networks. In *Proceedings of the Asia Conference on Computer and Communications Security, AsiaCCS*, pages 187–197, Incheon, Korea, 2018.
- [52] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant. *Applied logistic regression*, volume 398. 2013.
- [53] 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.

- [54] W. Hu and Y. Tan. Generating adversarial malware examples for black-box attacks based on GAN. *arXiv preprint arXiv:1702.05983*, abs/1702.05983, 2017.
- [55] 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*, pages 611–620, 2009.
- [56] 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.
- [57] D. Kirat, G. Vigna, and C. Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, pages 287–301, San Diego, CA, Aug. 2014.
- [58] E. Knorr. How PayPal beats the bad guys with machine learning, Retrieved, 2015.
- [59] M. Koch. An introduction to Linux-based malware. *SANS Institute InfoSec Reading Room*, 2015.
- [60] 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*, pages 1137–1145, 1995.
- [61] C. Koliass, G. Kambourakis, A. Stavrou, and J. M. Voas. DDoS in the IoT: Mirai and other botnets. *IEEE Computer*, 50(7):80–84, 2017.
- [62] D. Koller and M. Sahami. Toward optimal feature selection. Technical report, Stanford InfoLab, 1996.

- [63] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *The 26th European Signal Processing Conference, EUSIPCO*, pages 533–537, 2018.
- [64] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *the 26th European Signal Processing Conference, EUSIPCO*, pages 533–537, 2018.
- [65] D. Kong and G. Yan. Discriminant malware distance learning on structural information for automated malware classification. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD*, pages 1357–1365, 2013.
- [66] KrebsOnSecurity. Hacked cameras, DVRs powered todays massive internet outage. Available at [Online]: <https://tinyurl.com/zxr36>, 2016.
- [67] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet. Deceiving end-to-end deep learning malware detectors using adversarial examples. In *Workshop on Security in Machine Learning (NIPS)*, 2018.
- [68] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 26th Annual Conference on Neural Information Processing Systems NIPS*, pages 1106–1114, 2012.
- [69] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of Advances in neural information processing systems*, pages 1097–1105, 2012.
- [70] A. Kurakin, I. J. Goodfellow, and S. Bengio. Adversarial examples in the physical world. In *the 5th International Conference on Learning Representations, ICLR*, 2017.

- [71] 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*, pages 1–13, 2012.
- [72] S. Liang and X. Du. Permission-combination-based scheme for Android mobile malware detection. In *Proceedings of the IEEE International Conference on Communications, ICC*, pages 2301–2306, 2014.
- [73] F. Liao, M. Liang, Y. Dong, T. Pang, X. Hu, and J. Zhu. Defense against adversarial attacks using high-level representation guided denoiser. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 1778–1787, 2018.
- [74] 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*, 4(6):1899–1909, 2017.
- [75] X. Liu, J. Zhang, Y. Lin, and H. Li. ATMPA: attacking machine learning-based malware visualization detection methods via adversarial examples. In *International Symposium on Quality of Service, IWQoS*, pages 38:1–38:10, 2019.
- [76] Z. Ma, H. Ge, Y. Liu, M. Zhao, and J. Ma. A combination method for android malware detection based on control flow graphs and machine learning algorithms. *IEEE Access*, 7:21235–21245, 2019.
- [77] J. C. Matherly. Shodan the computer search engine. Available at [Online]: <https://www.shodan.io/>, 2009.
- [78] D. Meng and H. Chen. Magnet: A two-pronged defense against adversarial examples. In *ACM Computer and Communications Security, CCS*, pages 135–147, 2017.

- [79] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff. On detecting adversarial perturbations. In *the 5th International Conference on Learning Representations, ICLR, 2017*.
- [80] 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*, pages 73–84, 2016.
- [81] T. P. Minka. Automatic choice of dimensionality for PCA. In *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS)*, pages 598–604, 2000.
- [82] T. Miyato, A. M. Dai, and I. J. Goodfellow. Adversarial training methods for semi-supervised text classification. In *the 5th International Conference on Learning Representations, ICLR, 2017*.
- [83] A. Mohaisen and O. Alrawi. Unveiling Zeus: automated classification of malware samples. In *Proceedings of the 22nd International World Wide Web Conference, WWW*, pages 829–832, 2013.
- [84] 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*, pages 112–131, 2014.
- [85] A. Mohaisen, O. Alrawi, and M. Mohaisen. AMAL: high-fidelity, behavior-based automated malware analysis and classification. *Computers & Security*, 52:251–266, 2015.
- [86] R. Montella, M. Ruggieri, and S. Kosta. A fast, secure, reliable, and resilient data transfer framework for pervasive IoT applications. In *IEEE Conference on Computer Communications Workshops, INFOCOM Workshops*, pages 710–715, 2018.

- [87] S. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. DeepFool: A simple and accurate method to fool deep neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2574–2582, 2016.
- [88] L. H. Newman. Github survived the biggest DDoS attack ever recorded. Available at [Online]: <https://www.wired.com/story/github-ddos-memcached/>, 2018.
- [89] S. Ni, Q. Qian, and R. Zhang. Malware identification using visualization images and deep learning. *Computers & Security*, 77:871–885, 2018.
- [90] NVD. Nvd vulnerability metrics. Available at [Online]: <https://nvd.nist.gov/vuln-metrics/cvss>, Retrieved, 2018.
- [91] 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*, 24:522–533, 2016.
- [92] N. Papernot, P. D. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against deep learning systems using adversarial examples. [abs/1602.02697](https://arxiv.org/abs/1602.02697), 2016.
- [93] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387, 2016.
- [94] N. Papernot, P. D. McDaniel, A. Sinha, and M. P. Wellman. Sok: Security and privacy in machine learning. In *IEEE European Symposium on Security and Privacy, EuroS&P*, pages 399–414, 2018.

- [95] N. Papernot, P. D. McDaniel, X. Wu, S. Jha, and A. Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE Security and Privacy, SP*, pages 582–597, 2016.
- [96] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas. Malware classification with recurrent networks. In *Proceedings of the 40th IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 1916–1920, 2015.
- [97] S. M. Pontiroli and F. R. Martinez. The Tao of .NET and PowerShell malware analysis. In *Virus Bulletin Conference*, 2015.
- [98] D. Reading. Carna compromise delivers data, but casts suspicions. Available at [Online]: <https://www.darkreading.com/vulnerabilities---threats/carna-compromise-delivers-data-but-casts-suspicious/d/d-id/1139462>, 2013.
- [99] A. Rozsa, M. Günther, and T. E. Boult. Are accuracy and robustness correlated. In *the 15th IEEE International Conference on Machine Learning and Applications, ICMLA*, pages 227–232, 2016.
- [100] G. Rusak, A. Al-Dujaili, and U.-M. O’Reilly. AST-based deep learning for detecting malicious powershell. In *Proceedings of the Conference on Computer and Communications Security, CCS*, pages 2276–2278, 2018.
- [101] P. Samangouei, M. Kabkab, and R. Chellappa. Defense-gan: Protecting classifiers against adversarial attacks using generative models. In *the 6th International Conference on Learning Representations, ICLR*, 2018.

- [102] A.-D. Schmidt, H.-G. Schmidt, J. Clausen, K. A. Yuksel, O. Kiraz, A. Camtepe, and S. Albayrak. Enhancing security of linux-based android devices. In *Proceedings of 15th International Linux Kongress*, 2008.
- [103] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. AVClass: A tool for massive malware labeling. In *Proceedings of 19th International Symposium on Research in Attacks, Intrusions, and Defenses, RAID*, pages 230–253, 2016.
- [104] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. AVclass: A tool for massive malware labeling. In *Processing of the International Symposium on Research in Attacks, Intrusions, and Defenses, RAID*, pages 230–253, 2016.
- [105] 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*, pages 113–120, 2010.
- [106] 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*, pages 2430–2437, 2017.
- [107] 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*, 5(2):1043–1054, 2018.
- [108] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *Proceedings of the 24th USENIX Security Symposium*, pages 611–626, 2015.
- [109] S. Siby, R. R. Maiti, and N. O. Tippenhauer. IoTScanner: Detecting privacy threats in IoT neighborhoods. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, pages 23–30, Dallas, TX, Oct.–Nov. 2017.

- [110] O. Starov, J. Dahse, S. S. Ahmad, T. Holz, and N. Nikiforakis. No honor among thieves: A large-scale analysis of malicious web shells. In *Proceedings of the 25th International Conference on World Wide Web, WWW*, pages 1021–1032, 2016.
- [111] I. Steinwart and A. Christmann. *Support vector machines*. Springer Science & Business Media, 2008.
- [112] J. Su, D. V. Vargas, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai. Lightweight classification of IoT malware based on image recognition. *arXiv preprint arXiv:1802.03714*, 2018.
- [113] 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.
- [114] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations, ICLR*, 2014.
- [115] 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*, pages 1524–1533, 2014.
- [116] Y. Tian, J. Wang, Z. Zhou, and S. Zhou. Cnn-webshell: Malicious web shell detection with convolutional neural network. In *Proceedings of the VI International Conference on Network, Communication and Computing, ICNCC*, pages 75–79, 2017.
- [117] T. D. Tu, C. Guang, G. Xiaojun, and P. Wubin. Webshell detection techniques in web applications. In *Proceedings of the International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–7, 2014.

- [118] H. Uguz. A two-stage feature selection method for text categorization by using information gain, principal component analysis and genetic algorithm. *Knowledge-Based Systems*, 24(7):1024–1032, 2011.
- [119] J. Uitto, S. Rauti, J. Mäkelä, and V. Leppänen. Preventing malicious attacks by diversifying Linux shell commands. In *Proceedings of the 14th Symposium on Programming Languages and Software Tools, SPLST*, pages 206–220, 2015.
- [120] A. Verikas, A. Gelzinis, and M. Bacauskiene. Mining data with random forests: A survey and results of new tests. *Pattern Recognition*, 44(2):330–349, 2011.
- [121] 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.
- [122] R. Wang, A. M. Azab, W. Enck, N. Li, P. Ning, X. Chen, W. Shen, and Y. Cheng. SPOKE: scalable knowledge collection and attack surface analysis of access control policy for security enhanced android. In *Proceedings of the Asia Conference on Computer and Communications Security, AsiaCCS*, pages 612–624, 2017.
- [123] W. Wang, Z. Gao, M. Zhao, Y. Li, J. Liu, and X. Zhang. Droidensemble: Detecting android malicious applications with ensemble of string and structural static features. *IEEE Access*, 6:31798–31807, 2018.
- [124] 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*, pages 1–7, 2018.
- [125] N. Wells. Busybox: A swiss army knife for Linux. *Linux Journal*, 2000(78es):10, 2000.

- [126] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu. DroidMat: Android malware detection through manifest and API calls tracing. In *Proceedings of the Seventh Asia Joint Conference on Information Security, AsiaJCIS*, pages 62–69, 2012.
- [127] 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*, pages 98–118, 2015.
- [128] W. Xu, D. Evans, and Y. Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. In *25th Annual Network and Distributed System Security Symposium, NDSS*, 2018.
- [129] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers. In *the 23rd Network and Distributed System Security Symposium, NDSS*, pages 21–24, 2016.
- [130] 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*, pages 590–604, 2014.
- [131] J. Yan, G. Yan, and D. Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, pages 52–63, 2019.
- [132] K. Zhang, X. Liang, R. Lu, and X. Shen. Sybil attacks and their defenses in the Internet of Things. *IEEE Internet of Things Journal*, 1(5):372–383, 2014.
- [133] 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*, pages 1105–1116, 2014.

- [134] 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.