SHELLCORE: Automating Malicious IoT Software Detection by Using Shell Commands Representation

Hisham Alasmary[‡†], Afsah Anwar[†], Ahmed Abusnaina[†], Mohammad Abuhamad[¶],
An Wang[◇], DaeHun Nyang[§], Amro Awad[⋆], and David Mohaisen[†]
[†]University of Central Florida [‡]King Khalid University
[¶]Loyola University [◇]Case Western Reserve University [§]Ewha Womans University [⋆]NCSU

***Abstract*—The Linux shell is a command-line interpreter that provides users with a command interface to the operating system, allowing them to perform a variety of functions. Although very useful in building capabilities at the edge, the Linux shell can be exploited, giving adversaries a prime opportunity to use them for malicious activities. With access to IoT devices, malware authors can abuse the Linux shell of those devices to propagate infections and launch large-scale attacks, e.g., DDoS. In this work, we provide a first look at shell commands used in Linux-based IoT malware towards detection. We analyze malicious shell commands found in IoT malware and build a neural network-based model, ShellCore, to detect malicious shell commands. Namely, we collected a large dataset of shell commands, including malicious commands extracted from 2,891 IoT malware samples and benign commands collected from real-world network traffic analysis and volunteered data from Linux users. Using conventional machine and deep learning-based approaches trained with term- and character-level features, ShellCore is shown to achieve an accuracy of more than 99% in detecting malicious shell commands and files (*i.e.,* binaries).**

***Index Terms*—Linux Shell Commands, IoT Security, Malware Detection, Machine Learning**

I. INTRODUCTION

INTERNET of Things (IoT) manufacturers and application developers have started to discover the benefits of the edge computing paradigm and do more compute and analytics on the devices themselves. The on-device approaches help reduce latency for critical applications, lower dependence on the cloud, and better manage the massive data generated by the IoT devices. An example of this trend is the Nest Cam IQ indoor security camera [1], which uses on-device vision processing power to watch for motion, distinguish family members, and send alerts. Such a paradigm provides new opportunities for IoT applications [2], [3]. To unleash the power of Linux-based systems, IoT devices at the edge employ shell commands, which would allow invocation of Linux capabilities in a seamless manner. This utilization, which is essential for many edge applications, is sometimes exploited by malicious actors (malactors) to launch malicious activities, and automate the process of attacks and malware proliferation.

This work was supported in part by AFRL (Air Force Research Lab) summer program, in part by NSF under Grant CNS-1809000 and Grant CNS-1814417, in part by NRF (National Research Foundation of Korea) under Grant 2016K1A1A2912757, and in part by Cyber Florida Seed Grant. The work of H. Alasmary and M. Abuhamad was done while the authors were at the University of Central Florida. (*corresponding authors: David Mohaisen; mohaisen@ucf.edu.*).

Indeed, the increasing use of IoT devices for everyday activities has been paralleled with IoT's susceptibility to risks, including major attack vectors, such as vulnerabilities in the hardware and software stacks and the use of default usernames and passwords. Those attack vectors are demonstrated by major high bandwidth Distributed Denial of Service (DDoS) attacks. Targets of those attacks include large companies, such as *Github* [4] and *Dyn* [5]. To launch those attacks, the attackers exploit infected IoT devices for executing a series of commands for malware and attack propagation. Since most IoT and embedded devices use a packed version of software, such as Busybox [6], to implement Linux capabilities, Linux-based shell commands are used for automating those attacks.

The Linux shell as an entry point to IoT devices is accessible to many attacks, including brute-force, privilege escalation, shellshock, and other vulnerabilities (*e.g.*, CVE-2018-9310, CVE-2019-1656, CVE-2018-0183, CVE-2017-6707) [7]–[10]. Using secondary information, such as the listings of IoT and embedded devices on the likes of Shodan [11], adversaries can utilize default passwords to connect to arbitrary devices on the Internet, gain control over them, and use them for their malicious activities through remote access and automation tools. For example, a simple "default password" search on Shodan returns 72,763 results, which all can be accessed, and used for attacks.

Shell commands are heavily utilized in IoT malware and botnet operation. Malware-infected hosts use Command and Control (C2) servers to obtain payloads that include instructions to compromised machines (or bots). Such instructions aim to synchronize actions and cycles of activities to attack targets and propagate the recruitment of new bots that eventually become a source of propagation. In this example, bots use the shell to execute *chmod* command to change privileges. Moreover, bots also 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].

Significance. Detecting malicious shell commands to harden the security of a device is of paramount importance. While the prior works have studied the malicious use of Windows *PowerShell*, the malicious use of the Linux shell for attack

automation in IoT devices is not fully-investigated. This work aims to study shell commands that appear in the static analysis of IoT malware binaries, and understand their intrinsic features towards their detection. It is important to note that there has been some work on understanding shell commands and their use by malicious software in the literature. However, the majority of the prior work has focused on other shell interpreters (*e.g.* power and web), and the emergence of Linux-based IoT malware that heavily uses shell commands makes the detection of shell commands associated with malicious IoT software of paramount importance.

Our Approach. To address this threat, in this work we design, implement, and evaluate ShellCore, a system for detecting malicious shell commands used in IoT malware. To evaluate ShellCore, we collect a dataset of residual shell commands from IoT malware samples. Our preliminary analysis shows that shell commands can be found embedded in the disassembled code of malware binaries. Therefore, we employ static analysis to search through the disassembled code of malware to extract the shell used in the 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-world environment. For analyzing and detecting malicious commands, ShellCore employs a Natural Language Processing (NLP) approach for feature generation, followed by deep learning-based modeling for detecting the malicious commands.

Contributions. This work aims to utilize static analysis to detect the malicious use of shell commands in IoT binaries, and to use them as a modality for IoT malware detection. As such, we make two broad contributions. **C-1:** Using shell commands extracted from 2,891 recent IoT malware samples along with a benign dataset, we design a detection system that can detect malicious shell commands with an accuracy of more than 99%. Compared to the state-of-the-art approaches, our system is more efficient and accurate. Using term- and character-level features, the feature space on the shell commands is easy to explain and interpret. Features contributing to malicious behaviors can be easily identified so that shell commands could be restricted to legitimate use. **C-2:** 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 an accuracy of more than 99%. Our detection approach can be applied to files compiled for any processor architecture (*e.g.* ARM, MIPS, Power PC, etc.) as long as the shell commands are extracted, which can be done efficiently.

Organization. The rest of this paper is organized as follows. In section II, we present the problem statement and a high-level overview of our approach. In section III, we review our approach in details; the feature extraction respecting various specifics of the application domain, learning algorithms, and representations. In section IV, we review the evaluation of our approach; heuristics developed for extracting shell commands from malicious binaries and benign use contexts, evaluation metrics and settings, and results. In section V we review the related work, and draw concluding remarks in section VI.

II. Problem Statement and Approach Overview

In this section, we begin by the problem statement and a high-level overview of our approach.

A. Problem Statement

The problem we tackle in this paper is an offline malware classification using shell commands. By offline we mean that the detector operates in a postmortem mode, where a dataset is provided and the detector is tasked with identifying whether each binary in this dataset is malicious or benign. Given the modality of the analysis of interest, we are also interested in determining whether a given shell command extracted from a binary or a use context is malicious or benign. We approach this problem systematically by modeling shell commands that appear in the residual artifacts of IoT malware binaries.

The shell command classification problem is formally defined as follows. First, let $\{x_i, y_i\}_{i=1}^{N}$ be a training set, where $x_i \in \mathbb{R}^d, y_i \in \{0, 1\}$; that is, x_i is a feature representation of a shell command c_i, where the representation has d real-valued features, and y_i is the corresponding label of "zero" if x_i is a benign shell command, and "one" otherwise. The classification problem of shell commands is formulated as finding a set of parameters that make up a function f such that $f(x_i) = \hat{y}_i$ where $||y_i - \hat{y}_i||$ for all i is minimized (i.e., minimal prediction error). The transformation of c_i into x_i is called feature extraction, denoted by $\Phi(c_i) = x_i$, and is a central contribution of this work through character- and word-level representations. We use those two approaches for their prevalence in representing text and text-like data, which is the case of shell commands.

The malware classification problem is defined as an extension of the shell command-level classification problem. For that, we use a combined set of shell commands associated with each malware sample as a representation to conduct malware classification (or detection). The same definition above is extended to malware s_i, where s_i is a collection of shell commands c_i^j, for $j = 1, \ldots, k$, where x^j is the corresponding feature representation of the malware sample s_i. Note that the same function Φ can be extended for the feature representation (e.g., the features associated with the different commands extracted from the same binary sample can be stacked to represent the binary). Similarly, the function f is defined for the binary-level from the command-level classification.

B. A High-level Overview of Our Approach

The shell is a single point of entry for malware to launch attacks. As such, 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 the malicious commands will help mitigate such exploits. Our analysis highlights the use of shell commands for infection, propagation, and attack by malware. The Linux capabilities of embedded IoT devices give adversaries the required power to abuse the shell.

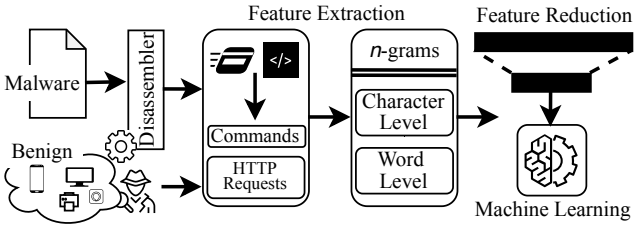


Fig. 1. The workflow of ShellCore, highlighting the sources of data and its division by class (malicious or benign). The raw data is preprocessed to extract shell commands. The shell commands are represented as (1) characters and (2) words, which are fed to learning networks for detection.

Objectives. The main objective of ShellCore is to effectively detect malicious IoT binaries (files) based on their usage of the shell commands. Upon detecting individual malicious shell commands (i.e., shell commands associated with malware samples), it will be natural to extend the detection to malicious binaries (files) as a whole. Thus, we break down the problem into two parts – 1) detecting malicious commands and 2) detecting malicious files.

High-Level Design. Our design operates on various binaries of malicious and benign IoT programs. The key idea of ShellCore is to employ static program analysis tools to extract meaningful representations that can be used eventually to distinguish between benign and malicious binaries. To do so, we start with (potential) IoT malware samples and disassemble them to extract shell commands. We establish various heuristics for extracting those commands, and we outline those heuristics in section IV. We repeat the process for (potentially) benign samples as well to explore the power of our representation for malware detection. To make the processing of these commands computationally tractable, we embed those commands into a representation space by extracting term- and character-level feature representations from them using the bag-of-words technique, which is commonly used in NLP tasks. Along with the bag-of-words, we use the n-grams to represent the commands as feature vectors. Given that those representations may result in high dimensional data representation, we employ the Principal Component Analysis (PCA) for feature reduction before implementing the classification over commands (see section II-A for problem statement).

Upon representing the malicious and benign commands as feature vectors, ShellCore aims to detect malicious commands, as shown in Figure 1. To do so, ShellCore employs machine learning algorithms to classify commands. We use both simple and more advanced (deep) learning approaches. For evaluation, we use cross-validation to address bias and to ensure the generalization capabilities of the model. Using the same model architecture, we extend the detection system to detect malicious IoT binaries. To do so, we group the commands by each malware sample and benign application in one single set that is represented as one feature vector to be classified.

III. Our Detection System: ShellCore

The core of our detection system is a deep learning model built on top of NLP-based features. To better help learn the specifics of shell commands, we tune the default NLP algorithms to enrich the feature representations of the commands. We represent the commands as feature vectors using the bag-of-words approach. Then, we reduce the feature space using PCA. ML-based algorithms are then used for malicious command and sample detection. In the following, we review the technical details of the feature extraction, and classification methods of our detection system.

A. Feature Extraction and Reduction

The feature extraction process aims to present the attributes of samples, by cleansing and linking the data and transforming it into a format that is easier to process 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 feature extraction depending upon the nature of the data. Considering the textual nature of our samples, we focus on text-based representation methods. Towards this, we leverage the term-level NLP-based approach by considering words in the samples as features. Additionally, since such an approach misses very crucial attributes, we then employed a character-level NLP approach to meet our goals.

1) Term-level 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 $I_1 = I_{11}, I_{12}, I_{13}, ..., I_{1N}$.

2) Character-level NLP-based Model: The term-level NLP-based approach does not take the operational symbols, such as the logical operators, in a command into consideration, 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 $l \leq 2$ call for building a more accommodating feature generation mechanism. To do so, we changed the boundaries of the definition of a word by considering every space, special characters, alphabets, and numbers as words, along with the 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 $I_2 = I_{21}, I_{22}, I_{23}, ..., I_{2N}$.

3) Feature Representation: To represent every element in the dataset from a defined reference point, they are represented with respect to axes in space. In particular, every command/sample in the dataset is represented as a feature

vector in the defined feature space. 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 the commands, such as including the words in the commands and 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 implemented the bag-of-words method to define our feature space. The rest of this section explains our feature representation mechanism.

4) Bag-of-Words as Command Embedding: We generate 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 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 elements in the dataset, where the vector has an index for every word in the vocabulary— The final vector is represented as the occurrence of each word from the vocabulary in a given command (*i.e.,multi-hot encoding*).

Specifically, to generate the vector space, we add every word to an array. For every sample, we initialize its feature vector with a size equals 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.

5) Encoding Syntax: An important characteristic of the commands is their syntax. This syntax depends 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 commands are considered as 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 a separate bag of words, such that, the bag contains all the words I^k_{1i}, 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 term-level NLP model, can be combined as $I^1_{11}, I^2_{11}, I^3_{11}, ...I^1_{12}, I^1_{13}, ...I^m_{1N}$ Let B be the bag of word for the dataset, such that $B = B_1, B_2, B_3, ..., B_t$, 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 be represented as the frequency of occurrence, of the t^{th} word in the bag, in the command. $F = f_{B_1}, f_{B_2}, f_{B_3}, ..., f_{B_t}$, 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.

6) Feature Reduction: We capture as many features as possible to achieve accurate results. However, beyond a certain point, the model may suffer from the curse of dimensionality, which causes 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 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 the aim of addressing the two problems. We implement PCA for feature reduction to improve the performance and the quality of our classifier of ShellCore, where the PCA features (components) are extracted from the raw features. PCA itself 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 (thus the quality improvement). PCA normalizes the dataset by transforming them into a normal distribution with the same standard deviation [13], resulting in a standard representation of variables in order to identify a subset that can best characterize the underlying data [14].

We reduce the d-dimensional vector representation of commands to q number of principal components onto which the retained variance under projection is maximal.

B. Classification Methods

After representing each sample as a feature vector, we classify them into malicious and benign by leveraging the ML-based algorithms.

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, $w_1 + w_2 + w_3 + ...w_q$, which passes through a ReLU activation function ($y(x) = max(x, 0)$). The output of the hidden layers is then fed to the output layer, and passed to a sigmoid activation function h, defined as $h(x) = \frac{1}{1+e^{-x}}$, 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 sample, the classifier computes the margin and selects the hyperplane with the smallest margin (distance) from the sample as the output class [15]. We use SVM due to its

effectiveness in high dimensional spaces. To achieve the goal, we utilize the following decision function [16], [17] $sgn(\sum_{i=1}^{n} y_i \alpha_i K(x_i, x) + \rho)$ where x_i, $i \in [1, q]$, is the 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, $K(x_i, x) = \phi(x_i)^T \phi(x_j)$.

C. Term- and Character-level NLP-based Approaches

1) Term-level NLP-based Model: The term-level 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. 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 as is shown in Table IV. The results are shown for both SVM and DNN-based classifiers, with DNN-based classifier yielding better results.

2) Character-level NLP-based model: We note that the term-level considers the 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 of fewer than three characters, including *cd* and *ls*, or consist of special characters, such as `||` and `&&`. To address the shortcoming, we create the feature generation step considering 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 contains every letter, number, and character represented as an individual feature.

IV. Evaluation and Discussion

In this section, we evaluate ShellCore's performance. 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.

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

In addition to 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 the feature representations preserve 99% of the variance in the training dataset.

To set out, we begin by describing the process of assembling the dataset used in this evaluation. We obtain our shell commands by statically disassembling the malware binaries and extracting shell command strings (following some regular expression rules).

TABLE I
Malware dataset by architecture. Percentage is out of the total samples.

Architecture	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%

A. Malicious Dataset and Commands Extraction

We obtain a dataset of 2,891 randomly selected IoT malware samples from the IoTPOT project [18], a honeypot emulating IoT devices. IoTPOT emulates services, such as telnet and other vulnerable services including those of specific devices with distributed proxy sensors in several countries. Table I depicts the malware distribution according to their architectures and their percentage. Figure 1 shows our approach, end-to-end, split into three modules: initial discovery, command extraction, and detection. Our data collection is in the first two modules. In the following, we outline the steps we have taken in order to obtain the shell commands from the malware samples (binaries).

In the initial discovery module, we disassemble the malware binaries. To create a set of rules that automatically apply to samples for retrieving the relevant commands, we manually examine all shell commands extracted from the strings of **18 malware samples** and establish patterns of those commands. We then use them to automate the extraction of shell commands for the rest of the malware samples.

The second component in our workflow is a command extraction module, which takes the command patterns obtained in the initial discovery phase and applies those patterns to the strings of each sample. As a result, we extract the shell commands from the malicious binary samples, by concentrating on the *strings* only, and label them as malicious.

Commands Extraction. Using *Radare2*, an open-source static analysis tool with an API for automation, we first disassemble each malware binary in our 2,891 samples and extract the *strings* from the disassembled code. We then use the strings appearing in each sample to obtain the shell commands in them, creating our malicious commands. For coverage, we gather all *strings* from the disassembled code. For a faster extraction of the shell commands, we calculate the offset, or memory address where the string is referenced in the disassembled code, then conduct the disassembly from that offset. We pull the instruction set at the offset and extract the desired command. Before automating the command extraction, we manually analyze the 18 samples to observe patterns that could uniquely identify the shell commands.

From these 18 malware samples, we identify 1,273 patterns and use them to extract the shell commands from other

samples. For example, strings beginning with shell command keywords, such as *cd* , between *if* and *fi*, *kill*, *wait*, *disown*, *suspend*, *fc*, *history*, *break*, among other similar command structures, are extracted. For coverage of those patterns, we use online resources to build a dataset of the keywords of shell commands to augment our automation process.

Based on the identified patterns, we use regular expressions to search for the specific patterns in the *strings* obtained from the malware to automate the process for all malware samples. Although the commands contained in the strings may not be syntactically correct, *e.g.* spaces are masked with special characters or spaces, they, however, hint to the location of shell command references. Therefore, we navigate to the address where a particular string is quoted and disassemble at that offset. This analysis generate a total of 2,008 unique malicious commands.

B. Benign Dataset and Commands Extraction

To evaluate ShellCore, acquiring a benign shell commands dataset is a necessary step, although a challenging task for multiple reasons. For example, while Linux-based applications are ubiquitous, extracting the corresponding shell commands and using them as a baseline for our benign dataset might be only partially representative, since these binaries may not be necessarily intended for embedded devices.

Another approach to collect benign shell commands is by observing shell access and their usage by benign users, which requires monitoring network traffic to "sniff" the shell commands by benign users. However, we notice that a majority of the traffic nowadays is carried over HTTPS, the encryption limits our visibility into those benign shell commands.

To cope with these shortcomings, we rely on volunteers for providing their usage of shell commands as a representative of benign usage. In order to do so, we conduct collection efforts at both the host and network levels. At the host-side, we gather the bash history data from nine volunteer users. To protect the users' privacy, we anonymize their identities by manually observing the commands and removing every clearly identifying information, such as usernames, domain names and IP addresses, in a consistent manner. In total, we collect a dataset of about 143 MB from these volunteers, consisting of 5,772 commands. The collected commands correspond to services, such as *ssh*, *git*, *apt*, *Makefile*, and *curl*, among others, and generic Linux commands, such as *cd*, *rm*, *chmod*, *cp*, and *find*, among others.

For the network-side profiling, we rely on high-level network traffic monitoring from **two networks** to obtain network-level artifacts (*e.g.* GET, POST, etc.) that are not part of an encrypted payload. 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 and TFTP, with the intent to distribute malicious payloads and scripts, we attempt to monitor those protocols in benign use setup for benign data collection. As such, we built our benign command collection framework with two separate networks, as highlighted in Figure 2.

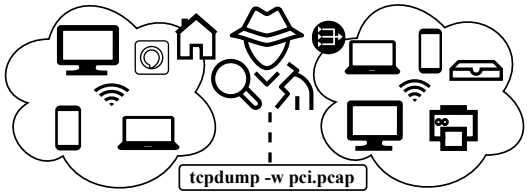


Fig. 2. Monitoring stations for creation of benign dataset creation. Two network implementations are used: NAT, and a home network.

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 the home network. Our home network in this experimental setup consists of two 64-bit Linux devices, one Amazon Alexa, one iPhone device, one Mac device with a voice assistant, Siri, which is continuously used, and a router. Figure 2 is a high-level illustration of our benign data collection system. In the first network (right), 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.

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 as the devices are added gradually 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 gather a dataset of approximately 34 GB from the first network and approximately 1 GB from the second network.

The traffic gathered from the five volunteers (with consent) in the first network (Network 1) result in a total of 28,578,754 individual payloads, and only 1,625,143 of them are not encrypted, which we utilize for our benign dataset. From the second network (Network 2), five sources generate 4,735 unencrypted payloads in total, which we use as part of our dataset. In total, our benign dataset consists of three parts, *bash* (5,772 commands), *network 1* (1,625,143 commands), and *network 2* (4,755 commands).

Table II shows samples of the payloads from the four data sources. We analyze the samples to find the architecture for which they are compiled using the Linux *File* command.

C. Evaluation Settings and Metrics

To evaluate ShellCore, we use the dataset highlighted in IV-A and IV-B. In the following, we review settings, parameters tuning, validation technique, and evaluation metrics.

1) Dataset: Table III shows the number of commands as well as the commands' length statistics (maximum, minimum, average, median, and standard deviation). We notice that commands in Network 1 have similar lengths, as indicated with

TABLE II
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	Commands	Example
PCAP Net. 1	5	1,625,143	GET /update-delta/hfnkpimlhhgieaddgfemjhofmfblmnib/5092/5091/193cb84a 0e51a5f0ca68712ad3c7fddd65bb2d6a60619d89575bb263fc5dec26.crxd HTTP/1. 1\r\nHost: storage. googleapis.com\r\nConnection: keep-alive\r\nUser-Agent: Mozilla/5.0 (X11; Linux x86˙64) AppleWebKit/537.36 (KHTML, like Gecko) C hrome/72.0.3626.121 Safari/537.36 \r\nAccept-Encoding: gzip, deflate\r\n
PCAP Net. 2	5	4,735	GET /favicon.ico HTTP/1.1\r\nConnection: close\r\nUser-Agent: Mozilla/5.0 (compatible; Nmap Scripting Engine; https://nmap.org/book/nse.html)\r\n Host: 192.168.2.1\r\n
Bash cmd.	9	5,772	sudo wget https://download.oracle.com/otn-pub/java/jdk/8u201-b09/4297048 7e3af4f5a a5bca3f542482c60/jdk-8u201-linux-x64.tar.gz
Malware	2,891	2,008	GET /cdn-cgi/l/chk˙captcha?id=%s & g-recaptcha-response=%s HTTP/1.1 User-Agent: %s Host: %s Accept: */ Referer: http://%s/ Upgrade-Insecure-Requests: 1 Connection: keep-alive Pragma: no-cache Cache-Control: no-cache

TABLE III
SIZE CHARACTERISTICS OF THE DIFFERENT DATASETS. LEN. STANDS FOR LENGTH.

Dataset	Commands	Command Length Statistics				
		Maximum	Minimum	Average	Median	Standard deviation
Network 1	1,625,143	1,564	52	184.68	185	4.88
Network 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

the low deviation. We notice that Network 2 (corresponding to the IoT devices setting) and Malware datasets have the closest lengths overall, per the average and standard deviation characteristics of their distributions.

2) Parameters Tuning: For a better features representation, 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 using five hidden layers.

3) K-Fold Cross-Validation: To generalize the evaluation, cross-validation is used. For K-fold cross-validation, the data are sampled into K subsets, where 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 use 10 for K.

4) Evaluation Metrics: For a class C_i, (where $i \in \{0, 1\}$), False Positive (FP), False Negative (FN), True Positive (TP), and True Negative (TN) are defined as:

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

We used the Accuracy (AC), False-Negative Rate (FNR), and False-Positive Rate as evaluation metrics, which are defined as follows:

- AC = (TP+TN)/(TP+TN+FP+FN),
- FNR = FN/(TP+FN),
- and FPR = FP/(FP + TN).

We report the metrics as mean AC, mean FNR, and mean FPR for the ten folds.

TABLE IV
EVALUATION RESULTS OF MALICIOUS COMMANDS DETECTION. T- AND C-LEVEL STAND FOR TERM- AND CHARACTER-LEVEL, RESPECTIVELY.

Metric	Command Detection			Malware Detection	
	T-level		C-level	T-level	C-level
	SVM	DNN	DNN	DNN	DNN
AC	0.929	0.990	0.998	0.979	0.998
FNR	0.032	0.020	0.001	0.005	0.002
FPR	0	0.001	0.001	0.050	0.001

D. Detecting Malicious Commands

We use ShellCore to detect individual malicious commands. We first present the results of the term-level model, followed by the character-level NLP model. When using DNN, the term-level model provides an accuracy of 99.0% along with an FNR of only 0.1% and FPR of 2.0% as shown in Table IV; SVM, however, provides an accuracy of 92.9%. Given that the DNN-based model performs better, we select the DNN-based model as the classifier for our following evaluations. We then test the performance of the character-level NLP-based model for detecting individual malicious commands over the same dataset. As shown in Table IV, the approach improves the performance of the model, where the accuracy increases from 99.0% to 99.8% and the FNR improves from 2.0% to 0.1%.

The difference in the evaluation results of the two models is attributed to the difference in the underneath learning algorithms of the two models, and hint at the importance of special characters and letters with length less than three. Figure 3 plots the accuracy of malicious command and files detection using both character- and term-level feature representations using the DNN-based model with 10-fold cross-validation. As shown, the accuracy is in the range of 96.1% to 99.8% in every fold when using the term-level model for the training phase, while

the accuracy for the testing phase is in the range of 96.0% to 99.9%, with an average accuracy of 99.0%. For the character-level model, however, the testing accuracy is noticeably higher, between 99.6% and 99.9%. Overall, ShellCore achieves an average of 99.85% accuracy in detecting malicious commands, highlighting the effectiveness of our model.

E. Malware Detection

To generalize from the shell command detection to binaries (malware) detection, 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.

1) Dataset: For this task, we generate benign samples, drawn from benign commands randomly selected to follow similar command-frequency distribution as the malicious samples. We first generate the command-frequency distribution, *i.e.,* defined the distribution of number of commands per sample, of the real-world malicious samples in our dataset. Then by using the sampling techniques, we generate a statistically similar (size-wise) dataset of benign samples that fall in the same size as the malicious samples.

2) Model Training and Detection Performance: Subsequently, we train and test the model over the file specific dataset. In doing so, the commands corresponding to a file are represented as a feature vector of that file. Similar to the individual commands detection, as shown in Table IV, we try both the term-level and character-level NLP-based approaches, with the character-level model yielding a higher detection rate of 99.8% with 0.2% and 0.1% of FNR and FPR, respectively. Compared to the term-level model, the character-level model performs better and improves the accuracy by ≈2% and also reduces the FPR and the FNR. This reflects the improved feature representation technique and also emphasizes the importance of special characters. Figure 3 presents the performance results of ShellCore for malware detection, where it demonstrates that ShellCore can detect malicious commands with high accuracy and very low error (false positive and false negative) rate. Moreover, the same table shows that the accuracy of ShellCore is improved when using the character-level NLP-based model.

F. Discussion

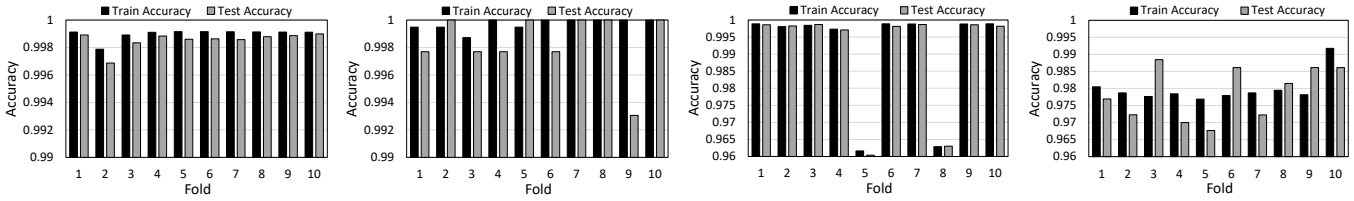
1) The Use of Shell as a Weapon: Shell is a command-line interpreter providing a command-line interface for operating systems by executing a particular command from the terminal to perform specific tasks by calling the appropriate OS command. It abstracts the details of the communication between the application and the operating system. 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 and 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 [19].

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, has been reported [7]. 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.

2) 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 [20], and installing ELF executables on Android systems [21]. 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. We propose a system to detect malicious commands with 99.8% accuracy.

3) Malware Detection: Many efforts have been dedicated to address the security threats to IoT from the hardware, the software and the application perspectives. Some also argue that there is a need for a cross-layer approach for comprehensive protection of the IoT systems [22]. Meanwhile, 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 V 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 help 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 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 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.

4) Limitations: In this study, we analyze the IoT malware statically to extract shell commands from the malware



(a) C-level command detection. (b) C-level malware detection. (c) T-level command detection. (d) T-level malware detection.

Fig. 3. Accuracy of detecting malicious commands using 10 fold cross-validation. T- and C-level stand for Term- and Character-level, respectively.

disassembly. Thus, our approach is limited to malware that do not employ obfuscation. Prior studies have shown that obfuscation is still uncommon among the IoT malware [23], making our model applicable under existing circumstances. Additionally, prior studies have also shown the existence of standard packers, such as UPX [23], [24]. Their unpack module can thus be leveraged to extract the malware binary; our model can then be used to detect malicious software.

5) Applications: DETECTS MALICIOUS SOFTWARE BY THE COMMANDS THEY EXECUTE ON THE SYSTEM. GIVEN THE INCREASE IN IOT MALWARE ATTACKS, THEIR USE OF SHELL COMMANDS CAN BE USEFUL IN DETECTING THE INTENT OF THE MALWARE. ADDITIONALLY, CAN BE LEVERAGED TO DETECT FILELESS ATTACKS []. SUCH ATTACKS MAKE USE OF THE DEVICE'S TERMINAL TO EXECUTE SUCCESSIVE COMMANDS TOWARDS THEIR MALICIOUS INTENT. AS A FILE IS UNAVAILABLE FOR ANALYSIS, THE EXECUTION OF COMMANDS CAN BE USED AS A MODALITY TOWARDS THEIR DETECTION.

V. Related Work

A summary of the related work is in Table V. Broadly, there have been some work on *PowerShell* and *Web Shell* commands detection, as well as IoT malware detection, which are related to this work. No prior work exists on IoT shell commands.

1) Shell Commands: Hendler *et al.* [28] detected malicious *PowerShell* commands using several machine learning approaches, *e.g.* NLP and Conventional Neural Network (CNN). Both studies have focused on shell commands that can only run on Microsoft Windows, *i.e.,* handling binaries of a single architecture, with very little insight of whether the approach can be applied to IoT software and command artifacts. Additionally, Uitto *et al.* [10] proposed a command diversification technique, by modifying and extending commands, to protect against injection attacks. Further, Anwar *et al.* [31] statically analyze the IoT malware and specify about the presence of shell commands in their disassembly. They use them along with other features, such as, strings, Control Flow Graphs (CFG) towards malware detection.

2) Web Shell: Web shell is a script that allows an adversary to run on a targeted web server remotely as an administrator. Starov *et al.* [25] statically and dynamically analyzed a set of web shells to uncover features of malicious hypertext preprocessor shells. Tian *et al.* [26] proposed a system to detect malicious web shell commands using CNN and word2vec-based approaches. In a similar context, Rusak *et al.* [27] proposed a deep learning approach to classify malicious *PowerShell* by families using the abstract syntax trees representation of the *PowerShell* commands. Li *et al.* [29] propose an ML model to detect malicious web shells written in PHP, achieving an accuracy 91.7%. Moreover, Stokes *et al.* [30] employ a recurrent deep learning model to detect malicious VBScripts by using a dataset of first 1000-bytes of 240,504 VBScript files and achieving a TPR of 69.3% and an FPR of 1.0%.

3) IoT Malware Detection: IoT malware has been on the rise and has received the attention of researchers which is evident by the growing body of work in this domain. Pa *et al.* [18] proposed IoTPOT, a detection system that supports different malware architectures to analyze and detect Telnet-based attacks on IoT devices. [Dang*et al.* [32] deployed four IoT honeypots to study the recent fileless attacks launched by Linux-based IoT devices. These attacks do not rely on the malware files and leave no footprint, so, they found that 99.7% of fileless attacks use shell commands, making ShellCore is more relevant in detecting this types of attack. However, recent works have focused on detecting IoT malware network traffic, such as, IoT network packets [33], [34].] Bertino and Islam [35] proposed a behavior-based approach that combines behavioral artifacts and external threat indicators for malware detection. The approach, however, relies on external online threat intelligence feeds (*e.g.* VirusTotal) and cannot be generalized to other than home network environments (due to computations offloading). On the other hand, Hossain *et al.* [36] proposed Probe-IoT, a forensic system that investigates IoT-related malicious activities. Similarly, Montella *et al.* [37] 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. Cozzi *et al.* [38] analyzed a large Linux malware dataset by studying their behavior, and discussed obfuscation techniques that malware authors use. Furthermore, Alasmary *et al.* [39] and Anwar *et al.* [31] use the different artifacts of the IoT malware, such as, CFGs, strings, and functions, to build detection systems. Taking this forward, Abusnaina *et al.* examined the robustness of CFG-based IoT malware detection models to adversarial attacks [40] and also proposed effective defenses [41]. Recent arts have also focused on exploring the IoT network environment. Choi *et al.* [42] explored the presence of endpoints the disassembly of the IoT malware binaries towards characterizing IoT malware spread and affinities. This emphasis on the network has also enabled the monitoring and detection of anomalies and vulnerabilities in wireless communication and network traffics [43]–[45].

TABLE V
COMPARISON WITH RELATED WORK. AUC: AREA UNDER THE CURVE, TPR: TRUE POSITIVE RATE, TNR: TRUE NEGATIVE RATE, AC: ACCURACY, FNR: FALSE NEGATIVE RATE, FPR: FALSE POSITIVE RATE, NLP: NATURAL LANGUAGE PROCESSING, CNN: CONVOLUTIONAL NEURAL NETWORKS, MS: MALWARE SIGNATURE,MF: MALWARE FUNCTIONS, LW: LONGEST WORD IN FILES HEADER, DL: DEEP LEARNING, MLP: MULTI-LAYER PERCEPTRON, SVM: SUPPORT VECTOR MACHINE, GBT: GRADIENT BOOSTED TREE, LSTM: LONG SHORT TERM MEMORY, SDA: STATIC AND DYNAMIC ANALYSIS, PR.: PRECISION, RE.: RECALL, AND F1: F1-SCORE. NOTE THAT OUR SYSTEM IS CAPABLE OF CLASSIFYING BOTH SHELL COMMANDS AND HOSTING MALWARE. *THE LAST ROW DEMONSTRATES THE RESULTS OF OUR SYSTEM IN CLASSIFYING MALWARE SAMPLES USING THEIR SHELL COMMANDS.

Study	Shell Type	Dataset	Capability	Performance (Best Result)	Method
Starov *et al.* [25]	Web shell	481	Analysis	—	SDA
Uitto *et al.* [10]	Linux shell	13,257	Analysis	—	Diversification
Tian *et al.* [26]	Web shell	7,681	Detection	Pr. (98.6%), Re. (98.6%), F1 (98.6%)	CNN
Rusak *et al.* [27]	PowerShell	4,079	Detection	AC (85%)	DL
Hendler *et al.* [28]	PowerShell	66,388	Detection	AUC (98.5-99%), TPR (0.24-0.99%)	NLP, CNN
Li *et al.* [29]	PHP web shell	950	Detection	AUC (98.7%), AC (91.7%), FPR (1.0%)	RF, SVM, GBT
Stokes *et al.* [30]	VBScripts	240,504	Detection	TPR (69.3%), FPR (1.0%)	LSTM, CNN
Ours (Command-level)	Linux shell	190,897	Detection	AC (99.89%), FNR (0.08%), FPR (0.13%)	DNN, SVM
Ours (Binary-level)	Linux shell	2,891*	Detection	AC (99.83%), FNR (0.13%), FPR (0.20%)	DNN, SVM

VI. CONCLUSION

We proposed ShellCore, a machine learning-based approach to detect shell commands used in IoT malware. We analyze 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 NLP-based approaches for feature creation. ShellCore detects individual malicious commands and malware with an accuracy of more than 99%, 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.

REFERENCES

[1] Google. (2017) Nest cam iq indoor: State-of-the-art smart. Available at [Online]: https://tinyurl.com/yatod9zp.

[2] R. Wolski, C. Krintz, F. Bakir, G. George, and W.-T. Lin, "Cspot: portable, multi-scale functions-as-a-service for iot," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 236–249.

[3] S. Y. Jang, Y. Lee, B. Shin, and D. Lee, "Application-aware iot camera virtualization for video analytics edge computing," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2018, pp. 132–144.

[4] L. H. Newman. (2018) Github survived the biggest DDoS attack ever recorded. Available at [Online] : https://www.wired.com/story/github-ddos-memcached/.

[5] KrebsOnSecurity. (2016) Hacked cameras, DVRs powered todays massive internet outage. Available at [Online]: https://tinyurl.com/zxrfm36.

[6] N. Wells, "Busybox: A swiss army knife for Linux," *Linux Journal*, vol. 2000, no. 78es, p. 10, 2000.

[7] NVD. (Retrieved, 2018) Nvd vulnerability metrics. Available at [Online] : https://nvd.nist.gov/vuln-metrics/cvss.

[8] Developers. (Retrieved, 2010) Cve-2010-4258: Turning denial-of-service into privilege escalation. Available at [Online]: https://tinyurl.com/y8ex6ltj.

[9] 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*, 2011, p. 5.

[10] 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*, 2015, pp. 206–220.

[11] J. C. Matherly, "Shodan the computer search engine," Retrieved, 2009.

[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*, Vancouver, BC, Canada, Aug. 2017, pp. 1093–1110.

[13] L. H. Chiang, E. L. Russell, and R. Braatz, "Fault detection and diagnosis in industrial systems," vol. 12, 2001.

[14] H. Uguz, "A two-stage feature selection method for text categorization by using information gain, principal component analysis and genetic algorithm," *Knowledge-Based Systems*, vol. 24, no. 7, 2011.

[15] I. Steinwart and A. Christmann, *Support vector machines*. Springer Science & Business Media, 2008.

[16] 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*, 1992, pp. 147–155.

[17] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[18] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "IoTPOT: A novel honeypot for revealing current IoT threats," *Journal of Information Processing*, vol. 24, pp. 522–533, 2016.

[19] M. Koch, "An introduction to Linux-based malware," *SANS Institute InfoSec Reading Room*, 2015.

[20] D. Geer, "Malicious bots threaten network security," *IEEE Computer*, vol. 38, no. 1, pp. 18–20, 2005.

[21] 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.

[22] A. Wang, A. Mohaisen, and S. Chen, "Xlf: A cross-layer framework to secure the internet of things (iot)," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1830–1839.

[23] E. Cozzi, P.-A. Vervier, M. Dell'Amico, Y. Shen, L. Bilge, and D. Balzarotti, "The tangled genealogy of iot malware," in *Annual Computer Security Applications Conference*, 2020, pp. 1–16.

[24] UPX: the Ultimate Packer for eXecutables. Available at [Online]: https://upx.github.io/.

[25] 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*, 2016, pp. 1021–1032.

[26] 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*, 2017, pp. 75–79.

[27] 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*, 2018, pp. 2276–2278.

[28] 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*, Incheon, Korea, 2018, pp. 187–197.

[29] Y. Li, J. Huang, A. Ikusan, M. Mitchell, J. Zhang, and R. Dai, "Shellbreaker: Automatically detecting php-based malicious web shells," *Computers & Security*, vol. 87, p. 101595, 2019.

[30] J. W. Stokes, R. Agrawal, and G. McDonald, "Detection of malicious vbscript using static and dynamic analysis with recurrent deep learning," in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020, pp. 2887–2891.

[31] A. Anwar, H. Alasmary, J. Park, A. Wang, S. Chen, and D. Mohaisen, "Statically dissecting internet of things malware: Analysis, characterization, and detection," in *International Conference on Information and Communications Security, ICICS*. Springer, 2020, pp. 443–461.

[32] F. Dang, Z. Li, Y. Liu, E. Zhai, Q. A. Chen, T. Xu, Y. Chen, and J. Yang, "Understanding fileless attacks on linux-based iot devices with honeycloud," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys*. ACM, 2019, pp. 482–493.

[33] C. D. McDermott, F. Majdani, and A. Petrovski, "Botnet detection in the internet of things using deep learning approaches," in *International Joint Conference on Neural Networks, IJCNN*. IEEE, 2018, pp. 1–8.

[34] A. Kumar and T. J. Lim, "EDIMA: early detection of iot malware network activity using machine learning techniques," in *5th IEEE World Forum on Internet of Things, WF-IoT*. IEEE, 2019, pp. 289–294.

[35] E. Bertino and N. Islam, "Botnets and Internet of Things security," *IEEE Computer*, vol. 50, no. 2, pp. 76–79, 2017.

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

[37] 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* , 2018, pp. 710–715.

[38] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding Linux malware," in *IEEE Symposium on Security & Privacy, S&P*, 2018.

[39] 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.

[40] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, and A. Mohaisen, "Adversarial learning attacks on graph-based iot malware detection systems," in *39th IEEE International Conference on Distributed Computing Systems, ICDCS*, 2019, pp. 1296–1305.

[41] 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 classifier," in *40th IEEE International Conference on Distributed Computing Systems, ICDCS*, 2020, pp. 1296–1305.

[42] J. Choi, A. Abusnaina, A. Anwar, A. Wang, S. Chen, D. Nyang, and A. Mohaisen, "Honor among thieves: Towards understanding the dynamics and interdependencies in iot botnets," in *2019 IEEE Conference on Dependable and Secure Computing (DSC)*, 2019, pp. 1–8.

[43] Y. Jia, Y. Xiao, J. Yu, X. Cheng, Z. Liang, and Z. Wan, "A novel graph-based mechanism for identifying traffic vulnerabilities in smart home iot," in *2018 IEEE Conference on Computer Communications, INFOCOM*. IEEE, 2018, pp. 1493–1501.

[44] Y. Wan, K. Xu, G. Xue, and F. Wang, "Iotargos: A multi-layer security monitoring system for internet-of-things in smart homes," in *39th IEEE Conference on Computer Communications, INFOCOM*. IEEE, 2020, pp. 874–883.

[45] T. Gu, Z. Fang, A. Abhishek, H. Fu, P. Hu, and P. Mohapatra, "Iotgaze: Iot security enforcement via wireless context analysis," in *39th IEEE Conference on Computer Communications, INFOCOM*. IEEE, 2020, pp. 884–893.