SHELLCORE: Automating Malicious IoT Software Detection Using Shell Commands Representation

Hisham Alasmary, Afsah Anwar, Ahmed Abusnaina, *Graduate Student Member, IEEE*, Abdulrahman Alabduljabbar, Mohammed Abuhamad, An Wang, *Member, IEEE*, Daehun Nyang, Amro Awad, and David Mohaisen

***Abstract*—The Linux shell is a command-line interpreter that provides users with a command interface to the operating system, allowing them to perform various 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 Internet of Things (IoT) devices, malware authors can abuse the Linux shell of those devices to propagate infections and launch large-scale attacks, e.g., Distributed Denial of Service. In this work, we provide a first look at the tasks managed by shell commands in Linux-based IoT malware toward 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 data set of shell commands, including malicious commands extracted from 2891 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 a 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*—Internet of Things (IoT) security, Linux shell commands, machine learning, malware detection.**

Manuscript received January 1, 2021; revised March 19, 2021 and April 28, 2021; accepted May 26, 2021. Date of publication June 3, 2021; date of current version February 4, 2022. This work was supported in part by the Global Research Laboratory (GRL) Program of the National Research Foundation (NRF) funded by the Ministry of Science, Information, and Communication Technologies (ICT) and Future Planning under Grant NRF-2016K1A1A2912757; in part by Air Force Research Laboratory (AFRL) Summer Program; in part by NSF under Grant CNS-1809000 and Grant CNS-1814417; in part by Cyber Florida Seed Grant; in part by the Institute for Smart, Secure and Connected Systems at Case Western Reserve University through a grant provided by the Cleveland Foundation. *(Hisham Alasmary and Afsah Anwar contributed equally to this work.) (Corresponding author: David Mohaisen.)*

Hisham Alasmary is with the Department of Computer Science, King Khalid University, Abha 61421, Saudi Arabia, and also with the Department of Computer Science, University of Central Florida, Orlando, FL 32816 USA.

Afsah Anwar, Ahmed Abusnaina, Abdulrahman Alabduljabbar, and David Mohaisen are with the Department of Computer Science, University of Central Florida, Orlando, FL 32816 USA (e-mail: mohaisen@ucf.edu).

Mohammed Abuhamad was with the the Department of Computer Science, University of Central Florida, Orlando, FL 32816 USA. He is now with the Department of Computer Science, Loyola University Chicago, Chicago, IL 60660 USA.

An Wang is with the Department of Computer and Data Science, Case Western Reserve University, Cleveland, OH 44106 USA.

Daehun Nyang is with the Cyber Security Major, Division of Software and Engineering, Ewha Womans University, Seoul 03760, South Korea.

Amro Awad is with the Department Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695 USA.

Digital Object Identifier 10.1109/JIOT.2021.3086398

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 seamlessly allow invocation of Linux capabilities. This utilization, which is essential for many edge applications, is sometimes exploited by malicious actors (malactors) to launch malicious activities and automate attacks and malware proliferation.

Indeed, the increasing use of IoT devices for everyday activities has been paralleled with IoT's susceptibility to risks, including significant 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. The 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 the software, such as Busybox [6], to implement Linux capabilities, the attacks are designed as task managed through the Linux/Unix-based shell commands.

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, and 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 use the shell to flood the HTTP of a victim and to remove the traces of execution by executing the *rm* command [12].

Significance: Detecting the malicious tasks managed through shell commands is essential. 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 the tasks managed through shell commands that appear in the static analysis of IoT malware binaries and understand their intrinsic features toward their detection. It is important to note that there have 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 data set 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 malware code to extract the shell used in the malware samples. For the shell commands that were initiated by a benign process, we collect a data set 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 malicious commands.

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

1) *C-1:* Using shell commands extracted from 2891 recent IoT malware samples along with a benign data set, 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.
2) *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%. Furthermore, acknowledging the different variants and flavors of Linux/Unix-like operating systems (OSs), which presents the complexity of encompassing all supported commands in these systems, we conduct a set of experiments by exclusively learning upon the malicious data set followed by testing upon the benign and the malicious data set. Our detection approach can be applied to files compiled for any processor architecture [e.g., ARM, MIPS, Power principal component (PC), etc.] as long as the shell commands are extracted, which can be done efficiently.

Organization: The remainder of this article 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 detail; 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 VI we review the related work, and finally, draw concluding remarks in Section VII.

II. Problem Statement and Approach Overview

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

A. Problem Statement

The problem we tackle in this article is malware detection using shell commands. 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 shell command initiated by a benign process, 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 detection 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 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. 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.

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 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 and 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, 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).

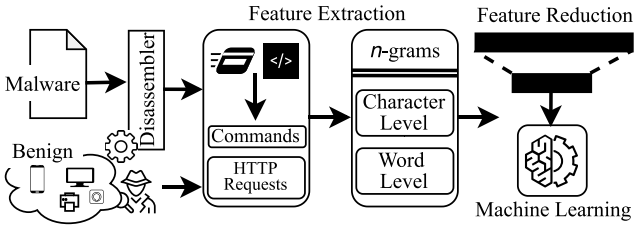

Fig. 1. Workflow of ShellCore, highlighting the sources of data and its division by class (malicious or benign). The raw data are preprocessed to extract shell commands. The shell commands are represented as 1) characters and 2) words, which are fed to learning networks for detection.

Upon representing the malicious and benign commands as feature vectors, ShellCore aims to detect malicious commands, as shown in Fig. 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 our detection system's feature extraction and classification methods.

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 data set. 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. Toward this, we leverage the term-level NLP-based approach by considering words in the samples as features. Additionally, since such an approach misses 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 the occurrence of space and/or characters as tokenizers. In contrast, 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 be 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 $I_1 = I_{11}, I_{12}, I_{13}, \ldots, I_{1N}$.

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 $l \leq 2$ calls 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 be 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 $I_2 = I_{21}, I_{22}, I_{23}, \ldots, I_{2N}$.

3) Feature Representation: To represent every element in the data set from a defined reference point, they are represented with respect to axes in space. In particular, every command/sample in the data set 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, commands are augmented such that every component of the commands in the data set has representation in the feature space. Every command in the data set is then represented in the space of n axes, where n is the size of feature space. To do so, we devise multiple representations of the commands, including the words in the commands and splitting the commands by spaces and every special character. We also form a feature vector by considering every letter and special character as corresponding features in the formed vector. 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 data set is then mapped to an index in the sparse vector representation, i.e., the feature vector for every element in the data set, 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., *multihot encoding*).

Specifically, to generate the vector representation of each command or sample, we first created a corpus consisting of the character-level and term-level combinations, referred to as tokens, occurring in either malware or benign commands. The vector for each command or sample reflects the frequency of each token in the shell command or sample, respectively.

5) Encoding Syntax: A vital 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 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 the two models mentioned above, we create their respective bag of words. The bags contain words I_{1i}^{k}, where $i \in [1, N]$ and $k \in [1, m]$, such that N is the number of words in a command and k is the total number of commands in the data set, along with n-grams. Therefore, the words in all the commands as per the term-level NLP model can be combined as $I_{11}^{1}, I_{11}^{2}, I_{11}^{3}, \ldots, I_{12}^{1}, I_{13}^{1}, \ldots, I_{1N}^{m}$ Let B be the bag of word for the data set, such that $B = B_1, B_2, B_3, \ldots, B_t$, 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 tth index be represented as the frequency of occurrence, of the tth word in the bag, in the command. $F = f_{B_1}, f_{B_2}, f_{B_3}, \ldots, f_{B_t}$, 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 model's performance to become 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, leading to: 1) high cost to perform learning and 2) overfitting, i.e., the model may perform very well on the training data set but poorly on the test data set.

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 is a statistical technique used to extract features from multiple raw features, where raw features are n-grams and statistical measurements. PCA creates new variables, named 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 data set 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 PCs 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.

Logistic Regression (LR): LR is a statistical method that employs a logistic function to model a binary-dependent variable, referred to as binary classification ("0" or "1"). Given (X, Y) as an input training set, LR learns to differentiate between positive ("1") and negative ("0") segments for each category, with the assumption that they have a linear relationship. LR, in the higher domain, estimates and optimizes the boundary between the positive and negative classes by minimizing the following function:

$$\mathsf{Loss}(f(X), Y) = \begin{cases} -\log(f(X)), & Y = 1 \\ -\log(1 - f(X)), & Y \neq 1 \end{cases} \tag{1}$$

where $f(X)$ is the LR model's current prediction and Y are the labels of the ground-truth set.

Random Forest (RF): RF is a nonlinear classification algorithm that consists of N decision trees each of which is trained on a collection of random features. The RF method reduces the variation in the performance of individual trees and minimizes the impact of noise on the training process. The final prediction of RF classifier with N decision trees is determined by a majority vote over the predictions or by averaging the prediction of all trees, determined as follows:

$$f_{\mathrm{RF}} = \frac{1}{N} \sum_{n=1}^{N} f_n\left(X'_s\right) \tag{2}$$

where for a randomly selected feature set, $(X'_{\cdot} \subset X_{\cdot})$, f_n is the nth tree's prediction and X'_s is the segment's s vector.

1) Deep Neural Networks: The deep neural network (DNN) is a type of connected and feedforward 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, $w_1 + w_2 + w_3 + \cdots w_q$, 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 softmax activation function h, defined as $h(x) = [1/(1+e^{-x})]$, outputting the prediction of the classifier.

C. Term- and Character-Level NLP-Based Approaches

1) Term-Level NLP-Based Model: The term-level 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. Notably, it uses 1- to 5-grams, with tenfold cross-validation.

2) Character-Level NLP-Based Model: We note that the term level considers the words and neglects the characters, spaces, and words with a length of less than three. This, in turn, presents a significant 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

We divide our evaluation into two parts. First, we build a detection system to detect malicious commands by considering every individual command in the data set. Second, this detection system is then extended to detect 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 data sets, 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 99.9% of the variance in the training data set.

To set out, we begin by describing the process of assembling the data set 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).

A. Malicious Data Set and Commands Extraction

We obtain a data set of 2891 randomly selected IoT malware samples from the IoTPOT project [15], 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 [16]. Additionally, IoTPOT covers eight different architectures. Table I depicts the malware distribution according to their architectures and percentages. Fig. 1 shows our approach, end to end, split into three modules: 1) initial discovery; 2) command extraction; and 3) detection. Our data collection is represented 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

TABLE I
MALWARE DATA SET 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%

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 2891 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, and 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 1273 patterns and use them to extract the shell commands from other samples. Our definition of shell commands covers the tasks that can be instructed using a terminal, such as the Linux/Unix-like system commands, HTTP messages, and automated tasks. For example, strings beginning with shell command keywords, such as cd, between if and fi, kill, wait, disown, suspend, fc, history, break, GET, and POST, among other similar command structures, are extracted. Malware samples use the shell commands to achieve their objectives, such as traversing directories (cd), killing a running process of interest (kill), communicating with a C2 (GET and TFTP), and exfiltrating data (POST). For coverage of those patterns, we use online resources to build a data set 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 at the location of shell command references. We then navigate to the address where a particular string is quoted and disassemble at that offset.

B. Benign Data Set and Commands Extraction

To evaluate ShellCore, acquiring a benign shell commands data set 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 data set 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, and 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, consistently. In total, we collect a data set of about 143 MB from these volunteers, consisting of 5772 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, we attempt to monitor those intent to distribute malicious payloads and scripts 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 Fig. 2.

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 primary 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. Fig. 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

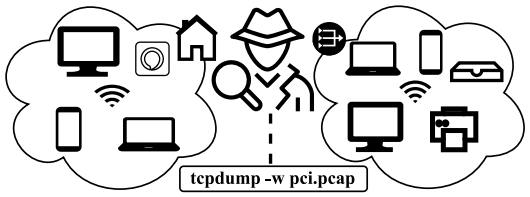

Fig. 2. Monitoring stations for creation of benign data set creation. Two network implementations are used: 1) NAT and 2) a home network.

network is monitored for 24 h, where all the network traffics are 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 7 h. Using these settings, we gather a data set 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) results in 28 578 754 individual payloads, and only 1 625 143 are not encrypted, which we utilize for our benign data set. From the second network (Network 2), five sources generate 4735 unencrypted payloads in total, which we use as part of our data set. In total, our benign data set consists of three parts: 1) *bash* (5772 commands); 2) network 1 (1 625 143 commands); and 3) network 2 (4755 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. In the data representation, we first consider a corpus compiled from both the malware and benign commands to extract the vector representation of each sample. However, determining what is benign is an open challenge, particularly, in malware detection using machine learning, where benign is assumed not to have a fixed pattern, while the malicious samples share behavioral patterns. Toward this, we also investigate using malware-only commands corpus in the process of extracting the vector representation of the samples. This will reduce the bias toward the benign data set and demonstrate the effectiveness of the proposed approach, as the feature representation is generated solely using the malware samples, and is only reflected on the benign samples.

C. Evaluation Settings and Metrics

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

1) Data Set: 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 the low deviation. We notice that Network 2 (corresponding to the IoT devices setting) and Malware data sets have the closest lengths overall, per their distributions' average and standard deviation characteristics.

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 follows.

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.

We used the accuracy (AC), F-score (F-1), false-negative rate (FNR), and false-positive rate as evaluation metrics, which are defined as follows.

1) $\text{AC} = (\text{TP} + \text{TN})/(\text{TP} + \text{TN} + \text{FP} + \text{FN})$.
2) $\text{F-1} = 2\text{TP}/(2\text{TP} + \text{FP} + \text{FN})$.
3) $\text{FNR} = \text{FN}/(\text{TP} + \text{FN})$.
4) and $\text{FPR} = \text{FP}/(\text{FP} + \text{TN})$.

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

D. Detecting Malicious Commands

We use ShellCore to detect individual malware commands. We first present the results of the term-level model, followed by the character-level model. On average, the term-level model provides an accuracy of more than 99% along with an FNR of less than 0.1% and FPR of less than 0.20% as shown in Table IV (left), with all approaches performing similarly. We then test the performance of the character-level NLP-based model for detecting individual malicious commands over the same data set. As shown in Table IV (left), the approach achieved similar results on the term level, with up to 99.87% accuracy using LR and DNN.

Malware-Based Corpus: Next, we investigate the performance of both term- and character-level representations extracted using the malware samples only, and without considering the benign samples. Benign software are diverse in behavior, intuition, appearance, and goal, while malicious software share commonalities within their design [17]. Therefore, identifying benign samples should not be dependent on the existence of "benign" patterns, but the nonexistence of "malicious" patterns. This, in turn, will reduce the bias toward the training benign data set, which is essential considering that obtaining a representative

TABLE II
DATA SOURCES IN OUR DATA SET. "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	178,261	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 DATA SETS. 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	178,261	984	5	293.91	384	168.03

TABLE IV
EVALUATION RESULTS (%) OF MALICIOUS COMMANDS DETECTION FOR BOTH TERM- AND CHARACTER-LEVEL REPRESENTATIONS. LEFT: THE VECTOR REPRESENTATION IS GENERATED FROM COMMANDS EXTRACTED FROM BOTH BENIGN AND MALWARE. RIGHT: ONLY THE MALWARE COMMANDS ARE USED TO GENERATE THE VECTOR REPRESENTATION

Target	ML	Term-level				Character-level				Term-level				Character-level			
		Acc.	F-1	FNR	FPR	Acc.	F-1	FNR	FPR	Acc.	F-1	FNR	FPR	Acc.	F-1	FNR	FPR
Command	LR	**99.86**	**99.86**	**0.03**	0.20	**99.87**	**99.87**	**0.12**	0.15	89.44	89.43	14.03	**8.43**	98.03	98.03	**1.79**	2.08
	RF	99.84	99.84	0.09	0.20	99.78	99.78	0.27	0.19	84.95	84.96	19.52	12.29	98.19	98.19	2.32	**1.49**
	DNN	99.85	99.85	0.08	**0.19**	**99.87**	**99.87**	**0.12**	**0.14**	**89.52**	**89.52**	**13.27**	8.77	**98.24**	**98.24**	**1.79**	1.74
File	LR	97.76	97.77	0.63	3.04	99.70	99.70	**0.07**	0.42	66.37	65.83	55.82	22.62	**99.84**	**99.84**	**0.07**	0.21
	RF	**99.08**	**99.08**	1.25	**0.76**	**99.91**	**99.91**	0.14	**0.07**	**67.48**	**67.43**	**49.06**	24.32	99.79	99.79	0.28	**0.17**
	DNN	97.16	97.18	**0.35**	4.08	99.28	99.29	0.14	1.00	66.67	65.58	55.95	**22.10**	99.26	99.26	0.14	1.04

benign data set is an open challenge. Therefore, the benign samples were excluded from the process of constructing the feature space for this evaluation. Table IV (right) shows the performance of both representations. In contrast to the previous results, the term-level representation performance was significantly affected (∼89.52% accuracy), while the character-level representation maintains a performance of up to 98.24% (only 1.6% performance degradation) using the DNN-based 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) Data Set: 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 the number of commands per sample, of the real-world malicious samples in our data set. Then, by using the sampling techniques, we generate a statistically similar (sizewise) data set 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 data set. In doing so, the commands corresponding to a file are represented as a feature vector of that file. Similar to the detection of the individual command, as shown in Table IV (left), we evaluated both the term-level and character-level representations, with the character-level model yielding a higher detection rate of 99.91% with 0.14% and 0.07% 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 FNR. This reflects the improved feature representation technique and emphasizes the importance of special characters.

Malware-Based Corpus: We also investigate the performance of malware-only term- and character-level representations for malicious files and commands detection. Table IV (right) shows the performance of both representations. Like the malware command detection, the term-level approach performance is highly affected, reduced from 99% to 67% accuracy. However, the character-level approach maintained similar results (i.e., 1% performance reduction in malware command detection). This indicates the stability

of the character-level representations in malicious behavior modeling, a characteristic that may not hold true for the term-level representation.

V. Discussion

Prior works have used different approaches for IoT malware detection, including control flow graphs (CFGs) and image-based representation of binaries. Studies have also shown that Linux-based malware is structurally different from the traditional Android malware [18]. Furthermore, although studies have shown the abuse of the windows Powershell [19], they differ functionally from the Linux shell. Considering the use of shell by the adversaries toward their intent [20], we argue that shell commands can be used as a modality for effective detection as well. Additionally, a shell command-based modality can be used to detect fileless malware. We further discuss the implications of this work in the following.

A. Use of Shell as Weapon

The shell abstracts details of the communication between the application and the OS, and is used by applications for interacting with the file system, OS, etc. However, adversaries use shell commands for their malicious intents, e.g., interacting with the command and control server, directory traversal, and data exfiltration. This can be facilitated by using 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 [21].

A recent vulnerability (CVE-2019-1656), which results from the improper input validation in Linux OS 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 users' credentials 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.

B. 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 [22], and installing ELF executables on Android systems [23]. 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.

C. Malware Detection

Many efforts have been dedicated to addressing IoT's security threats from the hardware, software, and application perspectives. Some also argue that there is a need for a cross-layer approach for comprehensive protection of the IoT systems [24]. Meanwhile, IoT malware has been on the rise. Given the difficulty of obtaining samples, very few works have been done to detect IoT malware, and even less using residual strings in the binaries. Section VI 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 abuses the host device's shell, detecting them at the shell will help safeguard the device from becoming infected. Additionally, the malware accesses 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.

D. Applications

ShellCore detects malicious software by leveraging the use of the shell in binaries. Given the increase in IoT malware attacks, their use of shell commands can be leveraged for their detection and the associated malware intent unveiled by the commands. Additionally, ShellCore can be leveraged to detect fileless attacks [25], a new and emerging type of attack where the adversary uses a target device's terminal to execute successive commands that implement the malicious intent. As a file is unavailable for analysis, the execution of commands can be used as a modality toward their analysis and detection.

E. Limitations

In this study, we analyze the IoT malware statically to extract shell commands from the malware disassembly. Thus, our approach is limited to malware that does not employ obfuscation. Prior studies have shown that obfuscation is still uncommon among IoT malware [26], making our model applicable under existing circumstances. Additionally, prior studies have also shown the use of standard packers by IoT malware, e.g., UPX [26], [27]. The standard packers' *unpack* module can, thus, be leveraged to extract the malware binary, and our model can then be used to detect the malicious software.

TABLE V
COMPARISON WITH RELATED WORK. AUC: AREA UNDER THE CURVE, TPR: TP RATE, TNR: TN RATE, AC: ACCURACY, FNR: FN RATE, FPR: FP 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: MULTILAYER 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.* [28]	Web shell	481	Analysis	—	SDA
Uitto *et al.* [10]	Linux shell	13,257	Analysis	—	Diversification
Tian *et al.* [29]	Web shell	7,681	Detection	Pr. (98.6%), Re. (98.6%), F1 (98.6%)	CNN
Rusak *et al.* [30]	PowerShell	4,079	Detection	AC (85%)	DL
Hendler *et al.* [19]	PowerShell	66,388	Detection	AUC (98.5-99%), TPR (0.24-0.99%)	NLP, CNN
Li *et al.* [31]	PHP web shell	950	Detection	AUC (98.7%), AC (91.7%), FPR (1.0%)	RF, SVM, GBT
Stokes *et al.* [32]	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

A significant challenge in our study is generating a reliable data set of both malware and benign samples. While we reconstruct the command usage by malware through extracting commands from the malware codebase, we extract the benign usage from the shell by the Linux-based devices. We acknowledge that the benign data set might not be considered as a representative ground-truth benign data set with absolute confidence, considering that the different variants and flavors of Linux/Unix-like OSs present the complexity of encompassing all supported commands in such systems. Therefore, and to account for that shortcoming, we evaluated our models using representations extracted exclusively from the malware samples.

VI. 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 and IoT malware detection, which are related to this work. No prior work exists on IoT shell commands.

A. Shell Commands

Hendler *et al.* [19] 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. Furthermore, Anwar *et al.* [20] statically analyzed the IoT malware and specified about the presence of shell commands in their disassembly. They use them along with other features, such as strings and CFGs toward malware detection.

B. 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.* [28] statically and dynamically analyzed a set of Web shells to uncover features of malicious hypertext preprocessor shells. Tian *et al.* [29] proposed a system to detect malicious Web shell commands using CNN and word2vec-based approaches. In a similar context, Rusak *et al.* [30] proposed a deep learning approach to classify malicious *PowerShell* by families using the abstract syntax trees representation of the *PowerShell* commands. Li *et al.* [31] proposed an ML model to detect malicious Web shells written in PHP, achieving an accuracy of 91.7%. Moreover, Stokes *et al.* [32] employed a recurrent deep learning model to detect malicious VBScripts by using a data set of first 1000 bytes of 240 504 VBScript files and achieving a TPR of 69.3% and an FPR of 1.0%.

C. 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.* [15] proposed IoTPOT, a detection system that supports different malware architectures to analyze and detect Telnet-based attacks on IoT devices. Dang *et al.* [33] 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. They found that 99.7% of fileless attacks use shell commands, making ShellCore very relevant, since it is capable of detecting these types of attacks. Another work proposed by Perdisci *et al.* [34] introduced IoTFinder, a multilabel classifier that automatically learns statistical DNS traffic fingerprints for large-scale detection of IoT devices. Alrawi *et al.* [35] proposed a modeling methodology for home IoT devices to identify unencrypted traffic and other vulnerability.

Recent works have focused on detecting IoT malware traffic, e.g., IoT network packets [36], [37], by introducing early detection of IoT malware network activity (EDIMA). EDIMA is an IoT malware detection method using supervised ML algorithms atop the analyzed traffic of IoT devices and large-scale network scanning. Bendiab *et al.* [38] proposed a zero-day malware detection and classification method using deep learning atop the analyzed IoT malware traffic and visual representations. The challenge of finding the best suitable algorithm to use in extracting features from executable files was

addressed by Darabian *et al.* [39] using multiview data extraction. Another approach proposed by Liu *et al.* [40] used a pretrained RF that considers the values of misclassification features to build a generic algorithm. Their proposed framework's primary goal is to detect IoT malware on the Android OS with prior knowledge about the devices.

Su *et al.* [41] used a lightweight CNN for IoT's malware families classification after converting their binaries to grayscale images and achieved 94.0% of accuracy in classifying DDoS malware in IoT networks and 81.8% of accuracy in detecting two prominent malware families (i.e., Mirai and Linux Gafgyt). Lei *et al.* [42] introduced a graph-based IoT malware detection technique called "EveDroid" as an event-aware Android malware detection tool. Instead of using the API calls to capture malware behavior, EveDroid uses event groups to exploit the apps' behavioral patterns at a higher level while providing an F1-score of 99%. In the health-care domain of IoT, Guerar *et al.* [43] discussed malware vulnerabilities of mobile OSs and IoT sensors. They introduced the Invisible CAPTCHA to decide if a user is a bot or not by considering the tap and vibration events from the user recorded behaviors while using the mobile devices rather than asking the user to enter the CAPTCHA content manually.

Bertino and Islam [44] 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.* [45] proposed Probe-IoT, a forensic system that investigates IoT-related malicious activities. Similarly, Montella *et al.* [46] 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.* [47] analyzed a large Linux malware data set by studying their behavior, and discussed obfuscation techniques that malware authors use. Furthermore, Alasmary *et al.* [48] and Anwar *et al.* [20] used 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 [49] and also proposed effective defenses [50]. Recent arts have also focused on exploring the IoT network environment. Choi *et al.* [51] explored the presence of endpoints the disassembly of the IoT malware binaries toward 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 [52]–[54].

VII. Conclusion

We proposed ShellCore, a machine learning-based approach to detect shell commands used in IoT malware. We analyzed malicious shell commands from a data set of 2891 IoT malware samples, along with a data set 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 can detect their source with high accuracy.

References

[1] Google. (2017). *Nest Cam IQ Indoor: State-of-the-Art Smart*. [Online]. Available: 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 *Proc. 4th ACM/IEEE Symp. Edge Comput.*, 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 *Proc. IEEE/ACM Symp. Edge Comput. (SEC)*, Seattle, WA, USA, 2018, pp. 132–144.

[4] L. H. Newman. (2018). *Github Survived the Biggest DDoS Attack Ever Recorded*. [Online]. Available: https://www.wired.com/story/github-ddos-memcached/

[5] KrebsOnSecurity. (2016). *Hacked Cameras, DVRs Powered Today's Massive Internet Outage*. [Online]. Available: https://tinyurl.com/zxrfm36

[6] N. Wells, "BusyBox: A Swiss army knife for Linux," *Linux J.*, vol. 2000, no. 78, p. 10, 2000.

[7] NVD. (2018). *NVD Vulnerability Metrics*. [Online]. Available: https://nvd.nist.gov/vuln-metrics/cvss

[8] Developers. (2010). *CVE-2010-4258: Turning Denial-of-Service into Privilege Escalation*. [Online]. Available: 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 *Proc. Asia–Pac. Workshop Syst. (APSys)*, 2011, pp. 1–5.

[10] J. Uitto, S. Rauti, J. Mäkelä, and V. Leppänen, "Preventing malicious attacks by diversifying Linux shell commands," in *Proc. 14th Symp. Program. Lang. Softw. Tools (SPLST)*, 2015, pp. 206–220.

[11] J. C. Matherly, "SHODAN, the computer search engine," Accessed: Jun. 5, 2021. [Online]. Available: https://www.shodan.io/

[12] M. Antonakakis *et al.*, "Understanding the Mirai Botnet," in *Proc. 26th USENIX Security Symp.*, 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. London, U.K.: Springer, 2001.

[14] H. Uğuz, "A two-stage feature selection method for text categorization by using information gain, principal component analysis and genetic algorithm," *Knowl. Based Syst.*, vol. 24, no. 7, pp. 1024–1032, 2011.

[15] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "IoTPOT: A novel honeypot for revealing current IoT threats," *J. Inf. Process.*, vol. 24, no. 3, pp. 522–533, 2016.

[16] IoTPOT. (2021). *IoTPOT—Analysing the Rise of IoT Compromises*. [Online]. Available: https://ipsr.ynu.ac.jp/iot/

[17] M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar, "The security of machine learning," *Mach. Learn.*, vol. 81, no. 2, pp. 121–148, 2010.

[18] H. Alasmary, A. Anwar, J. Park, J. Choi, D. Nyang, and A. Mohaisen, "Graph-based comparison of IoT and android malware," in *Proc. 7th Int. Conf. Comput. Data Social Netw. (CSoNet)*, 2018, pp. 259–272.

[19] D. Hendler, S. Kels, and A. Rubin, "Detecting malicious PowerShell commands using deep neural networks," in *Proc. Asia Conf. Comput. Commun. Security (AsiaCCS)*, Incheon, South Korea, 2018, pp. 187–197.

[20] A. Anwar, H. Alasmary, J. Park, A. Wang, S. Chen, and D. Mohaisen, "Statically dissecting Internet of Things malware: Analysis, characterization, and detection," in *Proc. Int. Conf. Inf. Commun. Security (ICICS)*, 2020, pp. 443–461.

[21] M. Koch, *An Introduction to Linux-Based Malware*, SANS Inst. InfoSec Reading Room, Rockville, MD, USA, 2015.

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

[23] A.-D. Schmidt *et al.*, "Enhancing security of Linux-based android devices," in *Proc. 15th Int. Linux Kongress*, 2008, pp. 1–16.

[24] A. Wang, A. Mohaisen, and S. Chen, "XLF: A cross-layer framework to secure the Internet of Things (IoT)," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Dallas, TX, USA, 2019, pp. 1830–1839.

[25] S. Mansfield-Devine, "Fileless attacks: Compromising targets without malware," *Netw. Security*, vol. 2017, no. 4, pp. 7–11, 2017.
[26] E. Cozzi, P.-A. Vervier, M. Dell'Amico, Y. Shen, L. Bilge, and D. Balzarotti, "The tangled genealogy of IoT malware," in *Proc. Annu. Comput. Security Appl. Conf.*, 2020, pp. 1–16.
[27] *UPX: The Ultimate Packer for eXecutables*. Accessed: Jan. 23, 2021. [Online]. Available: https://upx.github.io/
[28] 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 *Proc. 25th Int. Conf. World Wide Web (WWW)*, 2016, pp. 1021–1032.
[29] Y. Tian, J. Wang, Z. Zhou, and S. Zhou, "CNN-webshell: Malicious Web shell detection with convolutional neural network," in *Proc. VI Int. Conf. Netw. Commun. Comput. (ICNCC)*, 2017, pp. 75–79.
[30] G. Rusak, A. Al-Dujaili, and U.-M. O'Reilly, "AST-based deep learning for detecting malicious powershell," in *Proc. Conf. Comput. Commun. Security (CCS)*, 2018, pp. 2276–2278.
[31] Y. Li, J. Huang, A. Ikusan, M. Mitchell, J. Zhang, and R. Dai, "*ShellBreaker*: Automatically detecting PHP-based malicious Web shells," *Comput. Security*, vol. 87, 2019, Art. no. 101595.
[32] J. W. Stokes, R. Agrawal, and G. McDonald, "Detection of malicious Vbscript using static and dynamic analysis with recurrent deep learning," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, Barcelona, Spain, 2020, pp. 2887–2891.
[33] F. Dang *et al.*, "Understanding fileless attacks on Linux-based IoT devices with honeycloud," in *Proc. 17th Annu. Int. Conf. Mobile Syst. Appl. Serv. (MobiSys)*, 2019, pp. 482–493.
[34] R. Perdisci, T. Papastergiou, O. Alrawi, and M. Antonakakis, "IoTFinder: Efficient large-scale identification of IoT devices via passive DNS traffic analysis," in *Proc. IEEE Eur. Symp. Security Privacy (EuroS&P)*, Genoa, Italy, Sep. 2020, pp. 474–489.
[35] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "SoK: Security evaluation of home-based IoT deployments," in *Proc. IEEE Symp. Security Privacy (SP)*, San Francisco, CA, USA, May 2019, pp. 1362–1380.
[36] C. D. McDermott, F. Majdani, and A. V. Petrovski, "BotNet detection in the Internet of Things using deep learning approaches," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Rio de Janeiro, Brazil, 2018, pp. 1–8.
[37] A. Kumar and T. J. Lim, "EDIMA: Early detection of IoT malware network activity using machine learning techniques," in *Proc. 5th IEEE World Forum Internet Things (WF-IoT)*, Limerick, Ireland, 2019, pp. 289–294.
[38] G. Bendiab, S. Shiaeles, A. Alruban, and N. Kolokotronis, "IoT malware network traffic classification using visual representation and deep learning," in *Proc. 6th IEEE Conf. Netw. Softw. (NetSoft)*, Ghent, Belgium, 2020, pp. 444–449.
[39] H. Darabian *et al.*, "A multiview learning method for malware threat hunting: Windows, IoT and android as case studies," *World Wide Web*, vol. 23, no. 2, pp. 1241–1260, 2020.
[40] X. Liu, X. Du, X. Zhang, Q. Zhu, H. Wang, and M. Guizani, "Adversarial samples on android malware detection systems for IoT systems," *Sensors*, vol. 19, no. 4, p. 974, 2019.
[41] J. Su, D. V. Vargas, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai, "Lightweight classification of IoT malware based on image recognition," in *Proc. IEEE 42nd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 2. Tokyo, Japan, 2018, pp. 664–669.
[42] T. Lei, Z. Qin, Z. Wang, Q. Li, and D. Ye, "EveDroid: Event-aware android malware detection against model degrading for IoT devices," *IEEE Internet Things J.*, vol. 6, no. 4, pp. 6668–6680, Aug. 2019.
[43] M. Guerar, A. Merlo, M. Migliardi, and F. Palmieri, "Invisible CAPPCHA: A usable mechanism to distinguish between malware and humans on the mobile IoT," *Comput. Security*, vol. 78, pp. 255–266, Sep. 2018.
[44] E. Bertino and N. Islam, "Botnets and Internet of Things security," *IEEE Comput.*, vol. 50, no. 2, pp. 76–79, Feb. 2017.
[45] M. Hossain, R. Hasan, and S. Zawoad, "Probe-IoT: A public digital ledger based forensic investigation framework for IoT," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM)*, 2018, pp. 1–2.
[46] R. Montella, M. Ruggieri, and S. Kosta, "A fast, secure, reliable, and resilient data transfer framework for pervasive IoT applications," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM)*, Honolulu, HI, USA, 2018, pp. 710–715.
[47] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding Linux malware," in *Proc. IEEE Symp. Security Privacy (S&P)*, 2018, pp. 161–175.
[48] H. Alasmary *et al.*, "Analyzing and detecting emerging Internet of Things Malware: A graph-based approach," *IEEE Internet Things J.*, vol. 6, no. 5, pp. 8977–8988, Oct. 2019.
[49] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, and A. Mohaisen, "Adversarial learning attacks on graph-based IoT malware detection systems," in *Proc. 39th IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Dallas, TX, USA, 2019, pp. 1296–1305.
[50] H. Alasmary *et al.*, "Soteria: Detecting adversarial examples in control flow graph-based malware classifier," in *Proc. 40th IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2020, pp. 1296–1305.
[51] J. Choi *et al.*, "Honor among thieves: Towards understanding the dynamics and interdependencies in IoT botnets," in *Proc. IEEE Conf. Depend. Secure Comput. (DSC)*, Hangzhou, China, 2019, pp. 1–8.
[52] 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 *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Honolulu, HI, USA, 2018, pp. 1493–1501.
[53] Y. Wan, K. Xu, G. Xue, and F. Wang, "IoTArgos: A multi-layer security monitoring system for Internet-of-Things in smart homes," in *Proc. 39th IEEE Conf. Comput. Commun. (INFOCOM)*, Toronto, ON, Canada, 2020, pp. 874–883.
[54] T. Gu, Z. Fang, A. Abhishek, H. Fu, P. Hu, and P. Mohapatra, "IoTGaze: IoT security enforcement via wireless context analysis," in *Proc. 39th IEEE Conf. Comput. Commun. (INFOCOM)*, Toronto, ON, Canada, 2020, pp. 884–893.