Efficient Signature Generation for Classifying Cross-Architecture IoT Malware

Mohannad Alhanahnah*, Qicheng Lin*, Qiben Yan*, Ning Zhang†, Zhenxiang Chen‡
*Computer Science and Engineering Department, University of Nebraska Lincoln, Lincoln, NE, USA Email: yan@unl.edu
†Computer Science Department, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA
‡Computer Science Department, University of Jinan, Jinan, Shandong, China

***Abstract*—Internet-of-Things (IoT) devices are increasingly targeted by adversaries due to their unique characteristics such as constant online connection, lack of protection, and full integration in people's daily life. As attackers shift their targets towards IoT devices, malware has been developed to compromise IoT devices equipped with different CPU architectures. While malware detection has been a well-studied area for desktop PCs, heterogeneous processor architecture in IoT devices brings in unique challenges. Existing approaches utilize static or dynamic binary analysis for identifying malware characteristics, but they all fall short when dealing with IoT malware compiled for different architectures. In this paper, we propose an efficient signature generation method for IoT malware, which generates distinguishable signatures based on high-level structural, statistical and string feature vectors, as high-level features are more robust against code variations across different architectures. The generated signatures for each malware family can be used for developing lightweight malware detection tools to secure IoT devices. Extensive experiments with two datasets of 5,150 recent IoT malware samples show that our scheme can achieve 95.5% detection rate with 0% false positive rate. Moreover, the proposed scheme can achieve 85.2% detection rate in detecting novel IoT malware.**

I. Introduction

Gartner recently forecasts that 8.4 billion connected things will be in use worldwide in 2017, which represents 31% increase from 2016, and will reach 20.4 billion by 2020 [1]. With the growing popularity, IoT devices have become enticing targets for cyber-attackers. Since IoT devices are fully integrated in our daily life, compromised devices can cause unprecedented damages. Even worse, IoT devices are usually resource-constrained with low-profile processors, which prevents the deployment of sophisticated host-based defenses as we commonly use on personal computers (PCs). Consequently, the attackers endeavor to recruit vulnerable IoT devices to build a *large-scale* bot army to launch attack, and the number of IoT malware has more than doubled in 2017 [2]. This can be exemplified by the recent devastating DDoS attacks against a popular DNS service provider (Dyn), known as Dyn attack, which was launched by the notorious Mirai malware, and caused a major Internet service breakdown for a few hours [3]. Therefore, there is a pressing need to provide lightweight protection mechanisms on IoT devices to prevent malware infections.

Security has become a major concern that affects the adoption of IoT devices [4]. Recently, research efforts have been devoted to discovering bugs and vulnerabilities in IoT devices [5], [6]. Yet, IoT malware, as a new type of attack that targets exclusively IoT devices, has not been rigorously investigated. Furthermore, the outcome of IoT malware analysis can lead to lightweight and accurate detection schemes to mitigate threats. One common approach for detecting specific types of malware is to search for known patterns within malware binaries such as constant numbers or specific strings [7], e.g., IP addresses. While this approach can be very effective and efficient in pinpointing specific malware, it fails when dealing with samples that lack distinct constants. Other researchers have proposed graph-based malware detection using control flow graph (CFG) [8], call graph (CG) [9] for conducting pairwise graph matching. These approaches are too complex and resource-consuming to be deployed on IoT devices.

There are two major challenges in designing IoT malware detection mechanisms: first, different from traditional PC and mobile malware, one unique characteristic of IoT malware is its ability to infect devices with *multiple CPU architectures* including ARM, MIPS, x86, etc. Therefore, we need to design a lightweight *cross-architecture* signature generation scheme for detecting/classifying IoT malware; second, due to the resource constraints of IoT devices, the generated signatures should be compact and accurate, while the detection/classification should be energy-efficient in order to be deployed locally.

In this paper, we endeavor to overcome these challenges using a **data-driven approach** by investigating a set of real-world IoT malware samples. Our goal is to identify malware samples from the same family that share similarity in both codes and functionalities. Since the functionality of IoT malware is largely constrained by the limited resources of IoT devices, IoT malware tends to be *simple and forthright* without using complicated customization, obfuscation, or evasive techniques. Therefore, our proposed technique takes a lightweight approach by capturing high-level code structural, code statistics, and string features to develop lightweight signatures for IoT malware. These learned features provide insights into **the cross-architecture invariants**, based on which we generate a compact feature representation as the signature that can withstand cross-architectural variations.

The proposed malware detection system *integrates malware*

clustering and signature generation with the goal of detecting IoT malware accurately and efficiently. Specifically, in order to reduce the malware detection costs, we perform malware clustering to group together malware samples from the same family. The similarity scores among IoT malware samples are computed based on code statistics feature and graph structural feature extracted using the Bindiff tool [10]. Then, high-level string and statistical features of each malware group will be extracted to generate signatures for the detection of IoT malware within IoT devices. To the best of our knowledge, this is *the first work on signature generation and classification of IoT malware*. In summary, the contributions of this paper are as follows:

- Using two real-world IoT malware datasets with $5,150$ malware samples, we observe the cross-architectural similarity among malware samples from the same family. Based on this keen observation, we propose a multi-stage clustering mechanism to cluster these IoT malware samples into multiple families using the code statistics feature, high-level structural similarity, and N-gram string features.
- We design an efficient signature generation scheme to create signatures using reliable and easily extractable string and statistical features. The string feature is extracted using N-gram text analysis, while the statistical feature contains the code-level statistics. These features are carefully constructed and integrated to minimize the *inter-cluster similarity*, while maximizing the *intra-cluster similarity*.
- We run extensive experiments using datasets consisting of benign firmware binaries and additional malware samples downloaded from product websites and malware sharing servers. The evaluation results show the effectiveness of our signature-based detection mechanism, which achieves 95.5% detection rate with 0% false positive rate, and 85.2% detection rate in detecting novel IoT malware.

II. Motivation

IoT malware/vulnerability research. The priority of most IoT vendors are functionalities and faster pace of bringing product to market. The security of IoT systems has not received much attention. The most relevant research focuses on developing an IoT honeypot [11] called IoTPOT, used to allure malware to infect emulated IoT devices in the honeypot, which aims at collecting IoT malware binaries and the corresponding network traffic for further analysis. The malware binaries have been clustered into four distinct families based on simple command sequence and unique strings through manual analysis. Yet, they neglect the rich code-level features that facilitate a fine-grained characterization of IoT malware.

Vulnerability and bug discovery of IoT devices is a problem gaining attentions recently [5], [6], [12]. Eschweiler *et al.* [5] utilize graph matching approaches in conjunction with statistical features extracted from the disassembled binary codes to detect bugs. However, their goal is to identify similarity between individual vulnerable functions rather than matching binary files and generating detection signatures. Recently,

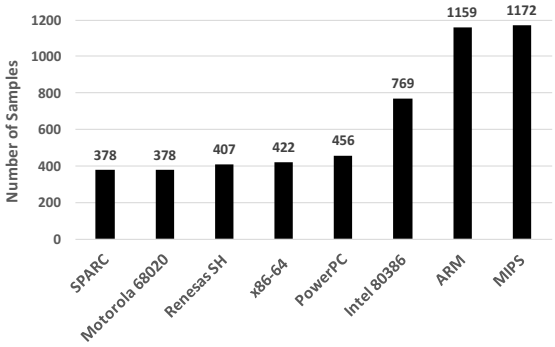

Figure 1: IoT malware distribution based on CPU types

Feng *et al.* [6] employ a scalable search method to improve the scalability and accuracy of cross-architecture bug search, where both the structural and statistical features are aggregated to create a high-level feature vector for vulnerability detection in real-time. All of the above methods use static analysis to extract features at basic block level using control flow graph (CFG). Yet, the high computational complexity of processing CFGs hinders their deployment on IoT devices.

Converting the assembly code to Intermediate Representation (IR) code has been adopted to handle syntax differences to perform cross-architecture analysis [13]. However, available IR languages/platforms are limited to handle only a few architectures (i.e., MIPS, ARM, x86), which are not suitable for our dataset that contains malware with more diverse architectures.

IoT malware dataset. Our IoT malware dataset is provided by IoTPOT team, including two recently-collected datasets: one is collected within a three-month period between May 2016 and August 2016, and contains $1,150$ malware samples/binaries; and the other one is collected within a one-year period between October 2016 to October 2017, containing $4,000$ malware samples/binaries. Every sample has a MD5 name and a time label. To the best of our knowledge, this IoT malware dataset is the largest dataset currently available. To date, there are around $7,000$ IoT malware samples targeting smart devices as reported by Kapersky [2]. Therefore, we believe the research on $5,150$ malware set (74% of total amount) can faithfully reveal the characteristics of most IoT malware. All the malware binaries are *Linux Executable and Linkable Format (ELF) format* executable files. Figure 1 shows the diverse CPU architectures of the malware samples in our dataset, where ARM and MIPS are two most popular architectures for IoT malware. The detection rate of IoT malware is known to be low [14]. Therefore, *an accurate and lightweight cross-architectural detection mechanism that can be deployed on resource-constrained IoT devices is a pressing need.*

Malware statistical, string Features, and string obfuscation/encryption. As mentioned earlier, this work aims to develop lightweight IoT malware signatures, which implies the features used for generating the signatures should be easy to extract, and also the extracted features can differentiate between malicious and benign samples. In this work, we consider statistical and string features for clustering and signature generation of IoT malware families. Table I presents

Files	Redirect	Arithmetic	Logical	Transfer	Total
Benign-1	108719	23117	41817	300822	647654
Benign-2	129662	25204	57020	371085	767334
Benign-3	166767	36143	64249	487340	1025432
Malicious-1	10434	1744	2085	17087	45831
Malicious-2	13283	2104	2427	22513	58558
Malicious-3	5354	3707	408	363	31843

Table I: Number of instructions for benign and malicious binaries

the statistical features extracted from exemplar benign and malicious files. It shows a significant difference between the code statistics features of benign and malicious files.

We also extract printable strings from the malware samples, and discover that the extracted printable strings of many malware samples contain the same string sequences, yet, they are compiled for different architectures. For instance, our experimental results show that the same printable strings such as *"busybox iptables -A INPUT -p tcp –destination-port 7547 -j DROP"* appeared in different versions of Mirai for different architecture types, such as MIPS, ARM, PowerPC and Renesas SH. This implies that IoT malware is developed to infect multiple architectures. Also, we observe printable strings contain other rich information that can be used to distinguish between different malware families (see Section III-C). Such observation and the fact that printable strings are easy to extract motivate us to consider printable strings for signature generation.

Some printable strings from malware samples are obfuscated or encrypted. However, we find that the encrypted/obfuscated printable strings of some malware samples can also be overlapped, if these samples use the same encryption/obfuscation mechanisms. For instance, we find many samples from the same malware family contain the same encrypted/obfuscated sequences "eGAIM aJPMOG qCDCPK oMXKNNC uKLFMUQ", which can be used as signatures to identify samples from the same malware family.

III. Signature Generation of IoT Malware

In this section, we pesent the design of our system that aims to generate signatures for classifying and detecting IoT malware. The proposed system consists of two major phases: *offline signature generation* and *online detection/classification*. The offline signature generation takes IoT malware samples as input, which includes the following five steps, as presented in Fig. 2: 1) malware preprocessing, 2) coarse-grained clustering, 3) fine-grained clustering, 4) cluster merging, and 5) signature generation for online detection.

A. System Overview

The offline signature generation can be conducted at computationally rich clouds/hubs. The malware preprocessing removes non-binary files from the dataset, and disassembles all the binary files using IDA Pro [15] to retrieve their assembly codes. After preprocessing, the number of malware samples is reduced to 4,078. We propose three stages of clustering, including coarse-grained clustering, fine-grained clustering, and cluster merging. The coarse-grained clustering utilizes statistical features, while the fine-grained clustering clusters the malware samples based on their structural similarities computed using Bindiff. The combination of coarse-grained and fine-grained clustering allows us to decrease the computational cost of the clustering process, compared to using only fine-grained clustering. Cluster merging refines the clustering results by merging clusters based on the similarity of extracted string features in an iterative manner. The cluster merging allows us to attain more generic malware signatures, thus improving the malware detection rate. Finally, we generate a succinct signature by integrating string and statistical features for each malware cluster, which can distinguish between different malware clusters. We use string and statistical features due to their capability in producing distinguishable patterns and ease of extraction. Online detection/classification can be conducted on IoT devices by matching the signatures. In the following sections, we describe the system components.

B. Clustering IoT Malware

Instead of operating over each individual malware sample, we cluster these samples into groups, and perform group-level analysis to reduce the computational costs. In the end, the malware clustering contains three phases: coarse-grained clustering, fine-grained clustering, and cluster merging.

Coarse-grained clustering: After generating the assembly codes of the malware files, statistical features are extracted for each malware file. The statistical features are then normalized, and used to perform the coarse-grained clustering. We extract 8 high-level statistical features from the assembly codes, including: total number of functions, total number of instructions, number of redirect instructions, number of arithmetic instructions, number of logical instructions, number of transfer instructions, number of segments, and number of call instructions. The high-level code statistics features are resilient to cross-architecture variations, as they abstract away the different code syntax.

The average and standard deviation values of these statistical features are computed for each malware sample. Among the 8 statistical features, we select the statistical features that can distinguish between different clusters with low standard deviations. In the end, we retain 6 statistical features by discarding the number of segments and number of call instructions due to their low distinguishable capability.

Finally, we use *K-means clustering* to perform coarse-grained clustering. K-means is selected due to its high efficiency. The number of clusters K is determined and validated using *Davies–Bouldin (DB) cluster validity index* [16], which is a standard metric for evaluating cluster results. A lower DB index denotes a better separation of the clusters and the better tightness inside the clusters. Therefore, the number of clusters with lowest DB index is selected as the best K for coarse-grained clustering.

Fine-grained clustering: In the fine-grained clustering phase, we consider code structural similarity between malware binaries computed using Bindiff. We iteratively compute the pairwise structural similarity between every malware pair within a cluster derived from coarse-grained clustering. The intuition is that high-level structural similarity generated by Bindiff is more resilient to cross-architecture variations [17],

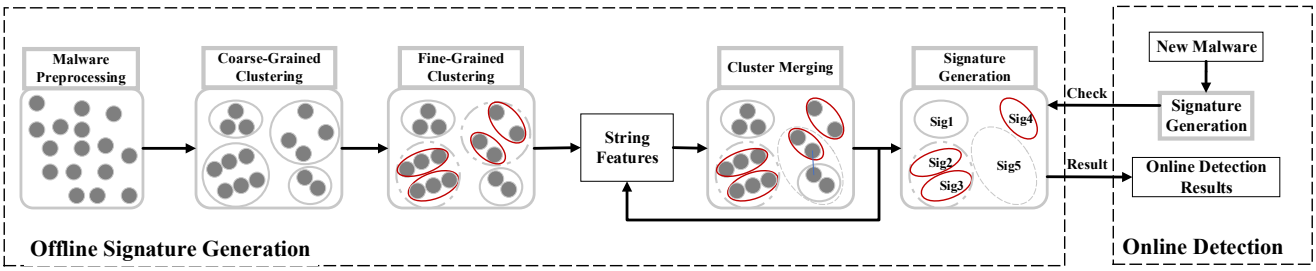

Figure 2: Detection system architecture

and can attain high accuracy in matching functions. This process yields an $N \times N$ similarity matrix (N is the number of malware samples), where the values in each row represent the similarity scores computed by Bindiff for a single malware sample against all other malware samples in the dataset.

Here, we utilize a popular binary similarity analysis tool, Bindiff [10], for computing the similarity between the malware samples in our dataset. Bindiff is a popular tool for computing structural similarity. It reconstructs Control Flow Graph (CFG) of each binary file to perform function and basic block matching, and then compares functions and basic blocks by extracting the graph-based features such as number of incoming/outgoing edges, the position of basic blocks in the CFG, etc. In order to tolerate the code differences brought by cross-architecture compilations, Bindiff abstracts the structural features of a binary file, while ignoring the specific assembly-level instructions. It retains a trade-off between similarity analysis accuracy and efficiency by applying a multi-level matching strategy based on function and basic block level structural attributes. The final result of Bindiff is a list of matched and unmatched functions from both binaries, based on which the similarity score (ranging in $[0, 1]$) is computed [10]. Bindiff is fairly efficient, and the matches it produces are proven accurate [17]. Therefore, in this clustering stage, we assign a relatively large cluster number to ensure every cluster is compact enough to contain all the similar malware samples.

In the fine-grained clustering, we partition the malware samples within each coarse-grained cluster into multiple fine-grained clusters using the single-linkage hierarchical clustering. We choose hierarchical clustering, because of its ability to find clusters of arbitrary shapes with arbitrary distance metrics. The hierarchical clustering takes a matrix of pairwise distances among malware samples and generates a *dendrogram*, which is a tree-like data structure to represent the clustering outcome. Dendrogram cutoff determines the number of clusters. Distance measurement is critical for performing hierarchical clustering, and we derive the distance measurements through binary similarity analysis, which identifies common characteristics of malware samples at different levels including basic block, function, and file levels.

Rather than converting the similarity score to distance measurement which may lose accuracies, we propose to utilize similarity scores as features. The rationale lies on the fact that similar malware samples will have comparative similarity scores with other samples. Therefore, among N malware samples, each sample will have N similarity scores as a feature vector, which contains the malware's similarity score with itself (i.e., 100%) and with all other malware samples in the dataset. In order to minimize the impact of this high and ineffective self-similarity score, for malware A, we replace this self-similarity score with the highest similarity score of malware A compared with all other samples. Using similarity scores as features, we compute the distance measurements using Euclidean distance. The hierarchical clustering is conducted using the calculated distances among all samples. Similar to the coarse-grained clustering, the best number of fine-grained cluster is determined using DB cluster validity index for every coarse-grained cluster.

C. Cluster Merging

After splitting the coarse-grained clusters into the fine-grained clusters, some of the generated clusters may actually share a high similarity, which can then be merged together. Consequently, we use string features to merge clusters and refine clustering results. The cluster refinement involves two steps: 1) string feature analysis: N-gram text analysis is used to extract distinguishable string features from the printable strings; and 2) merging clusters: clusters are merged based on the similarity analysis of string features.

String feature analysis: The goal of string feature analysis is to find the string features that represent all malware samples of each cluster, since string features exhibit *rich contextual information that is suitable for fine-grained analysis*. String feature analysis facilitates the detection of cross-architecture IoT malware, as printable strings likely remain the same for binaries even when they are compiled differently. To this end, printable strings are extracted from each malware sample, which are used as inputs to generate N-gram string vectors, capturing the sequential order of strings. Punctuation marks, such as ":", "//", ";", are used to segment the strings. For example, a string vector "wget http://198.12.97.79/bins.sh; chmod 777 bins.sh; sh bins.sh" is segmented into "wget http 198.12.97.79 bins.sh chmod 777 bins.sh sh bin.sh". In order to avoid the overfitting issue, we replace the ip address with a special word "reIPaddress". Using N-gram, we are able to extract meaningful word sequences, e.g., "wget http reIPaddress bins.sh", "chmod 777 bin.sh sh" (with 4-gram). For improving the effectiveness of N-gram string feature, we remove any printable strings that contain less than three characters. We select N value according to experimental analysis performed

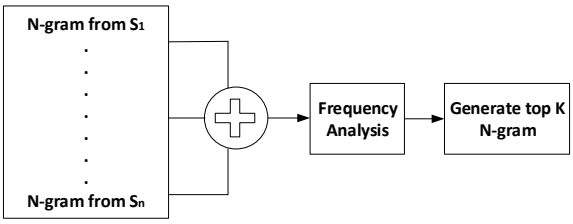

Figure 3: String feature analysis (S_1,$\cdots$, S_n are n samples in a cluster)

to retain a balance between signature matching accuracy and efficiency, as described in Section IV-B.

Fig. 3 illustrates the string feature analysis. For each malware sample, we extract the N-gram string vectors, and remove the duplicate N-gram string vectors. Then, we combine the generated N-grams of all malware samples within a single cluster. Frequency analysis is carried out to identify the most common N-gram string vectors in each cluster. Consequently, we generate string features for each cluster, consisting of the top K N-gram strings with the highest appearances among all the N-gram strings in the combined string vector. The selection of K is explained in Section IV-B, which aims at identifying the distinguishable N-gram strings while minimizing computational costs. Thus, the top K N-gram strings are used as features of each cluster to perform cluster merging.

Algorithm 1: Cluster Merging

Required: $\tau = merging\ threshold$, $|C|$ is the current number of clusters, JS represents Jaccard Similarity. c_i, c_j are two clusters under evaluation, and $SF(c_i)$, $SF(c_j)$ are their top K N-gram string features, respectively.

```
for (each pair of c_i, c_j) do
    if (JS(SF(c_i), SF(c_j)) ≥ τ) then
        Merge clusters c_i, c_j;
        |C| = |C| − 1;
        Generate new string feature for every cluster;
    end
end
```

Merging clusters: Cluster merging is performed iteratively to refine the clustering results, which guarantees the malware clusters are both compact and well-separated from each other. Algorithm 1 illustrates the cluster merging procedure. After generating the top K N-gram strings for each cluster, we use Jaccard similarity to compute the *inter-cluster similarity* scores between different clusters based on the generated top K N-gram strings for each cluster. Clusters with a high string similarity score (i.e., higher than a predefined *merging threshold*) will be merged. After merging clusters, we evaluate the merged clusters again by recomputing their top K N-gram string features, and the corresponding inter-cluster similarity scores. The cluster merging process can be iterated multiple times in sequence until different clusters bearing low similarities (i.e., lower than the merging threshold), which indicates a set of well-separated and distinguishable clusters. After performing cluster merging, we can perform lightweight signature generation for each cluster.

D. Signature Generation and Online Detection/Classification

The complete signature of each merged cluster contains the top-K N-gram string feature (SF) extracted using the aforementioned method, and statistical feature (ST) that represents the average values of each statistical feature. Both SF and ST can be easily extracted from malware files, making the signature generation fairly efficient.

The online detection/classification of IoT malware is a signature matching process, where the matching is performed by computing the similarity between the extracted signature from the suspicious file and a set of cluster signatures. *Euclidean distance* $d(ST_i, ST_j)$ is used for measuring similarity of the ST. To facilitate similarity analysis, we convert the Euclidean distances into similarity scores [18], named as Statistical Similarity (SS) and formalized as follows:

$$SS(ST_i, ST_j) = \frac{1}{1 + d(ST_i, ST_j)}. \quad (1)$$

On the other hand, *Jaccard Similarity (JS)* is used for computing similarity of the SF, denoted as $JS(SF_i, SF_j)$. The *Overall Similarity (OS)* score for the signature matching will be a weighted sum of JS and SS, written as follows:

$$OS(i, j) = w_1 \cdot JS(SF_i, SF_j) + w_2 \cdot SS(ST_i, ST_j), \quad (2)$$

where $w_1 + w_2 = 1$, and $OS(i, j)$ represents the signature matching score between file i and cluster j. In this research, we give a equal weight (i.e., 0.5) to SF and ST by default, but the weight can be tuned according to analysis results, e.g., if string obfuscation is identified, a higher weight can be assigned to ST. After computing the OS scores between a file and all cluster signatures, we identify the highest OS score as the file's suspicion level. Meanwhile, the file can be classified into the corresponding malware cluster or marked as benign. In our experiment in Section IV, we show that the file's suspicion level can be either a high value or a low value, which makes it straightforward to classify a suspicious file as IoT malware or benign samples.

The YARA tool [19] can be used for generating static signatures based on a sequence of specific printable strings or bytes belonging to a malware family. YARA signature for Mirai malware is shown in Listing 1, which includes the printable string signature. While most YARA signatures are manually identified, the proposed system can facilitate the automated identification of string signatures for YARA. Therefore, we can easily incorporate YARA signatures in our system for malware detection.

```
rule Mirai_1
{
  meta:
    description = "Mirai Variant 1"
    author = "Mohannad / @moh"
    date = "2017-04-16"
  strings:
      $dir1 = "/dev/watchdog"
```

```
        $dir2 = "/dev/misc/watchdog"
    condition:
        $dir1 and $dir2
}
```

Listing 1: YARA Signature for Mirai

IV. Evaluation

In this section, we evaluate the performance of our solution using the IoT malware dataset. We first discuss our methodology for selecting the values of K and N, and then evaluate the multi-stage clustering. The malware detection performance is evaluated in terms of malware detection rate and false positive rate. We further evaluate our system's performance in classifying novel malware samples in the testing dataset, and benign linux firmware gleaned from various commercial product websites. Since we have around $4,000$ IoT malware samples after disassembling malware in our dataset, we select the older $2,000$ samples as the training dataset to generate signatures, and use the newer $2,000$ samples (regarded as novel malware) as the testing dataset to evaluate the detection capability of the proposed system.

A. Selecting Parameters K and N

Similar to other empirical approaches for selecting N-gram parameter [20], [21], we also adopt an experimental approach driven by the data to select the best set of parameters. In our approach, we use a set of values {90, 100, 150, 200, 250} for selecting K, and a set of values {3, 4, 5, 6, 7} for selecting N, which are the key parameters for defining top K N-gram string feature. Our intuition is that the best K and N pair should produce the best clustering results, i.e., the samples in a cluster should bear high similarity with each other, and different clusters should be well separated. Thus, different combinations of K and N are considered with the goal of maximizing the similarity within the same cluster *(Intra-cluster similarity)*, and minimizing the similarity among clusters (*Inter-cluster similarity*). Specifically, the inter-cluster string similarity is defined as the average Jaccard similarity of string features among different clusters. For intra-cluster string similarity, we collect the top-K N-gram string features of all malware samples inside each cluster, and compute the average Jaccard similarity of cluster string features with its enclosed samples.

Determining best K: We examine the *inter-cluster* and *intra-cluster* string similarity with different K values and fixed N value in order to determine the best K. Fig. 4 shows the string similarity results with different K values (when N is fixed as 4), from which we can see there is a slight increase of both inter-cluster and intra-cluster string similarity with the increase of K. To strike a balance between the intra-cluster similarity (the higher the better) and the inter-cluster similarity (the lower the better), we measure the difference between inter and intra-cluster string similarity, i.e., the gap between two lines, and select the K with the maximal gap. In the end, we select the best $K = 100$. For validation, we evaluate the performance by fixing N as other values, and $K = 100$ always performs best. This process can be fully automated by measuring the difference between inter-cluster and intra-cluster similarity.

Table II: Inter-cluster string similarity with different N values (K=100)

N	3	4	5	6	7
Inter-Cluster	0.162	0.120	0.171	0.162	0.164

Table III: Fine-grained clustering results with best DB index values

Coarse Cluster Index	1	2	3	4	5	6
Best DB Index	0.58	0.6	0.49	0.607	0.56	0.6
# of Fine Clusters	13	17	6	31	3	47

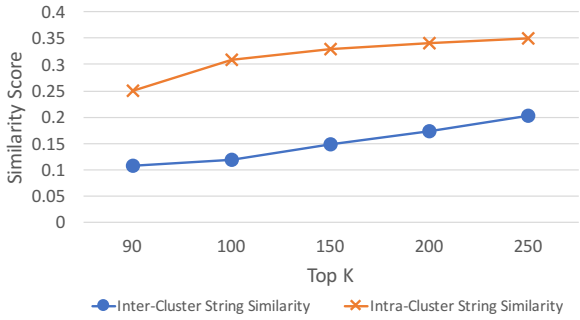

Figure 4: Inter-cluster and Intra-cluster string similarity with different K values ($N = 4$)

Determining best N: By fixing the value of $K = 100$, we use different N values to evaluate the inter-cluster string similarity. Table II shows $N = 4$ yields the lowest inter-cluster similarity. Note that we omit the measurement of intra-cluster string similarity to reduce the computational costs. Thus, we select the best N as 4. Finally, the string feature of each cluster is generated based on the ***top-100 4-gram*** string vectors. The selection of N is also an automated process by measuring the inter-cluster similarity w.r.t. different N values.

B. Evaluating IoT Malware Clustering

Coarse-grained clustering is conducted through K-means clustering mechanism. Fig. 5 presents the DB index values for different number of clusters in coarse-grained clustering, where we can see DB index value reaches the lowest value, i.e. 0.77, when the cluster number is 10. Fine-grained clustering utilizes hierarchical clustering mechanism, and we select the Dendrogram cutoff based on DB cluster validity index. In the end, we obtain 153 fine-grained clusters by splitting 10 coarse-grained clusters. A partial fine-grained clustering result is shown in Table III with corresponding best DB index values, which shows the number of fine-grained clusters produced by each coarse-grained cluster. The average Bindiff similarity for samples within a cluster is 95.46%, demonstrating the samples within the fine-grained cluster do have high structural similarity.

Evaluating cluster merging using string feature: For merging clusters, we compute the similarity scores using cluster string features. Recall that the cluster string feature represents the top-100 4-gram string vectors of a cluster. Two clusters will be merged, if the Jaccard similarity score of their cluster

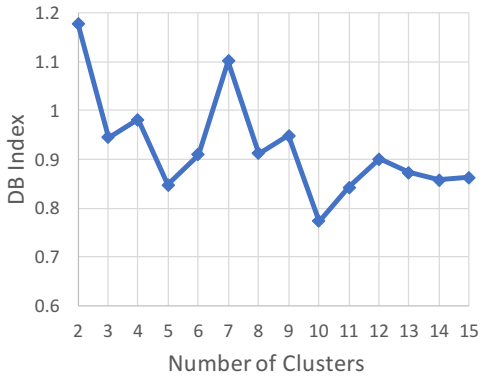

Figure 5: DB cluster validity index w.r.t different numbers of coarse-grained clusters

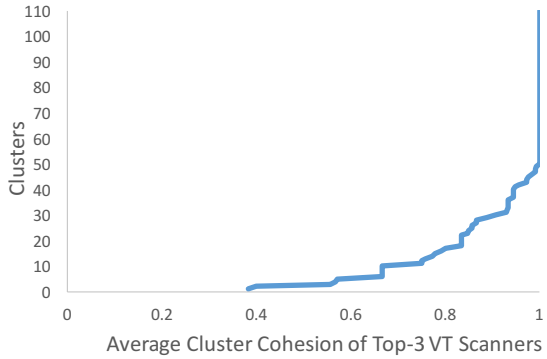

Figure 6: Distribution of average cluster cohesion

string features is higher than a *merging threshold*. The merging threshold should be set sufficiently high to avoid merging dissimilar clusters, but should not be set too high that may prevent appropriate cluster merging. In this paper, we empirically set the merging threshold as 0.7 to merge clusters that resemble each other [22]. In the end, 153 original clusters are merged into 110 clusters, which are re-evaluated to make sure they cannot be further merged.

Cluster cohesion: We further evaluate the compactness of our clustering approach based on VirusTotal detection results. We select top-3 scanners that have high detection rate including AVG, DrWeb, and Sophos. We define *cluster cohesion value* as ratio of the number of malware files that is assigned the same malware family by a scanner versus the total number of malware files in each cluster. The *average cluster cohesion value* is computed by averaging the cluster cohesion values across the top-3 scanners. The malware families are determined by removing the last section of the generated label string, which is separated by the *dot* (.) symbol. For example, *Linux.BackDoor.Fgt* is the malware family corresponding to the following labels generated by DrWeb, e.g., *Linux.BackDoor.Fgt.373, Linux.BackDoor.Fgt.578, Linux.BackDoor.Fgt.11, Linux.BackDoor.Fgt.229*.

The distribution of average cluster cohesion is presented in Fig. 6, which shows 70% of the clusters have average cluster cohesion higher than 0.9. Therefore, we can consider the generated clusters as compact, thus confirming the validity of our clustering process.

IP addresses in the strings: The malware authors usually hard-code the IP addresses of Command-and-Control (C&C) servers in the malware code. Therefore, a specific IP address can be a strong indicator of maliciousness. We extract all IP addresses in the strings of the malware binaries. Among all the 2,000 malware samples, there are 923 unique IP addresses. We assign the 110 cluster labels to the malware samples. We have some interesting findings, e.g., we find that the malware samples in some clusters do not contain any IP addresses. The samples in some clusters contain unique IP addresses. For example, all the samples in cluster 5 contain the same IP address $185.47.61.167$. Moreover, the samples in many clusters contain multiple IP addresses, which means the adversaries have multiple C&C server IP addresses in possession, or they modify the C&C server IP addresses to avoid being taken down. For these clusters, we discover that on average, one IP address is assigned to *12 malware samples*. The architecture types of 12 malware samples usually contain all the 8 identifiable types as shown in Fig. 1. The adversaries are obviously using automation tools to automate the binary generation processes, so that they can streamline the creation of malware samples for different IoT devices.

The matching of IP addresses with our clustering result implies unrigorously that our system is capable of clustering cross-architecture malware samples, as we can group together the samples using the same IP addresses but with different architectures. It also aligns with a common belief that IP addresses alone are not sufficient enough to correctly classify malware samples.

C. Evaluating Signature Detection

The cluster signature will include the string feature and statistical feature, generated using our IoT malware dataset. In this section, we evaluate the detection accuracy and effectiveness of our cluster signature. We use the cluster signature to detect the samples inside the cluster. A malware sample is detected if its maximum OS value (i.e., overall similarity in Eq. 2) is higher than a *detection threshold*. The detection threshold is empirically set as a high value (i.e., 0.7) to reduce false positive rate. The detection rate is defined as the ratio of the number of detected malware samples versus the total number of samples. We first perform a 10-fold cross validation using the $2,000$ malware samples in the training dataset. On average, our system achieves a detection rate of 95.5%.

To further evaluate clusters' signatures generated through our IoT malware dataset, we also download a set of benign firmware binaries from openwrt. Furthermore, we use the $2,000$ novel malware samples in the testing dataset to evaluate malware detection performance. All the tested files are in Linux ELF format. The benign firmware dataset contains 130 samples, while the testing dataset contains $2,000$ new malware samples. We generate the string and statistical features for these benign and malware samples.

Then, we match the string features of benign firmware and malware with the cluster signatures. Out of 130 benign samples, 0 sample is misidentified as malware. Therefore, the

false positive rate of our detection system is 0%. It proves that our cluster signature has excellent performance in classifying IoT malware. For the $2,000$ novel malware, we can reach 85.2% detection rate. We manually examine the samples that are not detected. These samples contain completely distinct characteristics compared with the samples in the training sets, which means these samples are unseen malware with new signatures, for which we need to generate new signatures to cover them.

Performance comparison: We also conduct performance comparisons with two existing works based on API call sequences [23] and operation code (OpCode) N-grams [20]. The results are shown in Table IV, and our detection method can achieve significant performance improvement by capturing the unique characteristics of IoT malware.

Table IV: Performance comparison

Methods	Detection Rate	False Positive Rate
API Calls [23]	64.8%	5.1%
OpCode N-gram [20]	66.0%	10.0%
Our solution	**85.2%**	**0%**

D. Evaluating Runtime Performance

Table V presents the average processing time required for processing and generating the clusters, from which we can see the coarse-grained clustering can complete the clustering of $2,000$ malware samples within 10 ms. The cluster merging process takes a longer time period compared with coarse-grained and fine-grained clustering. This is expected since the merging process performs the iteration several times until no similarity score exists above the threshold, and in every iteration N-gram string analysis is performed again. We also evaluate the runtime performance of our signature matching mechanism on an ARM platform using QEMU [24]. We run experiments inside an emulator with ARM Cortex-A9 CPU (0.8GHz to 2GHz) and 256MB of RAM. The running time is $15ms$ for matching/classifying a new binary file. Although many IoT devices have lower configurations than our emulator, we believe the running time performance indicates the efficiency of our mechanism. In future, we will evaluate our mechanism on real IoT devices.

Table V: Average cluster processing time (second)

Coarse-Grained Clustering	Fine-Grained Clustering	Cluster Merging
0.01	2.48	2.85

V. Limitations and Future Research

Evasive techniques: The proposed approach for generating lightweight detection signatures utilizes printable strings as string features. As a result, our approach is susceptible to string obfuscation and encryption techniques. To date, the IoT malware samples are still very premature compared with their sophisticated PC counterpart, and we have not observed any IoT malware employing sophisticated code obfuscation/encryption techniques that complicate the malware detection. We believe the proposed system addresses a timely need to provide an effective first-line defense on IoT devices and achieves a high detection rate, and it also contributes to a better understanding of IoT malware. On the other hand, we envision that the sophistication level of IoT malware will likely grow. In this case, we will incorporate dynamic analysis [25] to cope with such sophisticated malware samples, since malware samples will decipher the string contents at runtime.

Unknown malware: Another caveat is that our system *may not be able to deal with unknown IoT malware that bears no similarity with existing IoT malware*, so that we need to constantly update the signatures as new IoT samples are discovered. But the update can be conducted offline on a computationally rich hub. For instance, we can generate updated signatures everyday to keep up the rapid change of IoT malware. Moreover, as shown in [26], the structural features suffer from the variations caused by different compilation options, which may induce false classification results. We plan to incorporate behavioral features to enhance the structural analysis of BinDiff. One may argue that our mechanism only works on our specific IoT malware dataset, however, the proposed mechanism is indeed general as it can work on new IoT malware with updated signatures.

Threshold setting: There are multiple thresholds in our system, including: merging threshold (set as 0.7), sample matching threshold (set as 0.9), and detection threshold (set as 0.7). Currently, we use empirical approaches to identify an appropriate threshold by evaluating over two opposite training datasets, e.g., one contains the malware belonging to a family called "insider", and one contains the samples outside the family called "outsider". The minimum value of all insiders and maximum value of all outsiders can be identified, and the threshold is set to the mean of these two values that optimizes the separation of the two different worlds. This process can be automated to relieve the manual burden. It is evident that for different datasets, we may need to adjust these threshold accordingly. As a future work, we will investigate the deployment of online detection system to push new signatures and update the detection methods.

VI. Related Work

In this section, we focus on reviewing malware analysis approaches that aim at classifying malware families and generating signatures for effective detection.

Malware Classification: Recently, Alazab [23] proposes a Windows malware (in PE format) classification method based on features extracted dynamically and statically from the malware files, including windows API sequences and their frequencies of appearance. But the API calls are different across different architectures, and can be easily forged or modified by attackers to disguise their malicious activities. Santos et al. [27] present a malware classification method based on the frequency of opcode sequences, but opcode sequences can also be easily disrupted by simple code variations resulted from different compilation options. Zynamics Bindiff [8] and BinSlayer [17] measure binary similarities based on graph isomorphism between CFGs. BinSlayer further improves the binary comparison accuracy of BinDiff by incorporating graph edit distances, but also brings considerable overhead. Both Shabtai et al. and Hu et al. [20], [28] use static analysis to

examine the effectiveness of malware detection using OpCode N-gram analysis. Kong et al. [29] map malware instances to their corresponding malware family using structural features of function call graph and statistical features including lists of API calls and opcodes with their respective frequencies.

Malware Signature Generation: High level string features and statistical features extracted from file size and file content have been used to classify firmware images of embedded devices [30]. Besides our different goals (known firmware classification versus unknown malware classification), they use a simple intersection method on string features to identify firmware images, while our N-gram string features can extract more representative features for each malware family. We also consider statistical features by counting instructions and functions at the assembly code level, which have finer granularity. Perdisc et al. [22] propose a multi-stage clustering approach for generating malware signatures using the network traffic generated by malware samples. FIRMA [31] also utilizes network traffic to generate behavioral signatures for malware detection. Unlike [22], FIRMA generates network signatures for each network behavior regardless of traffic types, the format of which follow popular signature-matching IDS. While the previous work deals with generic PC malware, we focus on the newly emerging IoT malware.

VII. Conclusion

This paper investigated the emerging IoT malware detection problem, and proposed an efficient signature generation and classificastion mechanism for cross-architecture IoT malware. Based on static analysis, the proposed mechanism utilizes string, statistical and structural features for classifying IoT malware, where Bindiff is used for computing structural similarities, and N-gram printable string vectors and statistical features are extracted for characterizing the malware families. The experimental results show the effectiveness and efficiency of our signature generation system. In future, we will further investigate the IoT malware using dynamic analysis and develop robust detection tools for better protecting IoT systems.

Acknowledgement

This work was supported in part by the US National Science Foundation under grants CNS-1566388, CNS-1717898, and CNS-1731833. This work was also supported by the National Natural Science Foundation of China under Grants No.61672262.

References

[1] Gartner, "Gartner says 8.4 billion connected things will be in use in 2017, up 31 percent from 2016," http://www.gartner.com/newsroom/id/3598917, Feb. 2017. Accessed at Feb 21, 2017.

[2] "Amount of malware targeting smart devices more than doubled in 2017," https://www.kaspersky.com/about/press-releases/2017_amount-of-malware-targeting-smart-devices-more-than-doubled-in-2017, 2017, accessed at July 21, 2017.

[3] B. Herzberg, D. Bekerman, and I. Zeifman, "Breaking Down Mirai: An IoT DDoS Botnet Analysis," https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html, Accessed at Feb 21, 2017.

[4] S. Sicari, A. Rizzardi, L. A. Grieco, and A. Coen-Porisini, "Security, privacy and trust in internet of things: The road ahead," *Computer Networks*, vol. 76, pp. 146–164, 2015.

[5] S. Eschweiler, K. Yakdan, and E. Gerhards-padilla, "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code," in *Proc. of NDSS*, February 2016, pp. 21–24.

[6] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. of CCS*, 2016, pp. 480–491.

[7] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proc. of USENIX Security*, 2014, pp. 95–110.

[8] H. Flake, "Structural comparison of executable objects," *DIMVA 2004, July 6-7, Dortmund, Germany*, 2004.

[9] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proc. of CCS*, 2009, pp. 611–620.

[10] "BinDiff Manual," https://www.zynamics.com/bindiff/manual/, Accessed at July 21, 2017.

[11] Y. M. Pa Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "IoTPOT: Analysing the rise of IoT compromises," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., 2015.

[12] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 709–724.

[13] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proc. of FSE*, 2016, pp. 678–689.

[14] "Google's VirusTotal puts Linux malware under the spotlight," http://www.zdnet.com/article/googles-virustotal-puts-linux-malware-under-the-spotlight/, 2014, accessed at July 21, 2017.

[15] "Ida," https://www.hex-rays.com/products/ida/, Accessed at Feb 21, 2017.

[16] A. K. Jain and R. C. Dubes, *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[17] M. Bourquin, A. King, and E. Robbins, "Binslayer: Accurate comparison of binary executables," in *Proc. of PPREW '13*, 2013, pp. 4:1–4:10.

[18] T. Segaran, *Programming Collective Intelligence*. O'Reilly, 2007.

[19] "Welcome to YARA's documentation," http://yara.readthedocs.io/en/v3.7.0/index.html, Accessed at Dec 21, 2017.

[20] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, "Detecting unknown malicious code by applying classification techniques on opcode patterns," *Security Informatics*, vol. 1, no. 1, p. 1, 2012.

[21] E. B. Karbab, M. Debbabi, and D. Mouheb, "Fingerprinting Android packaging: Generating DNAs for malware detection," *Digital Investigation*, vol. 18, pp. S33–S45, Aug. 2016.

[22] R. Perdisci, W. Lee, and N. Feamster, "Behavioral clustering of http-based malware and signature generation using malicious network traces," in *Proc. of NSDI*, 2010.

[23] M. Alazab, "Profiling and classifying the behavior of malicious codes," *Journal of Systems and Software*, vol. 100, pp. 91–102, 2015.

[24] "Qemu: the fast processor emulator," http://www.qemu-project.org/, Accessed at Feb 21, 2017.

[25] D. D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards fully automated dynamic analysis for embedded firmware," in *Proc. of NDSS*, February 2016, pp. 21–24.

[26] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *Proc. of USENIX Security*, San Diego, CA, 2014, pp. 303–317.

[27] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-sequence-based malware detection," in *Engineering Secure Software and Systems: Second International Symposium, ESSoS 2010*, 2010, pp. 35–43.

[28] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "Mutantx-s: Scalable malware clustering based on static features." in *USENIX Annual Technical Conference*, 2013, pp. 187–198.

[29] D. Kong and G. Yan, "Discriminant malware distance learning on structural information for automated malware classification," in *Proc. of SIGKDD*, 2013, pp. 1357–1365.

[30] A. Costin, A. Zarras, and A. Francillon, "Towards Automated Classification of Firmware Images and Identification of Embedded Devices," in *32nd International Conference on ICT Systems Security and Privacy Protection (IFIP SEC)*, May 2017.

[31] M. Z. Rafique and J. Caballero, "Firma: Malware clustering and network signature generation with mixed network behaviors," in *Proc. of RAID*, 2013, pp. 144–163.