Hindawi
Security and Communication Networks
Volume 2020, Article ID 8810708, 12 pages
https://doi.org/10.1155/2020/8810708

Research Article

Automatic Analysis Architecture of IoT Malware Samples

Javier Carrillo-Mondejar, Juan Manuel Castelo Gomez, Carlos Núñez-Gómez, Jose Roldán Gómez, and José Luis Martínez

Research Institute of Informatics (I3A), Universidad de Castilla-La Mancha, Albacete 02071, Spain

Correspondence should be addressed to Javier Carrillo-Mondejar; javier.carrillo@uclm.es

Received 30 March 2020; Revised 16 July 2020; Accepted 7 October 2020; Published 26 October 2020

Academic Editor: Yin Zhang

The weakness of the security measures implemented on IoT devices, added to the sensitivity of the data that they handle, has created an attractive environment for cybercriminals to carry out attacks. To do so, they develop malware to compromise devices and control them. The study of malware samples is a crucial task in order to gain information on how to protect these devices, but it is impossible to manually do this due to the immense number of existing samples. Moreover, in the IoT, coexist multiple hardware architectures, such as ARM, PowerPC, MIPS, Intel 8086, or x64-86, which enlarges even more the quantity of malicious software. In this article, a modular solution to automatically analyze IoT malware samples from these architectures is proposed. In addition, the proposal is subjected to evaluation, analyzing a testbed of 1500 malware samples, proving that it is an effective approach to rapidly examining malicious software compiled for any architecture.

1. Introduction

The appearance of the Internet of Things (IoT) has greatly improved the application of technology in the everyday lives of people. Years ago, digital interaction between an individual and technology was in general only through a computer. With the development of smartphones, that communication became a more mobile, personal, and continuous task. And then, the IoT appeared to change all the previous concepts and insert technology into almost every imaginable object. Smart houses, eHealth, or smart cities are just a few examples of contexts that have their origin in the application of the IoT. Thus, not only has it helped to complement existing scenarios but it has also given rise to the ones in which technology is applied.

As a consequence, the volume of data that is now digitally handled has vastly increased as well. However, although the emergence of the IoT has clearly benefited people, the same positive verdict cannot be passed when speaking of the security measures implemented on the devices. Unfortunately, developers opted to prioritize usability over security, especially during the IoT's conception, when the thought of someone compromising an entire network by simply attacking a switch was unthinkable.

Therefore, there was a huge underestimation of the requirements that these devices and the information that they handle demand. Nowadays, this issue is being acknowledged, and companies are working on improving the protection, but they are still quite vulnerable, added to the fact that a great number of old devices is still being used. This makes the IoT the perfect environment for cybercriminals to operate in. They can gain access to very sensitive and valuable information with little effort. Recent studies [1] show the magnitude of the problem. Only in the first quarter of 2019, a hundred million attacks were detected on smart devices, a figure seven times greater than the number found in 2018. Unsurprisingly, the *Mirai* malware family was behind 39% of them, taking advantage of old devices with unpatched vulnerabilities. Another sample which exploits a trivial attack, namely, the brute-force, *Nyadrop*, closely followed *Mirai* and reached a percentage of 38.57%.

These attacks were the result of poorly designed security measures on the devices and could have been easily mitigated by just changing the default user and password of the device for a more secure one. Instead, they ended up affecting companies such as Twitter, Amazon, Spotify, and Netflix, costing them millions of dollars and affecting their customer's trust [2].

As mentioned above, most IoT attacks do not have their origin in new malware samples, but are based on previous ones that were successful. New versions of old attacks appear every day with minor modifications, but the way they work remains almost identical. Having information about how a sample interacts with the compromised device, and what actions it carries out, allows investigators to protect the device or, at least, limit its expansion over the network. For this reason, the ability to identify which malware samples are alike, that is, those that belong to the same family, can have a huge impact when determining what actions to be taken in order to reduce the impact of a cyberincident.

In addition, besides the existence of multiple operating systems, there are also several architectures used by IoT devices, such as ARM, PowerPC, MIPS, and x86. With the aim of expanding the range over which cybercriminals can carry out their attacks, they develop samples for more than one. This means that numerous pieces of malware have their origin in a sample, and then it is adapted to work on other architectures. Consequently, its behaviour remains similar, with only its structure varying in order to be compatible with them. This allows the malware analyst to analyze malware families independently of the architecture for which the sample was designed.

This analysis is neither a trivial task nor a speedy one. The number of existing samples, added to the appearance of new ones almost every minute, makes it impossible for an investigator to study all of them. Therefore, it is necessary to develop automatic solutions, such as architectures or frameworks, which can speed up the process and be able to examine multiple samples at once. In order to achieve that, a change of approach is needed: instead of focusing on the features that differentiate a sample, now it is mandatory to determine which characteristics allow a piece of malware to be grouped with another, as well as selecting the ones that can be collected and interpreted automatically.

Therefore, the contributions of this study are as follows:

- We study the current state of malware analysis, focusing on the development of automatic solutions to perform examinations
- We present a series of static and dynamic characteristics that are useful to automatically categorize malware samples
- We propose a modular framework for the automatic analysis and clustering of malware samples from the most widely used architectures, based on the evaluation of their static and dynamic features
- We evaluate the proposal with a testbed of nearly 1,500 pieces of malware, confirming its usefulness when analyzing and clustering samples from different IoT architectures

The rest of the paper is organized as follows. Section 2 describes the IoT's architecture, its malware threats, and how to obtain useful characteristics from them. An architecture to automatically cluster malware samples from different IoT architectures is presented in Section 3. An evaluation of the proposal through the analysis of 1500 malware samples is carried out in Section 4. Finally, our conclusions are presented in Section 5.

2. Background

As discussed in the previous section, the IoT environment is the perfect target for cybercriminals to attack. This section presents the problem related to the large number of devices with different architectures connected to the Internet, lists the reasons for the rise of IoT security threats, and defines the concepts of malware analysis and characterization. Then, the Service-Oriented Architecture (SOA) software paradigm used in the design of the framework is introduced. In addition, we present a review of the proposals from the research community in regard to this paper.

2.1. The IoT Environment. The IoT allows developers to model use cases that in the past were not feasible due to the specific limitations of traditional client-server architectures: resource centralization, expensive devices, and high latencies, among others. The IoT environment creates room for new contexts such as Industry 4.0 [3] and smart homes [4]. Its structure can be divided into three fundamental building blocks: the Cloud Layer, the Network Layer, and the Devices Layer. Figure 1 shows the hierarchy formed by these layers. Frequently, end devices interact with other IoT devices as well as with large data centers in the cloud layer to carry out the tasks (sometimes computationally intensive ones) assigned to these end devices. Accordingly, more and more end devices are exposed to the Internet every day, so it is important to adopt appropriate security measures if we do not want to expose our end devices to external attackers.

Another main problem of the IoT environment is the considerable heterogeneity of the devices that comprise it. Although it is important to define security, analysis, and clustering mechanisms against malware layer by layer, our work focuses on the constrained-resource devices of the device layer. These devices are built with different hardware specifications and run different operating systems. One of the most significant specifications is the processor architecture used by such devices. Each processor and its instruction set are designed in a specific way. For example, ARM is a more energy-usage-concerned architecture than x86-64. In our case, the proposed framework focuses specifically on modelling Intel 80386, x86-64, MIPS, ARM, and PowerPC architectures.

2.2. Threats. By scrutinizing the aforementioned recent studies focused on evaluating new trends in IoT malware, a drop in the number of attacks via Telnet can be observed for the second quarter of 2019. Now, the value almost reaches 60%, 20% less than in the previous one. This

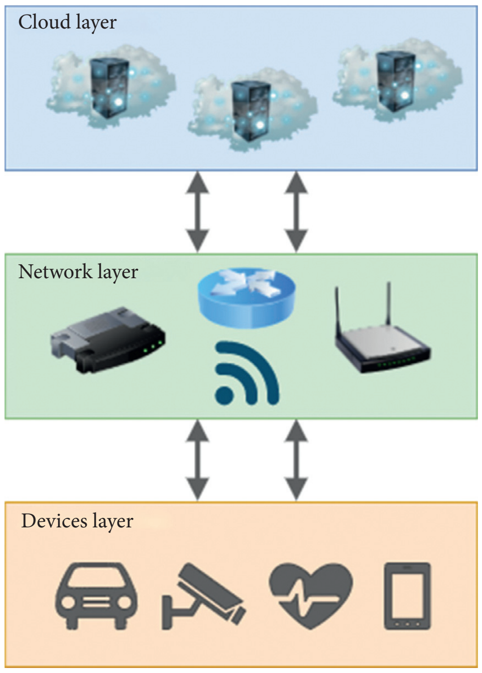


Figure 1: IoT environment architecture.

statistic can be seen as an encouraging one if we deduce that the decrease was due to developers no longer using that service, which is well-known to be deprecated and unsafe. The most worrisome data are that there are few changes in the most common malware families with respect to previous years, meaning that old attacks are still being successful. In addition, the number of malware samples is still growing and expanding into more areas [1]. Some of the main causes of the rapid growth in cybercrime in the IoT are the following:

Number of connected devices: during the year 2020, this figure is forecasted to reach 20.4 billion [5], with 5.8 billion of them being used in the enterprise and automotive market [6]. This means that there are more IoT devices than conventional ones, e.g., smartphones or computers. Therefore, it is preferable for cybercriminals to perform large-scale attacks in this environment rather than in the traditional one, as they can target more victims.

Implemented security measures: as briefly mentioned above, IoT devices can be easily compromised by carrying out simple brute-force or dictionary attacks. This is mainly due to the usage of weak default login credentials. Although it may seem ludicrous, the combination of user and password such as "admin-admin" or "admin-1234" is not that uncommon.

Data handled: the application of the IoT has led to the generation of data that previously did not exist or only did so in a smaller quantity. eHealth is a good example of this circumstance: metrics such as heart rate, blood pressure, or oxygen levels were only stored in special facilities such as hospitals or medical centers and were only available to restricted personnel. Nowadays, these data are also measured and stored by smart watches or smart bracelets that are connected to the cloud and create personal profiles for each user.

Limited computational capacity of the devices: this makes them easy to crash, which is quite convenient when a cybercriminal wants to perform a DoS (Denial of Service) attack. The number of petitions that can be handled by these devices is far more limited than in conventional ones. In addition, it hinders the task of using antiviruses or cryptography algorithms, since the current versions are only supported by more powerful devices.

2.3. *Malware Characterization.*

Characterization can be explained as a process in which a set of features are extracted from someone or something. This makes it possible to describe each item in an unambiguous way. Thus, malware characterization is the process of identifying and extracting these features from each malicious sample. In this field, the characteristics are divided into the following categories:

Static features: here, the focus is on the analysis of the intrinsic characteristics of a binary file without executing its code in the system. Information such as the strings that appear in it, its sections, architecture, opcodes, cyclomatic complexity, or entropy belongs to this category. The main advantage is that static characteristics are quick to extract automatically. On the other hand, the usefulness of the features may be affected if the sample is packed or obfuscated (i.e., disassembly code and strings).

Dynamic features: here, the target is the analysis of the behavior of the sample at runtime by monitoring the different actions that it carries out in the system. The data are extracted from the communication that the malware performs through the network and its interaction with the system, such as system calls or open files, among others. One of its disadvantages is that only characteristics of the executed portions of code are captured, so the criminals include monitoring detection techniques that prevent the sample from executing entirely. In addition, the extraction of dynamic features is more time consuming than the retrieval of static features due to the fact that the sample must be executed for a short period of time.

2.4. *SOA.*

SOA is a software design paradigm in which modules work as independent services providing a specific interface to be called upon. They communicate through an Enterprise Service Bus (ESB) which is formed of one or several protocols, allowing the addition of services with little effort. In order to call each service when it is needed, an orchestration process is used [7]. Under this scheme, it is possible to add new components or new protocols. In addition, this architecture allows the easy integration of multiple SOA-based applications.

2.5. Related Work. As far as the authors are aware, there are no approaches available in the literature that jointly tackle the task of analyzing large numbers of malware samples specifically designed for the IoT and that of classifying or clustering them. On the contrary, most of the approaches try to describe specific malware samples or families, as mentioned in Section 2.5.1. In terms of automatically analyzing a great number of malware samples, there are some articles, but they focus only on Linux-based operating systems for x86 architectures, as is shown in Section 2.5.2. Finally, Section 2.5.3 covers approaches focused on classifying IoT malware, but these do not take into account all IoT architectures or families and neither do they study both static and dynamic features.

2.5.1. Malware Survey. Pa et al [8] presented a Telnet honeypot for different IoT architectures. They conducted a study of the malware that was aimed at this service, showing the problem that it suffers from when it is accessible from the Internet. The authors also presented the first sandbox that supported different architectures and executed the binaries and commands received through their honeypot.

Cozzi et al. [9] presented a complete malware study aimed at Linux-based operating systems. They statically and dynamically analyzed more than 10,000 samples distributed among the main architectures, namely, ARM, PowerPC, and MIPS, among others. They presented the main techniques used by malware and numerically expressed their use in the samples that made up their dataset. To carry out their analysis, they introduced the first malware analysis framework aimed at analyzing Linux-based malware.

Costin et al. [10] introduced a study of 60 families of IoT malware. The authors studied the timeline of events related to each family as well as the most relevant vulnerabilities used by them. For the dynamic analysis, the authors presented a sandbox compatible with the main IoT architectures based on the open source project Cuckoo Box [11].

2.5.2. Linux-Based Sandbox. Limon [12] is a sandbox for analyzing Linux-based malware. It collects calls to the operating system as well as capturing network traffic. Its main problem is that it only supports binary analysis in x86 architectures, and the operating system used to perform dynamic analysis is based on Ubuntu, which is not a very common operating system in the IoT. Similar problems are present in Detux [13], which, although it supports five architectures, is based on the Debian operating system. Detux only performs basic static analysis and network analysis, ignoring malware behavior within the operating system.

Chang et al. [14] proposed a sandbox for analyzing malware samples in the IoT. It is able to collect network packages and malware behavior in the system. To test the functionality of their sandbox, they experimented with the Zollard botnet.

2.5.3. Classification. Nghi Phu et al. [15] presented a framework for analyzing and classifying malware in the IoT. Their framework supports the MIPS architecture and extracts features related to malware interaction with the system in order to train a machine learning model.

Alhanahnah et al. [16] suggested a new approach to classifying IoT malware compiled for different architectures. Its method is based on generating signatures at a high level since these are more robust and vary less between architectures.

Su et al. [17] introduced a method for malware classification in IoT environments. It is based on converting malware into an image and a convolutional neural network for classification. It is able to classify a sample into malware or goodware and recognizes two malware families: Mirai and Gafgyt.

Kumar et al. [18] proposed a new approach to differentiate between malicious and benign applications based on a ranking of permissions used in Android IoT devices. Their methodology included an improvement on the random forest algorithm, achieving an increase in the accuracy of malware detection.

Lei et al. [19] presented a system for malware detection on Android-based IoT devices. They proposed the use of event groups instead of API calls to capture malware behaviour at a higher level than in API level. They trained and evaluated their system with a dataset of around 15,000 and 29,000 benign and malicious Android apps, respectively.

3. Proposed Architecture

This section describes the proposed SOA-based modular framework for analyzing and classifying malware samples from different IoT architectures. It consists of six modules which are invoked as services by the orchestrator of the system, which is responsible for using each module and processing the information extracted in each of the stages. Due to its modular structure, each of the modules that make up the system can be used independently (i.e., deploying a virtual machine to execute commands from a honeypot or even for adding new components). These services use our Enterprise Service Bus (ESB), which allows us to integrate any new component easily. Figure 2 shows a global view of our architecture.

3.1. System Overview. The system uses an executable file from any of the architectures supported as input, analyzes it, and produces a cluster based on the similarity that it has with other previously examined files as output. Although the proposal is designed for malware analysis purposes, it is valid for clustering other types of executables. The following sections describe in detail the modules of which our system is composed.

3.2. The Orchestrator. This is the main module of the system and the one in charge of making the pipeline that interconnects the rest of the modules. Once it obtains a sample, it uses the static analysis module to obtain the information necessary to continue with the next phase. Then, it uses the deployment module to check whether the architecture of the analyzed file is supported, that is, whether there is a virtual

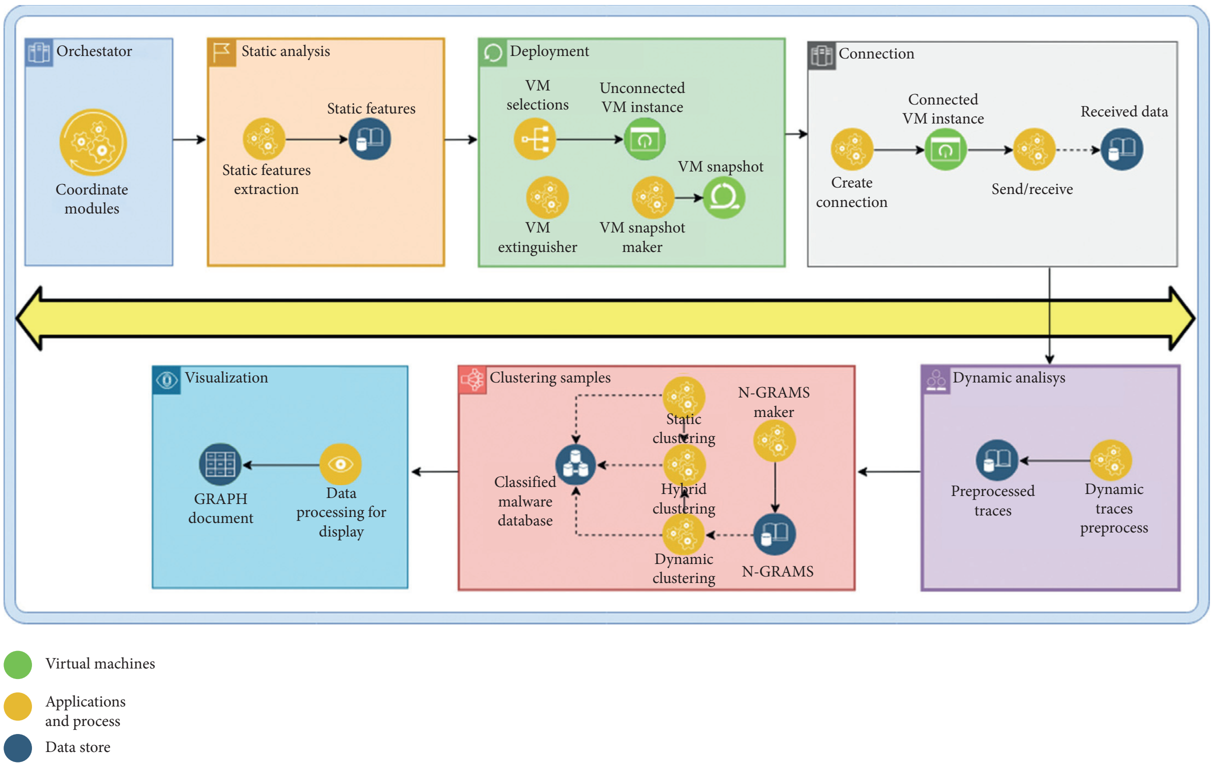


FIGURE 2: The proposed architecture for the analysis and clustering of IoT malware.

machine that supports that architecture, and if it is, it starts the virtual machine instance.

Once the virtual machine is on, it connects to it through the connectivity module and then proceeds with dynamic analysis, executing the file with the monitoring tool indicated in the configuration files. Then, the file is executed for a certain time which is indicated through the configuration commands of the framework. Once that timeout has elapsed, it obtains the result in the form of execution traces, destroys the virtual machine, and recovers the previous snapshot of the machine.

Finally, it calculates the similarity with other analyzed samples and adds it to the corresponding cluster if the similarity index is greater than the established threshold. The sample will be added to the cluster in which the most similar sample is located. If the threshold is not reached, a new cluster will be created to include the analyzed file.

Additionally, if the display parameter is active, it will calculate the similarity between all the samples and generate a graph connecting all of them.

3.3. Static Analysis. This module is responsible for obtaining and parsing the Executable Linkable Format (ELF) files. It is built upon *radare2* [20], a reverse engineering suite, and automates the process of obtaining information contained in the headers of the ELF files, as well as data regarding their sections. The static analysis module collects the following information.

Information file: characteristics of the headers of the executable file, such as architecture, whether the binary has been stripped of the symbols or not, and whether it was compiled with static or dynamic libraries.

Entropy: this measures the lack of predictability of a data set. In binary analysis, a high entropy value indicates that the sample is obfuscated or packed.

Cyclomatic complexity: this is a metric used in software engineering to calculate, in a quantitative way, the complexity at a logical level of a program or function [21]. Cyclomatic complexity is calculated for each of the functions found in the disassembled code.

Opcodes: the sequence of operation codes (opcodes) of all the functions present in the disassembly of the program are extracted and stored.

Libraries: the name of the shared libraries used by the program.

Sections: the sections into which the executable is divided are extracted, also determining their permissions and entropy.

Functions: the name of the functions imported from the libraries and used by the program.

Strings: all text strings present in the sample.

Hash: the hash to uniquely identify the executable.

3.4. Deployment Module. This module is responsible for starting the virtual machine, shutting it down, or restarting it. Its input is the architecture for which the malware was developed, which is searched for in the library in order to determine whether it can be emulated or not. It uses libvirt

[22] to manage the virtualization platforms and the QEMU [23] emulator as hardware virtualizer. To emulate an architecture, it has to be supported by QEMU, and a guest domain in an eXtensible Markup Language (XML) must be defined. This file contains the configuration of the machine in libvirt, that is, its storage, CPU architecture, kernel image, and network properties. Once the machine has been started, the module returns a handler, which allows you to shut down or restart the machine as well as to see which machines are currently active. Finally, when a machine is stopped, a previous snapshot of the machine is recovered in order to have a malware-free image for the next analysis. In this way, this module provides the flexibility to add user-defined virtual machines and uses them in our framework.

3.5. Connectivity and Dynamic Analysis. This is the module responsible for establishing connection with the virtual machine. It allows the upload and download of files through the Secure Copy Protocol (SCP) and the execution of commands through the Secure SHell (SSH). It provides the flexibility to upload any file type and execute commands in the virtual machine. For example, it can upload an executable file or script and use any type of monitoring tool available in the virtual machine for extracting information about its behavior, such as *strace* [24] or *systemtap* [25]. Finally, download the monitored traces and parses the collected data. The parsing function is responsible for extracting the executed syscalls from the execution traces as well as their parameters and results. Table 1 shows an example of a run sequence and the syscall data.

3.6. Clustering of Samples. This is in charge of clustering the binary files based on some of the previously extracted features. Given two executable files, it calculates the index of similarity between them and, if this is greater than a set threshold (set through the configuration parameters), these samples are considered to be related and, therefore, will be part of the same cluster. To calculate the similarity, the module uses the following approaches:

Dynamic approach. We use the execution traces obtained in the dynamic analysis to generate sequences of *syscall* names of size *n* (set through the configuration parameters), which are known as *n*-grams. An example for a sequence of size $n = 4$ is shown in Table 1, resulting in the following set of *n*-grams: (*brk, socket, fcntl64*, and *fcntl64*), (*socket, fcntl64, fcntl64,* and *setsockopt*), and (*fcntl64, fcntl64, setsockopt,* and *brk*). In order to determine the similarity, we use the Jaccard index [26] as a metric, which, for two sets of n-grams, is calculated as

$$\text{jaccard}\left(s_1, s_2\right) = \frac{\left|s_1\right| \cap \left|s_2\right|}{\left|s_1\right| \cup \left|s_2\right|}, \tag{1}$$

Table 1: Format execution trace.

Syscalls	Parameters	Results
Brk	0x32000	0x32000
Socket	AF_INET, SOCK_RAW, IPPROTO_TCP	0
fcntl64	0, F_GETFL	0x2
fcntl64	0, F_SETFL, O_RDWR|O_NONBLOCK	0
Setsockopt	0, SOL_IP, IP_HDRINCL, [1], 4	0
Brk	0x33000	0x33000

where the numerator indicates the number of unique subsets that are present in both sets, and the denominator indicates the total number of unique subsets between s_1 and s_2. The result is a value between 0 and 1 which indicates the degree of similarity between two sets of *n*-grams.

Static approach. We use two metrics to measure the similarity between two executable files. The first is based on sequences of opcodes of size *n* extracted from the disassembled code. This is calculated in the same way as in the dynamic approach but using opcodes instead of *syscalls.* The second is based on the cyclomatic complexity of each of the functions present in the disassembled binary. A distance function is used for the calculation of the similarity between two executable files. This function is formalized as follows:

$$\text{distance}\left(s_1, s_2\right) = \sum_{i=0}^{|F|} \frac{\min\left(f_i^{s_1}, f_i^{s_2}\right)}{\max\left(f_i^{s_1}, f_i^{s_2}\right)} x \frac{1}{F}. \tag{2}$$

For example, let us consider two executables with five and seven functions, the first with cyclomatic complexities 3, 5, 3, 7, and 4 and the second with complexities, 3, 3, 6, 6, 4, 5, and 2. The first sample has two functions with cyclomatic complexity 3, one with 5, one with 7, and another with 4. In the second sample, we have two functions with cyclomatic complexity 3, two with 6, one with 4, one with 5, and another with 2. We normalize the vectors so that they have the same number of elements, and the vectors (0, 2, 1, 1, 0, 1) and (1, 2, 1, 1, 2, 0) are obtained. Therefore, the similarity index between the two vectors is 0.5 and is calculated as follows: ((0/1 + 2/2 + 1/1 + 1/1 + 0/2 + 0/1)/6).

Hybrid approach. The hybrid approach allows clustering using the indexes described above. To do this, it assigns a weight to each of the indexes to calculate the final similarity index. The weight of each index can be configured in the framework configuration files.

3.7. Visualization. Its function is to visually represent the groupings generated based on the approaches described above. We denote *f* as a function that defines whether two malware samples are similar or not using the following expression:

$$f\left(x = \operatorname{metric}\left(s_1, s_2\right) s_1, s_2\right) = \begin{cases} 1, & x \geq z \\ 0, & x < z \end{cases}; z \quad x \in [0,1]; s_1, s_2 \in D, \tag{3}$$

where z being the selected threshold for determining the similarity between two samples, namely, s_1 and s_2, both belonging to the dataset of samples, which is defined as D. It generates a graph file in *dot* format [27] in which the nodes represent the executable files, and an edge between two nodes represents the fact that between them there is a similarity greater than the established threshold. The generation of the graphs is computationally expensive since it calculates the similarity for each different pair of samples.

4. Experiments and Results

In this section, the experiments and results obtained using our malware analysis and clustering framework are presented.

4.1. Overview. In order to test the platform described in Section 3, we built different custom virtual machines using *buildroot* [28], which automates the process of building an embedded Linux system. In total, we built machines for the five most widely used architectures in the current IoT market, namely, Intel 80386, x86-64, MIPS, ARM, and PowerPC, generating a file system and a compilation of a kernel image for each one. We used *strace* as a monitoring tool to obtain the execution traces.

To perform the analysis, we used different samples of Linux-based malware which targets IoT devices. The samples are distributed among the five architectures mentioned. The malware samples are labeled using AVClass [29], which categorizes them using a ranking of the labels provided by different antivirus engines. Table 2 summarizes the number of pieces of malware used for each architecture and how many of them are packed and labeled.

Finally, we used our framework to analyze all the samples and visualize the relationships between them according to the metrics described in Section 3.4. The following sections show the results obtained after analyzing the entire set of samples described above in terms of static and dynamic points of view.

4.2. Static. In this section we present the results of the analysis and clustering processes using the static features described in Section 3. We use a threshold, which can be adjusted by the user, of 0.8 to determine whether two samples are related for both metrics. This value selection is based on an empirical study which is out of the scope of this paper.

4.2.1. n-grams. We use the *n*-grams of the operation codes extracted in the static analysis process. The size was empirically determined to be four by using cross validation. Since the operation codes are architecture dependent, we generated clusters for each of the architectures independently. Figure 3 shows the graphic for all architectures in the study, namely, MIPS, PowerPC, x64, x86, and ARM. The nodes represent malware samples and the edges indicate whether there is a similarity greater than 0.8 at the *n*-gram level. Gray is used to represent malware samples that do not have a label and the rest of the colours represent each of the families that have been labeled (*AVClass*) in the dataset. As can be seen, there are different clusters formed mainly of samples from the same family. In some cases, there are related samples from several families. This may be because some of the samples are packed and, if they use the same packer, they may share the same code routines to unpack the executable at run time. One of the disadvantages of using static features is that they can be affected by code obfuscation. This metric can also be affected depending on whether the executable is compiled with static linking or with dynamic linking, since those binaries compiled with static linking could have more unique *n*-gram sequences because the functions imported from the libraries are included in the binary itself. In general terms, the proposed architecture detects well the families of malware samples for all the architectures.

4.2.2. Cyclomatic Complexity. We use cyclomatic complexity to cluster the samples. Since the metric is extracted from disassembled programs and depends on the assumptions of the compiler and the assembly code that it generates, we cluster the samples for each of the architectures independently. This is because, after looking at several executable files available for different architectures (e.g., *busybox*), we observe that the cyclomatic complexity for the same functions varies according to the architecture. Although it is not very different between one and the other, it does change even if they have been compiled with the same compilation options. Figure 4 shows the graph for all the architectures used in this paper. As we can see, the clusters generated belong to the same family, and there are several small clusters for the same family, such as *Gafgyt, Tsunami*, *or Mirai* for the ARM architecture. This is due to the fact that this metric measure similarity at a structural level between two samples. Therefore, it can also be affected by obfuscated code. In addition, if a sample is compiled in a static way and another in a dynamic way, there will not be a structural similarity between them (those compiled with static linking have imported library functions within the executable instead of being resolved at runtime as in binaries compiled with dynamic linking).

Observing the graphs generated for both metrics (Figures 3 and 4), it can be seen that, in general, the clusters created using *n*-grams are made up of more samples than those produced using cyclomatic complexity. In either case, most of the connected samples are related to others from their own family without producing many false positives.

4.2.3. Dynamic. In this section, we present the results obtained in the clustering process using the dynamic characteristics extracted in Section 3.5 and the metric described in the same section. As was done in Section 4.2.2, we use a threshold of 0.8 to match two malware samples. We use sequences of *n*-grams of size four for the *syscalls* executed for

Table 2: The number of malware samples distributed for each of the architecture.

Arch	Samples	Packed	Labeled
Intel 80386	279	58	211
X86-64	344	168	134
MIPS	318	63	288
ARM	246	24	200
PowerPC	275	12	258
	1462	325	1091

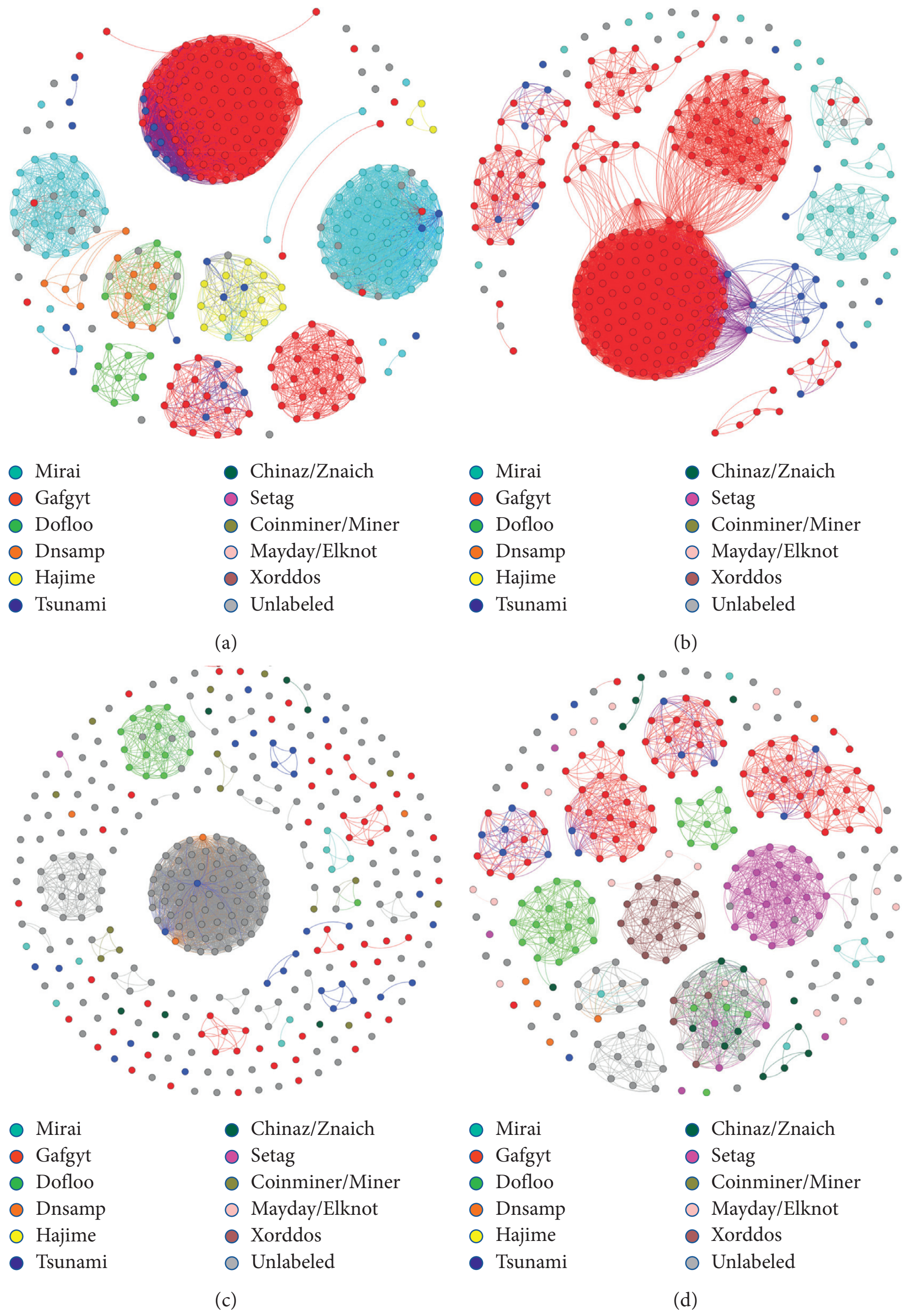
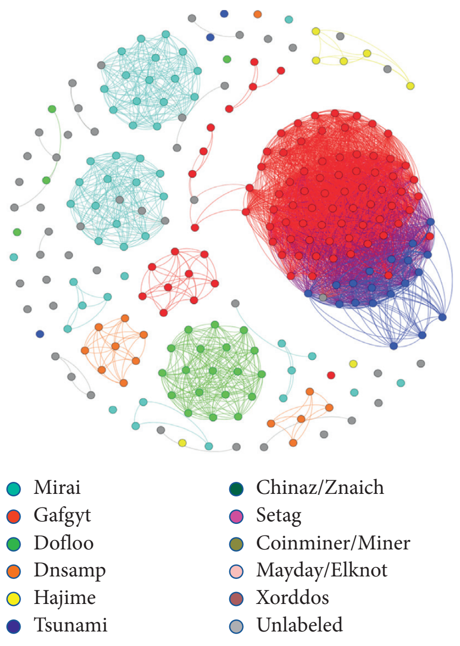


Figure 3: Continued.



(e)

Figure 3: Clusters generated for the MIPS (a), PowerPC (b), x64 (c), x86 (d), and ARM (e) architectures using *n*-grams and the Jaccard index to calculate the similarity.

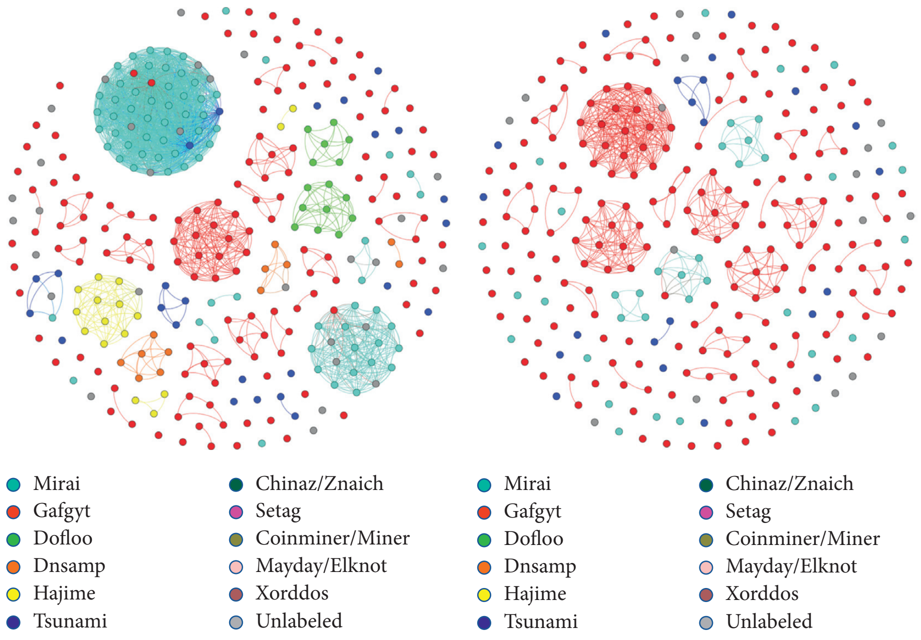


Figure 4: Continued.

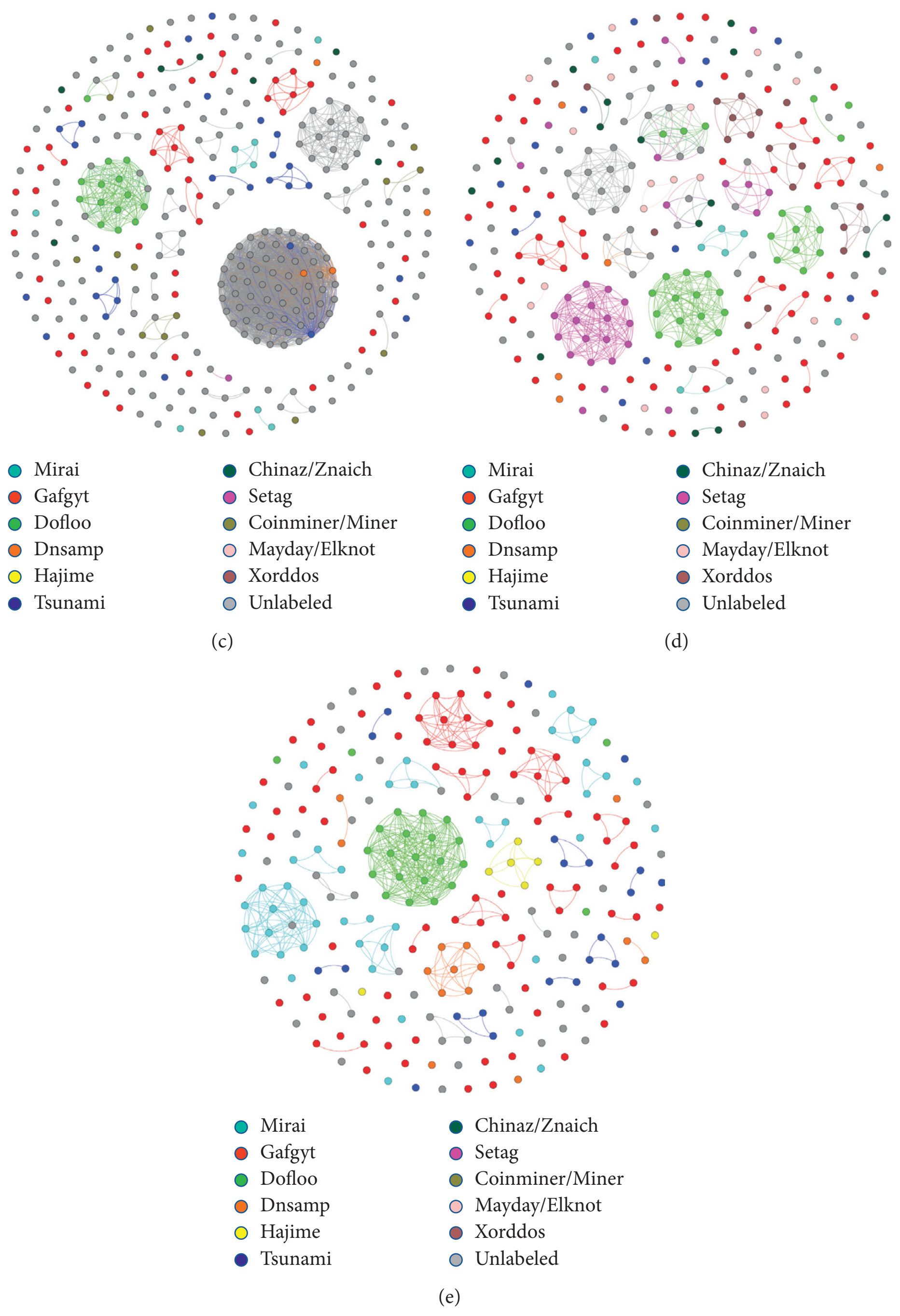


Figure 4: Clusters generated for the MIPS (a), PowerPC (b), x64 (c), x86 (d), and ARM (e) architectures using cyclomatic complexity and the custom function described in Section 3.

each of the samples. Since the *syscalls* are petitions to the operating system to request a service (e.g., create a socket and kill a process), and these have the same name in any Linux-based operating system, using them for clustering allows us to find similarities between the execution traces of samples from different architectures.

Figure 5 shows the clusters generated using the *syscalls* traces as features. On the left, each sample is colored depending on the architecture to which it belongs. On the right, each sample is colored depending on the family to which they belong, with gray indicating the unlabelled ones. It can be observed that there are clusters that are formed of samples from different architectures, such as MIPS, PowerPC, and Intel 80386. If we observe these same clusters in the family-categorized image, it can be seen that the samples belong to a particular malware family. In addition, it can be noticed that the clusters are made up of samples from the same family, and that, based on their behavior, pieces of malware from different architectures have been categorized into the same cluster.

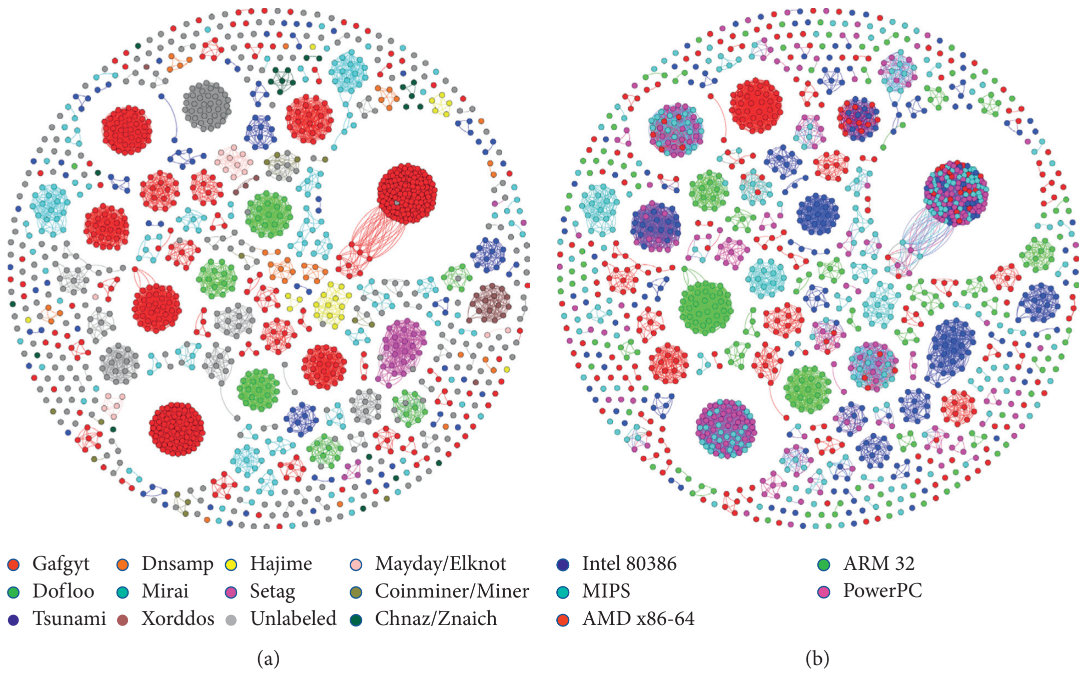


Figure 5: Clusters generated for all architectures using the execution traces obtained in the dynamic analysis. The n-gram size used for the *syscalls* sequence is four. The edges connect those samples with a similarity index greater than 0.8.

Finally, we observe that there are different clusters for the same family. Unlike the previous case, in which the samples may appear different depending on the architecture for which they were compiled or the different compilation options, now it may indicate that they belong to different campaigns of the same family. Malware is constantly evolving, and its creators add new functionalities or use existing ones from other pieces of malware that have proven effective and beneficial. Also, it should be noted that the original source code of some of the most widely used malware families is available on the Internet, such as *Gafgyt* or *Mirai* [18], and there may be variants created by different authors.

5. Conclusions

In this proposal, we have addressed IoT malware analysis, focusing on the automatization of the examining process. Our motivation for this is the huge increase in cyberattacks that have been carried out in this environment over recent years, which has led to the impossibility of manually studying the samples as the number is too immense. After evaluating the proposals from the community, it has been observed that there were none that focused on both analyzing (statically and dynamically) a large number of IoT malware samples at once and providing compatibility with several architectures.

Consequently, a multiarchitecture framework for automatic malware analysis and clustering has been presented. The proposal, which is based on a modular approach and supports samples from five different IoT architectures, namely, ARM, PowerPC, MIPS, Intel 8086, and x64-86, is able to extract static and dynamic features from a sample and compare it with previous analyzed ones, categorizing it into families depending on the similarity. In addition, besides saving a considerable amount of time when examining pieces of malware, it offers flexibility to the user, allowing them to define their own emulated architectures and to adapt the threshold used to determine whether a sample is categorized into a family or not.

The proposal has been evaluated through the examination of nearly 1,500 malware samples from the five architectures that are supported by the framework, offering promising results and proving its effectiveness when clustering malware samples. Especially relevant is the outcome of the dynamic analysis, in which the proposal has been able to cluster samples from multiple malware campaigns, even if they were designed for different architectures. In addition, it has been detected that, when clustering using the static features, samples may appear different depending on the architecture for which they were compiled or the different compilation options. Other factors, such as code obfuscation, also hinder the task, although the results generated by the static analysis are also satisfactory.

Given the good results offered by the framework when tested and knowing the importance of improving the analysis of malware samples, there are several lines of research that could be followed to complement this proposal. Some such projects could be to

- Study the network communications made by the malware samples when they are executed and use them as a feature to cluster them
- Expand the visualization features, offering the user an interactive representation of the results, allowing them to directly browse through the different samples or filter them by selecting certain characteristics.

Add other IoT architectures so that samples designed for them could also be examined.

Employ other metrics to determine sample similarity, and even to use advanced machine learning techniques to add a layer of intelligence to the framework.

Data Availability

The sample data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the MINECO and European Commission (FEDER funds) under project RTI2018-098156-B-C52, the JCCM under the project SB-PLY/17/-180501/-00035, the Spanish Education, Culture and Sports Ministry under grants FPU 17/03105 and FPU 17/02007, the University of Castilla-La Mancha under the contract 2018-PREDUCLM-7476 and the project 2020-GRIN-28846, and the Spanish State Research Agency under the project PEJ2018-003001-A.

References

[1] D. Demeter, M. Preuss, and Y. Shmelev, "IoT: a malware story-securelist," 2019.

[2] The Council of Economic Advisers - United States of America and CEA Report, *The Cost of Malicious Cyber Activity to the U.S. Economy*, The Council of Economic Advisers, Washington, DC, USA, 2018.

[3] E. L Xua and L. Ling, "Industry 4.0: state of the art and future trends," *International Journal of Production Research*, vol. 56, pp. 2941–2962, 3 2018.

[4] P. P. Gaikwad, J. P. Gabhane, and S. S. Golait, "A survey based on smart homes system using internet-of-things," in *proceedings of the 2015 International Conference on Computation of Power Energy, Information and Communication (ICCPEIC)*, Piscataway, NJ, USA, 2015.

[5] Gartner Says 8.4 Billion Connected Things Will Be in Use in 2017 https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016.

[6] Gartner Says 5.8 Billion Enterprise and Automotive IoT Endpoints Will Be in Use in 2020 https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io.

[7] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, R. Metz, and B. A. Hamilton, "Reference model for service oriented architecture 1.0," *The Organization for the Advancement of Structured Information Standards*, vol. 12, 2006.

[8] Y. M. P. Pa, S. Suzuki, K. Yoshioka et al., "IoTPOT: a novel honeypot for revealing current IoT threats," *Journal of Information Processing*, vol. 24, no. 3, pp. 522–533, 2016.

[9] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding linux malware," in *in proceedings of the 2018 IEEE Symposium on Security and Privacy*Security and Privacy, Francisco, CA, USA, July 2018.

[10] A. Costin and J. Zaddach, "IoT malware: comprehensive survey," *Analysis Framework and Case Studies*, BlackHat, Las Vegas, NV, USA, 2018.

[11] C. Guarnieri, "Cuckoo sandbox-automated malware analysis," 2016, https://cuckoosandbox.org/.

[12] https://github.com/monnappa22/Limon.

[13] detuxsandbox/detuxhttps://github.com/detuxsandbox/detux.

[14] K.-C. Chang, R. Tso, and M.-C. Tsai, "IoT sandbox: to analysis IoT malware zollard," in *Proceedings of the Second International Conference on Internet of Things*, Data and Cloud Computing, New York, NY, USA, September 2017.

[15] T. N. Phu, K. H. Dang, D. N. Quoc, N. T. Dai, and N. N. Binh, "A novel framework to classify malware in mips architecture-based IoT devices," *Security and Communication Networks*, vol. 2019, Article ID 4073940, 13 pages, 2019.

[16] M. Alhanahnah, Q. Lin, Q. Yan, N. Zhang, and Z. Chen, "Efficient signature generation for classifying cross-architecture IoT malware," in *Proceedings of the 2018 IEEE Conference on Communications and Network Security (CNS)*, Beijing, China, June 2018.

[17] J. Su, D. V. Vasconcellos, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai, "Lightweight classification of IoT malware based on image recognition," in *Proceedings of the 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Tokyo, Japan, July 2018.

[18] R. Kumar, X. Zhang, R. U. Khan, and A. Sharif, "Research on data mining of permission-induced risk for android IoT devices," *Applied Sciences*, vol. 9, 2019.

[19] 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 of Things Journal*, vol. 6, no. 4, pp. 6668–6680, 2019.

[20] R. Team, *Radare2 Book*, 2017.

[21] A. H. Watson, D. R. Wallace, and T. J. McCabe, "Structured testing: a testing methodology using the cyclomatic complexity metric," *US Department of Commerce, Technology Administration, National Institute of Standards and Technology*, vol. 500, 1996.

[22] libvirt The virtualization API https://libvirt.org/.

[23] Q. E. M. U. 2020, https://www.qemu.org/.

[24] strace trace system calls/signals, https://linux.die.net/man/1/strace.

[25] SystemTap, https://sourceware.org/systemtap/.

[26] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets*, Cambridge university press, Cambridge, UK, 2014.

[27] Graphviz-Graph Visualization Software, http://www.graphviz.org/documentation/.

[28] Buildroot-making embedded Linux easy, https://buildroot.org/.

[29] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "A tool for massive malware labeling," in *Proceedings of the International Symposium on Research in Attacks Intrusions, and Defenses*, Paris, France, September 2016.