Received July 16, 2020, accepted August 1, 2020, date of publication August 7, 2020, date of current version August 19, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3014891

V-Sandbox for Dynamic Analysis IoT Botnet

HAI-VIET LE[1,2] AND QUOC-DUNG NGO[3]
[1]Institute of Information Technology, Vietnam Academy of Science and Technology, Hanoi 10000, Vietnam
[2]Graduate University of Science and Technology, Vietnam Academy of Science and Technology, Hanoi 10000, Vietnam
[3]Department of Information Technology, Posts and Telecommunications Institute of Technology, Hanoi 10000, Vietnam

Corresponding author: Quoc-Dung Ngo (dungnq@ptit.edu.vn)

ABSTRACT With the increasing use of resource-constrained IoT devices, the number of IoT Botnets has exploded with many variations and ways of penetration. Nowadays, studies based on machine learning and deep learning have focused on dealing with IoT Botnet with many successes, and these studies have required relevant data during malware execution. For this, the sandbox environment and behavior collection tools play an essential role. However, the existing sandboxes do not provide adequate behavior data of IoT botnet such as the C&C server communication, shared libraries requirements. Moreover, these sandboxes do not support a wide range of CPU architectures, data is not exhaustively collected during executable file runtime. In this paper, we present a new practical sandbox, named V-Sandbox, for dynamic analysis of the IoT Botnet. This sandbox is an ideal environment for IoT Botnet samples that exhibit all of their malicious behavior. It supports the C&C servers connection, shared libraries for dynamic files, and a wide range of CPU architectures. Experimental results on the 6141 IoT Botnet samples in our dataset have demonstrated the effectiveness of the proposed sandbox, compared to existing ones. The contribution of this paper is specific to the development of a usable, efficient sandbox for dynamic analysis of resource-constrained IoT devices.

INDEX TERMS Sandbox, dynamic analysis, IoT botnet, machine learning.

I. INTRODUCTION

In recent years, the security of IoT devices has been of great interest to many researchers since a large number of IoT devices have been attacked and exploited vulnerabilities [1]–[6]. With the rapid growth in number [7] and less attention to information security [8]–[10], IoT devices gradually become an attractive target for attackers.

There are many criteria for classifying IoT devices, such as device connectivity, device manufacturers, etc. In this paper, based on Bencheton's classification approach [11], IoT devices are divided into resource-constrained and high-capacity ones. Furthermore, we propose a sandbox to deal with resource-constrained IoT devices for the following reasons:

- Due to resource-constrained (such as CPU, RAM, Flash memory, etc.), it is not trivial to integrate threat detection/protection solution in these devices. Resource-constrained devices have become an attractive target for hackers with many new variants of Botnets. Therefore, this field is a big challenge for researchers to deal with IoT Botnet detection for these devices.
- Using machine learning/deep learning in dynamic analysis of IoT botnet, researchers need to collect malicious behavior. However, existing sandboxes do not provide fully behavior data of this malware, especially the C&C servers connection, shared libraries for dynamic files, and a wide range of CPU architectures.

Malware on IoT devices not only spreads on individual personal devices but also targets businesses, organizations, and governments with increasing severity [12]. Kaspersky's statistics show that there were more than 100 million attacks on IoT devices in the first half of 2019, a 7-fold increase compared to the same period in 2018 [13]. Addressing the risks above, malicious researchers have developed new methods and frameworks to effectively analyze and detect malware samples on IoT devices [14]–[25]. These studies can be divided into two main groups, which are static analysis [14]–[19] and dynamic analysis [20]–[25].

As in [19], [25], authors have presented the static analysis method, which allows full control of the control flow (CFG) and data flow (DFG) to detect malicious code by specific analytical techniques such as byte code, system calls API or Printable Strings Information (PSI) [19], [26]. This method allows a detailed analysis of files and gives the activation capabilities of malicious code [27]. However, the static

The associate editor coordinating the review of this manuscript and approving it for publication was Chunhua Su.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

analysis method is challenging to apply to malicious code using complicated techniques (obfuscation) or difficult to collect samples because malicious code is only stored on the device's Random Access Memory (RAM) [7]. According to Andreas Moser [28], the static analysis method should be used as a complement to the dynamic analysis method.

Dynamic analysis is a method of monitoring, collecting, and analyzing system behaviors to detect malicious code [29]. One of the most popular approaches nowadays is to use machine learning/deep learning. These approaches are necessary to collect relevant data during malware execution. In general, this step requires an adequate sandbox to monitor the behaviors of executable files. Collected data plays an essential role in the accuracy of detecting malicious behaviors.

There are many studies on building sandboxes for IoT devices, but most focus on IoT devices running Android operating systems (smartphones, smart TVs, etc.) [26]–[30]. The reason is that Android devices have powerful and sufficient resources (CPU, RAM, Disk, GPU) to easily create simulation environments for dynamic analysis (Android SDK, Virtual Box, VMware, KVM, etc.). The problem of building a sandbox for resource-constrained IoT devices has not been given much attention.

There are several researched and developed sandboxes for IoT devices that can be mentioned, such as [35]–[41]. The survey of these IoT sandboxes is presented in section II-C. However, these sandboxes have some limitations, such as they do not support a wide range of CPU architectures, and data is not fully exhaustive collected during executable file runtime; the emulation environment does not provide crucial components such as C&C server simulator and shared libraries dynamically. In this paper, we focus on developing sandbox for resource-constrained IoT devices targeting IoT Botnet detection.

To overcome the above limitations, the proposed IoT Sandbox meets the following criteria:

- Support for a wide range of CPU architectures;
- Autoconfiguration of emulation environment to run executable files;
- No effect on the external environment;
- Provision libraries (shared objects) that the executable file requires;
- Provision a fully simulated network environment for IoT malware such as connecting to the C&C server, transmitting and receiving commands from the C&C server;
- Behavior monitoring of IoT malware affecting the sandbox environment (system calls, files, folders, network traffic, hardware performance, etc.).

Our main contributions are:

- Summarize a set of dynamic features needed to detect IoT Botnet based on surveying research results by using machine learning to detect IoT botnet dynamically. After that, assess the existing IoT Sandboxes based on the proposed set of dynamic features, thereby proposing a sandbox architecture to overcome the existing shortcomings.
- Develop an adequate IoT sandbox for IoT Botnet detection.
- Evaluate and prove the effectiveness of the sandbox on the data set of 9069 samples obtained with 6141 malware and 2928 benign samples.

The structure of this paper is as follows: section II consists of the background and survey of related works, section III describes the proposed architecture, section IV experiments and evaluates the results, section V discusses the results, and section VI concludes the paper.

II. RELATED WORK AND BACKGROUND

In this section, we will discuss the characteristics of IoT Botnet as well as the researches related to applying machine learning to the dynamic analysis of IoT Botnet and sandboxes for existing IoT devices.

A. OVERVIEW OF IoT BOTNET

According to Cisco forecasts, by 2020, there will be about 50 billion devices connected to the Internet, and this number will increase to even more [7]. As its number is increasing, IoT devices gradually become an attractive and accessible environment for hackers to attack. In the world, many studies have shown the danger from malicious code on IoT devices (IoT Botnet) such as Bashlite, Mirai, Tsunami, Psyb0t [8]. In particular, with more than 1 million infected devices, including IP cameras, DVRs, and routers, Bashlite malicious code can launch DDoS attacks up to 400 Gbps through simple techniques like UDP or TCP Flood [8]. Bashlite malicious code is considered the precursor to Mirai, and the malicious code affects a wide variety of IoT devices. The Mirai Botnet network used in the DDoS attacks reached a record of 1.1 Tbps, with more than 100,000 IoT devices for home use [10].

By focusing on analyzing 16 families of IoT Botnet, discovered from 2008 to 2018, Vignau *et al.* [42] identified IoT Botnet as a significant threat to the IoT ecosystem, and malicious behaviors focused on launching a DDoS attack, characterized by Botnet. Nguyen *et al.* [19], Angrishi [8], and Kolias *et al.* [1] also presented behavioral features of the IoT Botnet, including:

- Scan for appropriate targets: Malware performs a random scan of IP addresses with commonly targeted service ports, such as Telnet or SSH.
- Gain access to other vulnerable devices: Bots engage in brute force attacks to discover the default login credentials of weakly configured IoT devices.
- Infect IoT devices: After logging into the device, the bot infects the loader used to download and execute the corresponding binary version of the botnet, usually via FTP, HTTP, etc.
- Communicate with the C&C server via IP address or URL: IoT Botnets try to connect and relay various device

features, such as IP address and hardware architecture, to the reporting server via another port.
- Wait and execute commands: The bot goes into the main execution loop first, which establishes a connection with the C&C server and keeps it alive, waiting for the next commands. If an attack command is received, the corresponding routine is called, and the attack is executed.

With these characteristics, many researchers have come up with methods to analyze and detect IoT Botnet. Section II-B focuses on analyzing research findings using dynamic analysis of the IoT botnet.

B. RELEVANT FEATURES FOR IoT BOTNET DETECTION

In this section, we present a survey on IoT Botnet for both resource-constrained as well as high-performance IoT devices. The dynamic analysis could be classified into two main family methods: network-based intrusion detection system (NIDS) and host-based intrusion detection system (HIDS) method.

Dynamic analysis studies to detect IoT Botnet mainly focus on network traffic analysis to find out the characteristics for building NIDS [43]–[48], similar to traditional malware analysis running on i386. The drawback of these studies is that malware is only detected when the device has been successfully infected and becomes part of the IoT Botnet network. To overcome the above disadvantages and to detect the possibility of an early attack, it is recommended to monitor the behavior of malicious code from the beginning of affecting the target device to infection using HIDS. Traditional HIDS can monitor software activities on computers to detect whether or not it is malicious. However, traditional HIDS is ineffective with resource-limited IoT devices (such as IP cameras, smartwatches, smart lamps). There are not many studies analyzing IoT Botnet behaviors on IoT hosts to build IoT HIDS [25], [49], [50]. Building an environment for IoT behavioral analysis Botnet's impact on IoT devices restricts resources with difficulties such as diverse and inconsistent IoT hardware platforms, the firmware of the closed device comes with the device or is available via the manufacturer's website.

The sandbox for these IoT devices is in the research and development stage (presented in section II-C). In this section, we will discuss the most relevant researches related to the use of machine learning in dynamic analysis of IoT Botnet. We also study behavioral analysis studies affecting a host of IoT malware to be applied to research problems. From there, we are orienting the requirements to be able to build the sandbox for analyzing IoT Botnet dynamically.

Doshi *et al.* [51] proposes a NIDS model for DDoS detection in network data streams connecting IoT devices by low-cost machine learning algorithms (including KN, LSVM, DT, RF, NN) with more than 98% accuracy. Doshi recommends using network stream characteristics such as *Packet size, Inter-packet Interval, Protocol, Bandwidth, IP destination address cardinality, and novelty.*

Indre and Lemnaru [52] proposes a solution that combines features extracted from network packets, focuses on analyzing data from the URI and RESTful methods, and proposes features based on the characteristic set of the KDD99 dataset.[1] Experimental results show that the typical set of Indre proposes to use with Passive Aggressive Classifier [53] achieves 98.4% accuracy. However, the KDD99 dataset has mainly collected data on computer network attacks since 1999. At this time, IoT Botnet has not appeared yet. Therefore, identifying the IoT Botnet based on the KDD99 dataset is not adequate.

Alrashdi *et al.* [54] proposes the use of the UNSW-NB15 dataset [55] for building NIDS to detect IoT Botnet in Smart City more effectively, replacing the outdated KDD99 dataset. Alrashdi uses Random Forest with 12 features selected from the UNSW-NB15 dataset to achieve an accuracy of 99.34%. These selected features include: *srcip, dstip, dur, dsport, ct-dst-src-ltm, ct-rsv-dst, ct-dst-ltm, ct-src-ltm, ct-src-dport-ltm, dbytes, proto, is-ftp-login.*

Prokofiev *et al.* [43] presents the NIDS model using logistic regression for IoT devices with an accuracy of 97.3%. To create the logistic regression model the following network traffic features were selected: *destination port, source ports, number of requests, even number of requests, mean interval between requests, requests on other ports, mean size of packets, delta for packet size, mean entropy of packets, alphanumeric.*

Wu *et al.* [48] uses the IoTBOX sandbox [36] to execute IoT Botnet samples in the dataset, and proposes the model of learning sequential sequences of Shell commands to detect IoT Botnet with approximately 90% accuracy. The limitation of Wu's proposed model is that the use of a large deviation training data set (300 Gayfgt logs, 51 log nttpd, 2430 log Zorro) results in inaccurate prediction models. The IoTBOX sandbox, with the weakness shown in the section below, is presented as the second limitation of this study.

In addition to the above NIDS, studies using HIDS for detection of IoT botnets will be discussed below.

Breitenbacher *et al.* [56] presents a practical HIDS that requires low power for IoT devices to detect malicious code. Hades-IoT IDS uses data about system calls to detect malicious behavior with 100% accuracy. But, this experimental result is only applied on limited data sets (2 types of IoT devices with 7 device models and two malicious samples are too few to prove). Features used in Hades-IoT IDS include *names, addresses, and parameters of system calls.*

Ham *et al.* [21] applies the SVM classification algorithm to detect malware on Android devices with 99.7% accuracy. In this paper, Ham selects 32 features from the 88 features that Shabtai presents [32]. However, collected data agents are installed directly onto real devices to collect data, resulting in restrictions on expansion for many different hardware devices and operating system versions. In addition,

[1] "KDD Cup 1999 Data." [Online]. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html, Accessed on: Apr. 23, 2018.

32 selected features are only suitable for 14 families of malware in the Dataset, which reduces detection with newly emerging malware samples.

Shabtai *et al.* [32] introduces a malware detection framework, named Andromaly, on Android mobile devices with 98.18% accuracy. The advantage of this framework is that it is lightweight, can run on real devices, consume fewer resources, and is almost transparent to the system. The author has selected useful features for malware detection through the results of testing the samples included in the data set. However, the small number of samples in the dataset (4 malware and 40 benign) leads to the selection of non-optimal features (88 features selected for detection). Besides, many features are only allowed to collect for real devices such as *Keyboard_Backlight, LCD_Backlight, Blue_Led, Green_Led, Red_Led, Camera, USB_State, Standard Print_Calls, Out Send_Calls, Missed_Calls, Out send_Non_CL_Calls*.

Azmoodeh *et al.* [57] proposes a machine learning model to detect ransomware attacks by monitoring the power consumption of IoT nodes. The machine learning algorithm used is KNN in combination with DTW for accuracy of 95.65%. The proposed Azmooded method allows ransomware to be detected on real ARM devices (Samsung Galaxy SIII, Samsung Galaxy S Duos, Asus Padfone Infinity). However, this study has two limitations: the dataset used has too few samples (6 ransomware and 12 benign), and the collection characteristics are required to be extracted from real devices, challenging to implement in replication in reality.

Ficco *et al.* [58] proposes machine learning models that use API call sequences for the classification of the IoT Botnet. This model with an F-measure is 89% (Naive Bayes algorithm) on a dataset. In this model, Ficco collects API calls from executed applications to build Call Dependency Graph and Calls tree. From there, Ficco extracts the appropriate feature into the Markov Chain behavioral model to classify the IoT Botnet. However, using the mobSF tool [59] to extract features that lead to a limitation of this study is only suitable for devices running Android, challenging to extend to other embedded operating systems of IoT devices.

From the above studies, an adequate sandbox must provide at least the following main feature types: system calls, extended information of system calls (EXINFO), network traffic, host performance. Based on the accuracy of machine learning algorithms applied by the authors on these features, we summarize a set of practical dynamic features for analyzing and detecting IoT Botnets described in Table 2. Particularly for network traffic features, the author chooses the features of the UNSW-NB15 data set [55] based on Janarthanan's results [60].

TABLE 1. Information collected from dynamic analysis of IoT botnet.

Author	Collected Data	OS/CPU	ML Algorithm	ACC
Doshi [54]	Network traffic	General devices	Random Forest	99.8%
Indre [55]	URI and RESTful methods	General devices	Passive Aggressive Classifier	98.4%
Alrashdi [54]	Network traffic	General devices	Random Forest	99.34%
Breitenbacher [56]	System calls	Frimware with Linux kernel/ MIPS, ARM	Rule-based with whitelist	100%
Prokofiev [43]	Network traffic	General devices	Logistic regression	97.3%
Wu [48]	Shell command sequences	Linux, OpenWrt/ MIPS, MIPSEL, ARM	N-gram	90%
Hyo-Sik Ham [21]	Network, Telephone, SMS Message, CPU, Battery, Process, Memory	Android/ ARM	SVM	99.7%
Shabtai [32]	Touch screen, Keyboard, Scheduler, CPU Load, Messaging, Power, Memory, Application, Calls, OS, Network, Hardware, Binder, Led	Android/ ARM	DTJ48	98.18%
Amin Azmoodeh [57]	Power consumption	Android/ ARM	KNN+DTW	95.65%
Massimo Ficco [58]	API call sequences	Android/ ARM	Naive Bayes	89%

C. OVERVIEW OF IoT SANDBOX

A survey of the characteristics, advantages, and disadvantages of IoT sandboxes is presented in the following content:

Yan and Yin [30] presents DroidScope, an Android malicious code analysis platform. The architecture of the DroidScope is shown in Fig. 1. The entire Android software system runs on the QEMU emulator, and the analysis is done from the outside. DroidScope extracts OS-level semantic knowledge: system calls and running processes, including threads and memory maps, with Java level getting the current Dalvik virtual machine program counter and frame pointer, all virtual registers. (its operands and their values). To analyze Android malware, it has implemented four analysis plugins such as API tracer, Native instruction tracer, Dalvik instruction tracer, and Taint tracker.

Bläsing *et al.* [31] proposes an Android Application Sandbox (AASandbox), monitors the name of system calls, and library calls on ARM architecture, including their arguments and return value. However, the effectiveness of AASandbox has not been clearly demonstrated with only 150 test samples. Also, using the Android SDK to build the simulation environment and Android Monkey to allow interaction with the template results in the ability to customize the running environment and collect the full range of malicious behavior. In Fig. 2, the system monitoring results of AASandbox are described. Each line has five values, including timestamp, used system call, return value, an ID of the process, and ID its parent. The author describes the architecture of AASandbox in Fig. 3.

TABLE 2. Summarized dynamically features for detection of IoT Botnet.

#	Feature type	Features	Feature description
1	System-calls	Syscall_time_stamp	Timestamp for each system-call
2		Syscall_name	Name of system-call
3		Syscall_list_arg	Arguments for system-call
4		Syscall_ret	The return value of the system-call
5		Syscall_pid	The process ID that is called by the ELF file
6		PID_num	Number of process created by ELF
7	EXINFO	exinfo_name	The name of exinfo
8		exinfo_type	This type of exinfo
9		exinfo_path	The path of exinfo
10	Network traffic	Net_service	Service type (http, ftp, smtp, etc.)
11		Net_sbytes	Source to destination transaction bytes
12		Net_sttl	Source to destination time to live value
13		Net_smean	Mean of packet size transmitted by the source IP address
14		Net_ ct_dst_sport_ltm	The number of connections of the same destination address and the source port in 100 connections according to the last time.
15	CPU usage	Num_User_login	Number of active user sessions
16		Total_process_active	The total number of processes activated in the system
17		Num_process_running	The number of processes running
18		Num_process_sleeping	The number of processes sleeping
19		Num_process_stop	The number of processes stopped
20		Num_process_zombie	The number of processes waiting to be stopped from the parent process
21		CPU_percent_user	Percentage of the CPU for user processes
22		CPU_percent_sys	Percentage of the CPU for system processes
23		CPU_percent_ni	Percentage of the CPU processes with priority upgrade NICE
24		CPU_percent_ide	Percentage of the CPU not used
25		CPU_percent_wait	Percentage of the CPU processes waiting for I/O operations
26		CPU_percent_hd_ir	Percentage of the CPU serving hardware interrupts
27		CPU_percent_st_ir	Percentage of the CPU serving software interrupts
28		CPU_percent_st	The amount of CPU 'stolen' from this virtual machine by the hypervisor for other tasks (such as running another virtual machine)
29	Process usage	Process_PID	ID of the process
30		Process_User	The user that is the owner of the process
31		Process_PR	Priority of the process
32		Process_NI	The "NICE" value of the process
33		Process_Virt	Virtual memory is used by the process
34		Process_RES	Physical memory used from the process
35		Process_SHR	Shared memory of the process
36		Process_S	Indicates the status of the process: S=sleep R=running Z=zombie
37		Process_percent_cpu	The percentage of CPU used by this process
38		Process_percent_mem	The percentage of RAM used by the process
39		Process_time	The total time of activity of this process
40		Process_comm	The name of the process
41	RAM usage	Mem_total	Total System Memory
42		Mem_used	Current used memory by system
43		Mem_free	Free memory
44		Mem_buff	Total memory used by buffer
45	Swap memory	Swap_total	Total swap memory in system
46		Swap_free	Free swap memory
47		Swap_used	Current used swap memory by system
48		Swap_cached	Total cached memory by system

Oktavianto and Muhardianto [35] develops Cuckoo Sandbox, which is a dynamic analysis system consisting of two main components: Cuckoo Server, and Cuckoo Host. The architecture of Cuckoo Sandbox is described in Fig. 4. The analysis is performed using the automated scripts available that Cuckoo Host required to perform during the analysis of the target application. Cuckoo is heavily dependent on Python, and there are some Python applications needed to run Cuckoo properly (Magic, Pydeep, Yara, Pefile). Therefore, using Cuckoo with environments that do not fully support Python will face a few difficulties. Cuckoo Sandbox gives good results analysis for the X86 architecture. For analyzing Linux malware, Cuckoo Sandbox shows a lack of features. Liu *et al.* [61] uses Cuckoo Sandbox to perform dynamic analysis to extract host behaviors and network behaviors of IoT botnet samples (Mirai, Satori, OMG, and Wicked). However, the results extracted from Cuckoo Sandbox mainly obtained network behaviors.

Pa *et al.* [36] proposes an IoT honeypot (IoTPOT) and sandbox (IoTBOX), which attracts and analyzes Telnet-based attacks against various IoT devices running on eight different CPU architectures such as MIPS, MIPSEL, PPC, SPARC, ARM, MIPS64, sh4, and X86. The architecture of the IoTBOX Sandbox shown in Fig. 5. Yin uses QEMU [62] to build cross-compilation environments for different CPU architectures and the respective OpenWrt platform to run on the

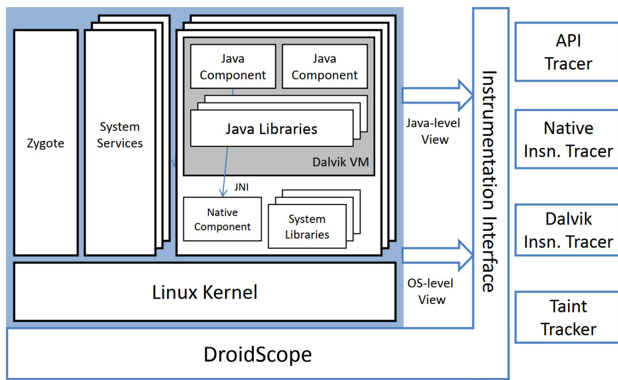

FIGURE 1. **DroidScope's architecture [30].**

```
found sct at 0xc0022f04
RK loaded
[1272980618.635790][recvfrom()-83][1;0]
[1272980618.636043][recvfrom()--11][1;0]
[1272980618.638795][munmap()-0][193;0]
[1272980618.640029][sigprocmask()-0][192;0]
[1272980618.641172][wait4()-192][39;0]
[1272980618.641429][read()--5][39;0]
[1272980618.641622][write()-4][39;0]
[1272980618.642570][close()-0][39;0]
[1272980618.643198][write()-24][81;0]
[1272980618.645523][write()-4][82;0]
[1272980618.712968][read()-61][37;0]
...
```

FIGURE 2. **The output of AASandbox [31].**

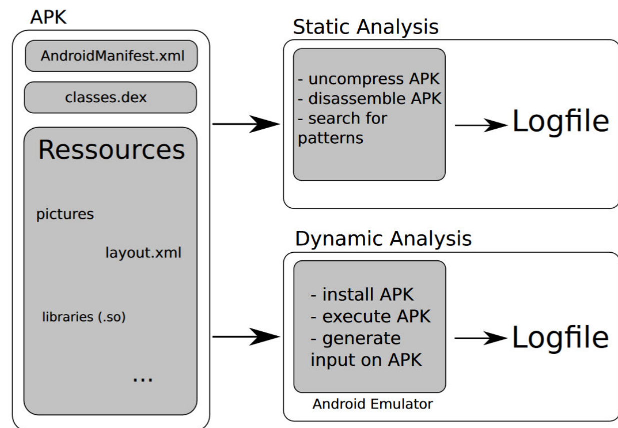

FIGURE 3. **Design of the AASandbox [31].**

emulated CPU environment. Then, the virtual bridge and the access controller are used to create a virtual networking environment. Through this virtual networking environment, IoTBOX can collect network traffic generated by the samples. In practice, the author only ran 52 samples on 3 CPU platforms, of which five samples failed to run. IoTBOX has only focused on monitoring network behavior, and the remaining behaviors are not mentioned.

Uhricek [43] raises the limitations of Sandbox such as Limon [38], REMnux [39], Detux, Padawan [40] including

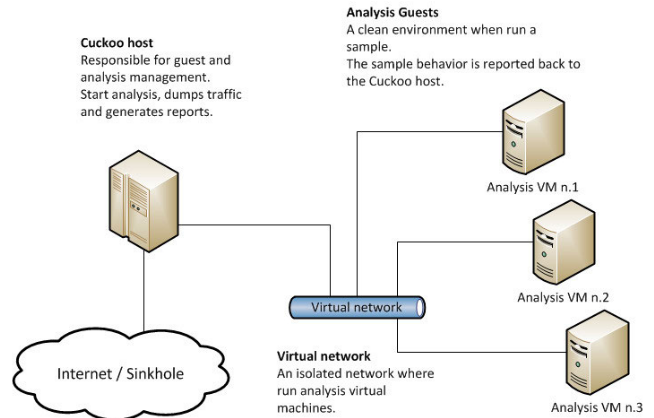

FIGURE 4. **Cuckoo's main architecture [35].**

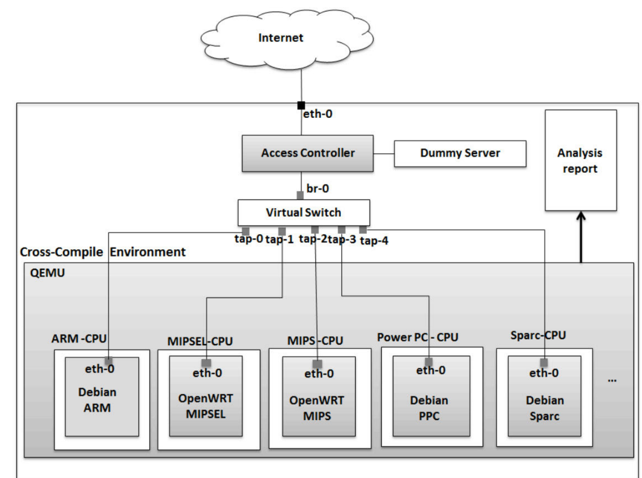

FIGURE 5. **Overview of IoTBOX [36].**

support for less CPU architecture, incomplete information gathering, etc. That is the motivation for Daniel Uhricek to introduce Linux Sandbox LiSa [41], which helps analyze Linux malware on MIPS, ARM, Intel 80386, x86-64, Aarch64 architectures. By tracking behaviors such as system calls, opening or deleting files, creating processes, network traffic of 150 IoT botnet samples (mostly for ARM and MIPS architectures), Uhricek demonstrates the efficiency of LiSa. LiSa uses SystemTap to collect system information to create process trees, trace syscalls, and mark open or deleted files. To collect network data, Uhricek uses *Scapy* and *dpkt* libraries. However, the role of a C&C server that is very important to demonstrate the behavior of IoT Botnet fully is not developed by Uhricek. Therefore, the behaviors of IoT botnet samples collected from LiSa are incomplete. Additionally, dynamically linked templates running on LiSa often fail due to the lack of proper libraries.

In Table 3, we summarize the descriptions and supported features (described in Table 2) of these IoT sandboxes. The main drawbacks of above sandboxes are:

TABLE 3. Describe the IoT Sandboxes.

Sandbox	Collected data	OS support	CPU support	Base-on	Supported features
DroidScope [30]	APIs	Android	ARM, i386	QEMU	1-9
AASandbox [31]	System calls	Android	ARM	Android SDK	1-6
Cuckoo [35]	System call, File actions, Memory dumps, Network traffic, Screenshots	Windows, Linux, Android	i386, X86-64, PPC, Sparc, ARM, MIPS, MIPSEL	Virtual Box, KVM, VMware	1-14, 41-48
IoTBOX [36]	Network traffic	Linux, OpenWrt	MIPS, MIPSEL, ARM	QEMU, OpenWrt	10-14
Limon [38]	Process, File system, Network, Systemcall	Linux	i386, x86-64	VMware	1-14, 29-40
REMnux [39]	Syscalls	Ubuntu	i386, x86-64	VMware	1-6
Detux [37]	Network traffic	Linux	i386, x86-64, ARM, MIPS, MIPSEL	QEMU	10-14
Padawan [40]	Low-level kernel probes and user probes	Linux	MIPS, ARM, i386, x86-64, Sparc, PPC	QEMU	1-6, 29-48
LiSa [41]	Process tree, Trace syscalls, File actions, Network traffic	Linux	MIPS, ARM, Intel 80386, x86-64, Aarch64	QEMU	1-14, 29-40

1) Features (shown in Table 2) are not fully exhaustive collected during executable file runtime;
2) Do not support communication with the C&C server;
3) Do not provide shared libraries.

Therefore, we will build an adequate sandbox that addresses these drawbacks. The proposed approach is presented in the next section.

III. THE PROPOSED V-SANDBOX

To give a better understanding of our proposed sandbox, we use a sample of IoT Botnet Mirai through each step of the data collection process. Mirai is one of the most popular IoT Botnet families, causing DDoS attacks with ground up to 1.1 Tbps of traffic in September 2016 [1]. Mirai easily infects IoT devices such as webcams, DVRs, routers, and IP cameras. There are times when the number of simultaneously infected devices ground to 400000.

A. OVERVIEW

Resource-constrained IoT devices play an important role in trend Internet of Things and are vulnerable to attacks. When analyzing and defending against these attacks, C&C servers connection, shared libraries for dynamic files and a wide range of CPU architecture and other capabilities and indicators are very necessary. However, the existing sandbox cannot satisfy them. Therefore, V-sandbox is proposed to solve this problem. The proposed approach described in Fig. 6, consists of 8 main components:

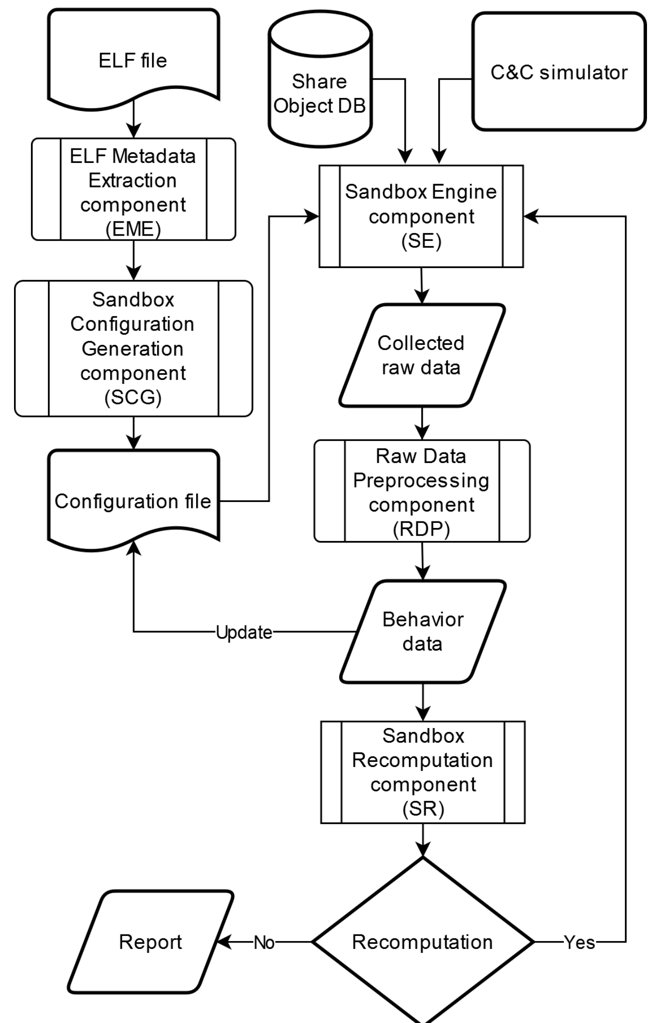

FIGURE 6. The architecture of V-Sandbox.

- ELF Metadata Extraction component (EME): automatically extracts the metadata from the input file.
- Sandbox Configuration Generation component (SCG): Create the appropriate configuration parameters of the *"Sandbox Engine component"* to execute the ELF file.
- Sandbox Engine component (SE): The appropriate sandbox environment allows behavior monitoring and facilitates the ELF executable file to exhibit the behavior fully.
- Raw Data Preprocessing component (RDP): analyzes the typical behaviors of the ELF executable file, provides data supporting the *"Sandbox Recomputation component"* making choices.
- Sandbox Recomputation component (SR): decide whether to export a report about ELF file behaviors

or need to rerun the *"Sandbox Engine component"* to collect more behavioral data.

- C&C simulator: C&C server emulator provides the ability to make connections, transmit and receive C&C commands.
- Share Object DB: a database of dynamic link libraries that the ELF executable file requires.
- Report: Generates a summary of the behavior that the ELF executable file exhibits in the *"Sandbox Engine component."*

First, the *"ELF Metadata Extraction component"* reads the ELF file header to extract information about the CPU and OS architecture needed to run. Then, the *"Sandbox Configuration Generation component"* relies on output information of the EMF block to create one of the basic configurations for V-Sandbox (output data is *"Configuration file"*). This *"Sandbox Engine component"* will execute and collect the raw behavior data of the ELF file using agents with the configuration in the *"Configuration file."* Collected raw data from these agents will be transferred to *"Raw Data Preprocessing component"*. Here, the behavior data of the ELF file is read and analyzed to update the *"Configuration file"*, including the IP address of the C&C server and the missing libraries. In addition, these behavioral data support the *"Sandbox Recomputation component"*, deciding to rerun the *"Sandbox Engine component"* or stop through the analyzer and generate the analysis report. In this case the *"Sandbox Recomputation component,"* decides to rerun the *"Sandbox Engine component,"* with support from the *"Configuration file," "C&C simulator"* and *"Shared Object DB"*, we can gather more data about the behavior of the ELF file. Finally, when the *"Sandbox Recomputation component"* determines it is not able to gather more behavioral information of the ELF file, it will generate a report about the behavioral data of the ELF file.

B. ELF METADATA EXTRACTION COMPONENT

For the SE component to launch the input file, the EME component needs to provide the following information:

- Format of file: Is the input file in ELF format or not? If it's ELF format then is it the 32-bit or 64-bit format?
- Type of file: Can the input file be executable?
- CPU architecture requirement: Determine what the CPU architecture needs to be run.
- Linux kernel requirement: Determine the required Linux kernel version.
- Statically or Dynamically linked: Determine the list of library files needed.

This information can be extracted from the metadata of the input file. The environment in which our V-Sandbox focuses on Unix includes GNU/Linux and BSD. The format of executable files in this environment is ELF, the Executable and Linkable Format. The basic architecture of the ELF file is shown in Fig. 7. ELF files are divided into three types [63]:

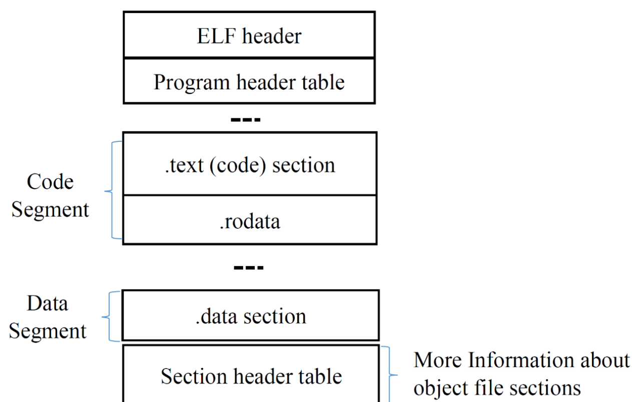

FIGURE 7. Basic architecture of ELF file format [64].

- Executable files: containing code and data suitable for execution. This specifies the memory layout of the process image of the program.
- Relocatable (object) file: containing code and data suitable for linking with other object files to create an executable or a shared object file.
- Shared object files (shared library): containing code and data suitable for the link editor (ld) at the link-time and the dynamic linker (ld.so) at runtime.

In particular, metadata of ELF is recorded mainly in the header. The ELF header is 32 bytes long and identifies the format of the file. It starts with a sequence of four unique bytes that are $0 \times 7F$ followed by 0×45, $0 \times 4c$, and 0×46 which translates into the three letters E, L, and F. Among other values, the header also indicates whether it is an ELF file for 32-bit or 64-bit format, executable or not, uses little or big endianness, shows the ELF version as well as for which operating system the file was compiled for in order to interoperate with the right application binary interface (ABI) and cpu instruction set. An illustrative example of the ELF header values described in Table 4.

To read the values in the ELF header, Linux provides many powerful tools such as *"Readelf", "File utility", "Elfdump", "Elfutils", "Objdump", "Scanelf", "Elfkickers", etc.*

Debian GNU/Linux offers the *"Readelf"* command that is provided in the GNU "binutils" package.[2] Oo *et al.* [64] uses *"Readelf"* with option *-h* (short version for *"–file-header"*), and it nicely displays the header of an ELF file. Fig. 11 shows this for our sample.

Elfdump is a command-line utility that provides detailed information about ELF binaries.[3] It works with executables, shared libraries and even relocatable object files. Wiederseiner [65] uses *"elfdump"* to generates the symbol table of the code-base and source code line data of the ELF file. Wong *et al.* [66] uses *"elfdump", "objdump"* and *"readelf"*

[2]"Readelf - Linux manual page." [Online]. Available: http://man7.org/linux/man-pages/man1/readelf.1.html. Accessed on: Apr. 12, 2020.

[3]"Elfdump - Sunos man page." [Online]. Available: http://man7.org/linux/man-page/sunos/1/elfdump/. Accessed on: Apr. 12, 2020.

TABLE 4. Fields of ELF header.

Field	Values	Explanation
ei_mag	0x7F, "ELF"	Constant signature
ei_class	1	Identifies the file's class, or capacity (1 - 32 bits, 2 – 64 bits)
ei_data	1	Specifies the encoding of both the data structures used by object file container and data contained in object file sections (1 – Little endian, 2 – Big endian)
ei_version	1	ELF indentifine version (always 1)
e_type	2	This member identifies the object file type (1 - Relocatable, 2 - Excutable, 3 - Shared object, etc)
e_machine	28	The required architecture for an individual file (2 – Sparc, 3 – Intel 80386, 8 – MIPS, 28 - ARM processor, etc)
e_version	1	ELF version (always 1)
e_entry	0x8000060	Address where execution starts
e_phoff	0x40	Program headers' offset
e_shoff	0xB0	Section headers' offset
e_ehsize	0x34	ELF header's size
e_phentsize	0x20	Size of a single program header
e_phnum	1	Count of program headers
e_shentsize	0x28	Size of a single section header
e_shnum	4	Count of section headers
e_shstrndx	3	Index of the names' section in the table

to extract ELF file format constraints. The ELFDump version has not had a new release or update since 2003.

Objdump is similar to readelf, but focuses on object files.[4] It provides a similar range of information about ELF files and other object formats. However, it is inefficient to support CPU architectures like MIPS, ARM, and PPC. In Fig. 8, this tool does not detect the CPU architecture of our sample.

Elfutils is a collection of utilities, including *eu-ld* (a linker), *eu-nm* (for listing symbols from object files), *eu-size* (for listing the section sizes of an object or archive file), *eu-strip* (for discarding symbols), *eu-readelf* (to see the raw ELF file structures), and *eu-elflint* (for checking for well-formed ELF files).[5] It provides alternative tools to GNU Binutils, and also allows validating ELF files. Note that all the names of the utilities provided in the package start with eu for *"elf utils"*.

Pax-utils is a set of utilities that provide some tools that help to validate ELF files.[6] *Scanelf* is one of this set utilities. In Fig. 9 showed how *scanelf* work with an option "-a".

There is also a software package called *"Elfkickers"* which contains tools to read the contents of an ELF file as well as manipulate it.[7] Unfortunately, the number of releases is rather low, and that is why we mention it and do not show further examples.

"File" utility displays information about ELF files, including the instruction set architecture for which the code in a

[4] "Objdump - Linux manual page." [Online]. Available: http://man7.org/linux/man-pages/man1/objdump.1.html. Accessed on: Apr. 12, 2020.

[5] "Debian – Details of package elfutils in sid." [Online]. Available: https://packages.debian.org/sid/elfutils. Accessed on: Apr. 12, 2020.

[6] "Debian – Details of package pax-utils in sid." [Online]. Available: https://packages.debian.org/sid/misc/pax-utils. Accessed on: Apr. 12, 2020.

[7] "ELF Kickers software" [Online]. Available: https://www.muppetlabs.com/~breadbox/software/elfkickers.html. Accessed on: Apr. 12, 2020.

```
ais:sample$ objdump -f a7192c394957ba17878e3c1f57aca67b

a7192c394957ba17878e3c1f57aca67b:     file format elf32-little
architecture: UNKNOWN!, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x004020c0
```

FIGURE 8. Use *Objdump* utility to display the contents of the overall file header.

```
ais:sample$ scanelf -a a7192c394957ba17878e3c1f57aca67b
 TYPE    PAX   PERM ENDIAN STK/REL/PTL TEXTREL RPATH BIND FILE
ET_EXEC PeMRxS 0644 LE RWX --- RW-    -       -    LAZY a7192c394957b
a17878e3c1f57aca67b
```

FIGURE 9. Use *scanelf* to scan ELF binaries.

```
ais:sample$ file a7192c394957ba17878e3c1f57aca67b
a7192c394957ba17878e3c1f57aca67b: ELF 32-bit LSB executable, MIPS,
MIPS-I version 1 (SYSV), dynamically linked, interpreter /lib/ld-,
stripped
```

FIGURE 10. Use *File* utility to read the file header.

relocatable, executable, or shared object file is intended.[8] In Fig. 10 shows that our sample is a 32-bit executable file following the Linux Standard Base (LSB), dynamically linked, and built for the MIPS architecture.

With the required information from SE, the author chooses to use the *"readelf"* utility for two reasons. First, the *"readelf"* utility provides complete information required by the EMF. Second, the *"readelf"* utility is built into the easy-to-use versions of Linux, without having to install additional external packages.

Use the *"Readefl"* utility to determine the file attributes shown in Figs. 11 and 12. Fig. 11 shows that the malware sample is a 32-bit executable ELF file following the Little Endian, and built for the Unix – System V, MIPS architecture. Fig. 12 shows a list of shared object requirements. Note that this is a list of shared objects that ELF requires when reading the file header, so it is incomplete. This issue will be discussed further in section III-D. This information is sent to the *"Sandbox Configuration Generation component"* to select the appropriate virtual machine to execute the ELF

[8] "File - Linux manual page." [Online]. Available: http://man7.org/linux/man-pages/man1/file.1.html. Accessed on: Apr. 12, 2020.

```
ais:sample$ readelf -h a7192c394957ba17878e3c1f57aca67b
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           MIPS R3000
  Version:                           0x1
  Entry point address:               0x4020c0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          70728 (bytes into file)
  Flags:                             0x1007, noreorder, pic, cpic, o32, mips1
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         6
  Size of section headers:           40 (bytes)
  Number of section headers:         21
  Section header string table index: 20
```

FIGURE 11. Use *Readelf* to read the ELF header file.

TABLE 5. Default configuration parameter for the sandbox engine.

CPU Arch.	Arch. version	Endianess	Emulated machine	Kernel [67]	RAM	Image virtual disk [67]
ARM	ARMv5l (32 bit)	Little	versatilepb	vmlinuz-3.2.0-4-versatile	128 MB	debian_wheezy_armel_standard.qcow2
MIPS	MIPS64 (64 bit)	Big	malta	vmlinux-3.2.0-4-5kc-malta	128 MB	debian_wheezy_mips_standard.qcow2
MIPS	MIPS32 (32 bit)	Big	malta	vmlinux-3.2.0-4-4kc-malta	128 MB	debian_wheezy_mips_standard.qcow2
MIPSEL	MIPS64 (64 bit)	Little	malta	vmlinux-3.2.0-4-5kc-malta	128 MB	debian_wheezy_mipsel_standard.qcow2
MIPSEL	MIPS32 (32 bit)	Little	malta	vmlinux-3.2.0-4-4kc-malta	128 MB	debian_wheezy_mipsel_standard.qcow2
Intel 80386	Intel 80386 (32 bit)	Little	pc-i440fx-xenial	NA	128 MB	debian_wheezy_i386_standard.qcow2
x86-64	x86-64 (64 bit)	Little	pc-i440fx-xenial	NA	128 MB	debian_wheezy_amd64_standard.qcow2
PowerPC	PowerPC (32 bit)	Big	g3beige	NA	128 MB	debian_wheezy_powerpc_standard.qcow2
SH4	SH4 (32 bit)	Little	r2d	vmlinuz-2.6.32-5-sh7751r	64 MB	debian_sid_sh4_standard.qcow2
SPARC	SPARC (32 bit)	Big	SS-10	NA	128 MB	debian_etch_sparc_small.qcow2

```
sandboxmips:sample$ readelf -d a7192c394957ba17878e3c1f57aca67b

Dynamic section at offset 0x108 contains 22 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.0]
 0x0000000c (INIT)                       0x401ef4
 0x0000000d (FINI)                       0x4105e0
 0x00000004 (HASH)                       0x4001e0
 0x00000005 (STRTAB)                     0x401514
 0x00000006 (SYMTAB)                     0x400834
 0x0000000a (STRSZ)                      2525 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x70000016 (MIPS_RLD_MAP)               0x451050
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x451060
 0x00000011 (REL)                        0x0
 0x00000012 (RELSZ)                      0 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x70000001 (MIPS_RLD_VERSION)           1
 0x70000005 (MIPS_FLAGS)                 NOTPOT
 0x70000006 (MIPS_BASE_ADDRESS)          0x400000
 0x7000000a (MIPS_LOCAL_GOTNO)           15
 0x70000011 (MIPS_SYMTABNO)              206
 0x70000012 (MIPS_UNREFEXTNO)            23
 0x70000013 (MIPS_GOTSYM)                0xb
 0x00000000 (NULL)                       0x0
```

FIGURE 12. Use *Readelf* to read the list of shared library requirements.

```
a7192c394957ba17878e3c1f57aca67b.argconfig (~/Desktop) - gedit
Open
ID: a7192c394957ba17878e3c1f57aca67b
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           MIPS R3000
  Version:                           0x1

Shared library:
 0x00000001 (NEEDED)                Shared library: [libc.so.0]
```

FIGURE 13. Output file from EME component.

file. The output of the EME component is a file ending in *"argconfig"* as illustrated in Fig. 13.

C. SANDBOX CONFIGURATION GENERATION COMPONENT

With the EME component output (*".argconfig"* file), the *"Sandbox Configuration Generation component"* (SCG) proceeds to generate the initial configuration (*".config"* file) to launch the *"Sandbox Engine component"* for the first time. These configuration parameters will be passed to QEMU to initialize the sandbox engine along with some other default environment parameters. Default configuration parameters of the sandbox engine shown in Table 5. Also, the list of shared libraries in the *".argconfig"* file was also added to the *".config"* file with the default path of *"/lib/..."* to be able to automatically add the Shared object from the database to the Image virtual disk after starting QEMU. The architecture of the *".config"* file is illustrated in Fig. 14.

```
a7192c394957ba17878e3c1f57aca67b.config (~/Desktop) - gedit
Open
ID: a7192c394957ba17878e3c1f57aca67b

QEMU arg:
        command:        qemu-system-mips
        Machine:        malta
        Kernel:         vmlinux-3.2.0-4-4kc-malta
        HDA:            debian_wheezy_mips_standard.qcow2
        RAM:            128

Shared library:
libc.so.0 => /lib/libc.so.0

IP C&C server:
No
```

FIGURE 14. Output file from SCG block.

D. SANDBOX ENGINE COMPONENT

The IoT Botnet is built to run on many different CPU architectures to suit the variety of IoT devices. To be able to run and analyze IoT Botnet models on different CPU architectures in a simulation environment, we choose to use QEMU [62]. The open-source project QEMU supports many CPU architectures such as ARM, MIPS, PowerPC, SPARC, etc. Inside each of the Sandbox Environment, there is a built-in Debian image of Aurel [67] with QEMU host, the agents that manage, monitor, and collect ELF files are executed (illustrated in Fig. 16).

The configuration parameters to launch the QEMU host, are read from the *".config"* file, including command, emulated machine type, Kernel, virtual disk image, RAM, file name, and path of the shared library, IP address of the C&C server. The C&C simulator uses these IP addresses to create a connection between the QEMU host and the C&C server simulation. The file name and path of the shared library are used to add the missing shared libraries from the shared object database to the virtual disk image. The QEMU host


```
sandboxmips@sandboxmips-virtual-machine: ~/Desktop/sample
sandboxmips:sample$ qemu-system-mips -M malta -kernel vmlinux-3.2.0-
4-4kc-malta -hda debian_wheezy_mips_standard.qcow2 -m 128 -append "r
oot=/dev/sda1" -net nic -net tap -nographic -snapshot
```

FIGURE 15. QEMU host boot command for 32-bit MIPS architecture Little Debian.

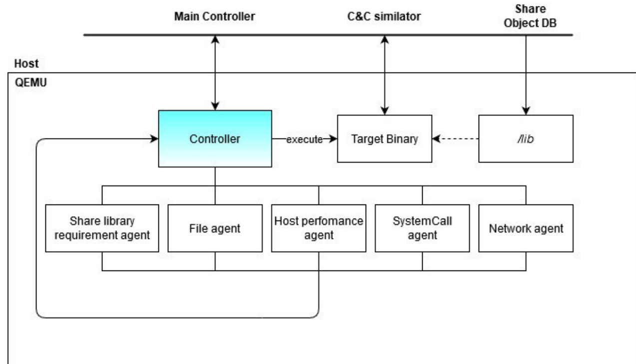

FIGURE 16. Architecture inside virtual host.

starts running with the configuration parameters, as shown in Fig. 15.

The agents are programmed based on the *C* language and cross-compiled by *Toolchains*[9] to the corresponding CPU architectures. A Toolchain is a set of distinct software development tools that are linked (or chained) together by specific stages such as *GCC, Binutils* and *Glibc* (a portion of the GNU Toolchain). Optionally, a toolchain may contain other tools such as a debugger or a compiler for a specific programming language, such as *C* language. Quite often, the toolchain used for embedded development is a cross toolchain, or more commonly known as a cross compiler. All the programs (like *GCC*) run on a host system of a specific architecture (such as x86), but they produce binary code (executables) to run on a different architecture (for example, ARM). This is called cross compilation and is the typical way of building embedded software. Verma *et al.* [68] demonstrates how to use *Toolchains* to create applications that run on IoT devices with different CPU architectures.

With the features of the IoT botnet presented in the subsection II-A, the sandbox environment will be integrated with tools to monitor malware behaviors, including system-calls, file and directory activity, host performance requirements, shared library, and network behaviors. For this, we have agents in each Sandbox Engine, including:

- **The controller agent:** managing common tasks of information gathering agents, activating ELF file execution, receiving and transmitting collected information to the “Analysis behavior” block.
- **File agent:** collecting behaviors related to files and directories (base on *“lsof”* utility). Everything on disk is a file. Normal files, devices and even directories are all presented as a file. The file system marks each of these entries in a file table, with the related type. To see open files, we can use the *“lsof”* utility, *“inotify”, “fswatch”,* etc. The *“lsof”* utility stands for “list open files” and definitely reveals its purpose.[10] It can show any open files, from the earlier mentioned special files (block and character devices), to tracking open network connections. *“Inotify”* in *“inotify-tools”* is part of the Linux kernel that triggers events on watched files, directories, or even the contents of entire directories.[11] These tools are command-line utilities that tap into the capabilities of inotify and allow you to use them, for example, in your shell scripts. *“Fswatch”* is a free, open-source multi-platform file change monitor utility that notifies us when the contents of the specified files or directories are modified or changed.[12] Using *“fswatch”* can easily monitor the changes being made in files and/or directories. We choose to use the *“lsof”* utility to monitor the interaction with files and directories of ELF files for two reasons. It is also quite useful to figure out what files are being accessed by what processes. Also, *“lsof”* is easy to use on different CPU architecture platforms because it is built into the Linux kernel. The monitoring results of the *“file agent”* are illustrated in Fig. 17.
- **Host performance agent:** collects behaviors related to using system resources such as CPU, RAM (base on *“top”* utility). To monitor host performance, there are various tools such as *“top”*,[13] *“iostat”*[14] and *“mpstat”*,[15] etc. However, *“top”* utility is more suitable for this problem. *“Top”* utility displays Linux processes. It provides a dynamic real-time view of the running system, id est actual process activity. By default, it dis-

```
vsandboxarm1@vsandboxarm1-virtual-machine: ~/Downloads
vsandboxarm1:Downloads$ lsof -p 6350
COMMAND  PID          USER  FD      TYPE           DEVICE SIZE/OFF    NODE
 NAME
gedit   6350 vsandboxarm1  cwd      DIR               8,1     4096 2490370
 /home/vsandboxarm1
gedit   6350 vsandboxarm1  rtd      DIR               8,1     4096       2
 /
gedit   6350 vsandboxarm1  txt      REG               8,1    10536 2884003
 /usr/bin/gedit
gedit   6350 vsandboxarm1  mem      REG               8,1   205748  657376
 /usr/share/fonts/truetype/ubuntu-font-family/UbuntuMono-R.ttf
gedit   6350 vsandboxarm1  mem      REG               8,1   426456 2883690
 /usr/lib/x86_64-linux-gnu/libibus-1.0.so.5.0.511
gedit   6350 vsandboxarm1  mem      REG               8,1    31784  131224
 /usr/lib/x86_64-linux-gnu/gtk-3.0/3.0.0/immodules/im-ibus.so
gedit   6350 vsandboxarm1  DEL      REG               0,5            5603349
 /SYSV00000000
```

FIGURE 17. Results of running *lsof* with a process with PID 6350.

[9]“Toolchains - eLinux.org.” [Online]. Available: https://elinux.org/Toolchains. Accessed on: Mar. 12, 2020.

[10]“Lsof - Linux manual page.” [Online]. Available: http://man7.org/linux/man-pages/man8/lsof.8.html. Accessed on: Apr. 13, 2020.

[11]“Inotify - Linux manual page.” [Online]. Available: http://man7.org/linux/man-pages/man7/inotify.7.html. Accessed on: Apr. 13, 2020.

[12]E. M. Crisostomo, “emcrisostomo/fswatch,” [Online]. Available: https://github.com/emcrisostomo/fswatch. Accessed on: Apr. 13, 2020.

[13]“Top - Linux manual page.” [Online]. Available: http://man7.org/linux/man-pages/man1/top.1.html. Accessed on: Apr. 13, 2020.

[14]“Iostat(1) - Linux manual page.” [Online]. Available: http://man7.org/linux/man-pages/man1/iostat.1.html. Accessed on: Apr. 13, 2020.

[15]“mpstat - Linux manual page.” [Online]. Available: http://man7.org/linux/man-pages/man1/mpstat.1.html. Accessed on: Apr. 13, 2020.

```
top - 15:12:24 up 13 days, 23:47,  1 user,  load average: 0,89, 0,33, 0,16
Tasks: 178 total,   1 running, 142 sleeping,   0 stopped,   0 zombie
%Cpu(s): 17,0 us, 11,4 sy,  0,0 ni, 71,1 id,  0,0 wa,  0,0 hi,  0,6 si,  0,0 st
KiB Mem :  2041572 total,   116044 free,  1315024 used,   610504 buff/cache
KiB Swap:   998396 total,   806396 free,   192000 used.   523780 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
20989 vsandbo+  20   0 1054460 118468  35104 S  5,6  5,8  65:49.84 anydesk
 2179 vsandbo+  20   0 1417384 332884  49288 S  4,7 16,3   2308:49 compiz
  872 root      20   0  472160 117840  29812 S  3,3  5,8 750:32.46 Xorg
  768 root      20   0  686596  13812   9328 S  1,7  0,7   2:53.19 anydesk
 2128 vsandbo+   9 -11  363028   5196   4500 S  1,0  0,3   1:25.85 pulseaudio
 2229 vsandbo+  20   0  990968  50808  25976 S  0,7  2,5   0:55.02 nautilus
    7 root      20   0       0      0      0 S  0,3  0,0   0:26.86 ksoftirqd/0
 7146 vsandbo+  20   0   50396   4156   3552 R  0,3  0,2   0:00.04 top
    1 root      20   0  119868   4240   2608 S  0,0  0,2   0:08.10 systemd
    2 root      20   0       0      0      0 S  0,0  0,0   0:00.16 kthreadd
    4 root       0 -20       0      0      0 I  0,0  0,0   0:00.00 kworker/0:0H
    6 root       0 -20       0      0      0 I  0,0  0,0   0:00.00 mm_percpu_wq
    8 root      20   0       0      0      0 I  0,0  0,0  22:37.56 rcu_sched
    9 root      20   0       0      0      0 I  0,0  0,0   0:00.03 rcu_bh
   10 root      rt   0       0      0      0 S  0,0  0,0   0:00.00 migration/0
   11 root      rt   0       0      0      0 S  0,0  0,0   0:03.40 watchdog/0
   12 root      20   0       0      0      0 S  0,0  0,0   0:00.00 cpuhp/0
```

FIGURE 18. Monitor host perfomance by *top* command.

plays the most CPU-intensive tasks running on the server and updates the list every five seconds (result monitor by *"top"* command showed in Fig. 18). *"Iostat"* utility report Central Processing Unit (CPU) statistics and input/output statistics for devices, partitions, and network filesystems (NFS). *"Mpstat"* command displays activities for each available processor, processor 0 being the first one. Command *"mpstat -P ALL"* is used to display average CPU utilization per processor. With the ability to provide all the necessary information and ease of use with many different CPU architectures, the host performance agent is built on top of the *"top"* utility.

- **SystemCall agent:** collects system-call of ELF file (based on *"strace"* utility[16]). To better understand the behavior of the ELF file when executed, the most useful information is its own system-calls. In Linux, there are many tools that support collecting system-call of an executable file (such as *ftrace*,[17] *ptrace*,[18] etc.) but basically function like *"strace"* utility. *"Strace"* monitors the system calls and signals of a specific program. It is helpful when you do not have the source code and would like to debug the execution of a program. This tool provides the execution sequence of a binary from start to end. The result of collecting the system calls of the *"ls"* program by *Strace* is illustrated in Fig. 19. In the Limon sandbox, Monnappa uses *"Strace"* for tracing system calls.
- **Shared library requirement agent:** In particular, when building lists of shared libraries in addition to *"readelf"* we also use the *"ldd"* utility [66]. Command *"ldd"* prints the shared objects (shared libraries) required by each program or shared object specified on the command line. Unlike the *"readelf"* utility in the EMF block (section III-B) that only analyzes static ELF file content to identify shared libraries, *"ldd"* executes the ELF file right in the sandbox environment to identify

[16]"Strace - Linux manual page." [Online]. Available: http://man7.org/linux/man-pages/man1/strace.1.html. Accessed on: Apr. 13, 2020.
[17]"Ftrace - eLinux.org." [Online]. Available: https://elinux.org/Ftrace. Accessed on: Apr. 13, 2020.
[18]"Ptrace - Linux manual page." [Online]. Available: http://man7.org/linux/man-pages/man2/ptrace.2.html. Accessed on: Apr. 13, 2020.

```
vsandboxarm1:~$ strace /bin/ls
execve("/bin/ls", ["/bin/ls"], [/* 71 vars */]) = 0
brk(NULL)                               = 0x18c8000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=92277, ...}) = 0
mmap(NULL, 92277, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f1b2557e000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260Z\0\0\0\0\0\0"..., 83
2) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=130224, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f1
b2557d000
mmap(NULL, 2234080, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f
1b2514e000
mprotect(0x7f1b2516d000, 2093056, PROT_NONE) = 0
mmap(0x7f1b2536c000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYW
RITE, 3, 0x1e000) = 0x7f1b2536c000
mmap(0x7f1b2536e000, 5856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONY
MOUS, -1, 0) = 0x7f1b2536e000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832)
 = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1868984, ...}) = 0
mmap(NULL, 3971488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f
1b24d84000
```

FIGURE 19. System calls of the *"ls"* program by *Strace*.

the shared libraries required by ELF. This has resulted in higher efficiency for identifying shared libraries and has been demonstrated experimentally on Dataset when compared to other sandbox capabilities (such as LiSa [41]). The results of the list of shared libraries by *"ldd"* and *"readelf"* are illustrated as shown in Fig. 20 and Fig. 21 (note that *ldd* identifies two more shared libraries required is libc.so.6 and ld-uClibc.so.0 and the full directory path rather than using *readelf*).

```
root@debian-mips:~/qemu# ldd 8b269f0eab1e09040c62ce78dff05c01
        libgcc_s.so.1 => /lib/mips-linux-gnu/libgcc_s.so.1 (0x77e44000)
        libc.so.0 => not found
        libc.so.6 => /lib/mips-linux-gnu/libc.so.6 (0x77cbc000)
        /lib/ld-uClibc.so.0 => /lib/ld.so.1 (0x55550000)
```

FIGURE 20. The list of shared libraries required is determined by the *ldd* utility.

```
ais@ais-virtual-machine:~/Desktop/compare/final_report/8b269f0eab1e09040c62
ff05c01_1585982654$ readelf -d ~/Downloads/botnet/benign/8b269f0eab1e09040c
8dff05c01

Dynamic section at offset 0x160 contains 23 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libgcc_s.so.1]
 0x00000001 (NEEDED)                     Shared library: [libc.so.0]
 0x0000000c (INIT)                       0x401174
 0x0000000d (FINI)                       0x419b50
 0x00000004 (HASH)                       0x400240
 0x00000005 (STRTAB)                     0x400b9c
 0x00000006 (SYMTAB)                     0x40055c
```

FIGURE 21. The list of shared libraries required is determined by the *readelf* utility.

- **Network agent:** collects network behavior of the ELF file (based on program *"TCPdump"*[19]). One of the critical behaviors of IoT Botnet malware is connecting to a bot network and receiving commands from Bot master (C&C server) - these are typical network behaviors of IoT Botnet. So the "Network agent" is built to collect the network behavior of the input ELF file using the *"TCPdump"* utility, which is a simple tool that dumps

[19]T. T. Group, "Tcpdump/Libpcap public repository." [Online]. Available: https://www.tcpdump.org. Accessed on: Mar. 25, 2020.

```
vsandboxarm1:~$ sudo tcpdump -i ens160 -c 5
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens160, link-type EN10MB (Ethernet), capture size 262144 bytes
15:59:57.506166 IP relay-f6e659b6.net.anydesk.com.http > 192.168.1.34.40691: Fla
gs [.], ack 2970710656, win 8098, options [nop,nop,TS val 2497445698 ecr 1061253
251], length 0
15:59:57.506683 IP 192.168.1.34.40526 > net.fpt.domain: 46676+ PTR? 34.1.168.192
.in-addr.arpa. (43)
15:59:57.507452 IP net.fpt.domain > 192.168.1.34.40526: 46676 NXDomain 0/0/0 (43
)
15:59:57.507575 IP 192.168.1.34.43066 > net.fpt.domain: 4372+ PTR? 81.116.128.17
8.in-addr.arpa. (45)
15:59:57.512062 IP 192.168.1.34.40691 > relay-f6e659b6.net.anydesk.com.http: Fla
gs [P.], seq 48:95, ack 0, win 501, options [nop,nop,TS val 1061253311 ecr 24974
45698], length 47: HTTP
5 packets captured
8 packets received by filter
0 packets dropped by kernel
```

FIGURE 22. **The network traffic dump by the *TCPdump* utility.**

and reads traffic on a network. This tool has been used in many network intrusion detection systems (NIDS) [69]–[72]. The result of the network traffic dump using this tool is shown in Fig. 22.

E. RAW DATA PREPORCESSING COMPONENT

In this paper, the type of IoT malware we focus on analyzing is the IoT Botnet. As stated in section II-A, Angrishi [8] and Nguyen *et al.* [19] also present the features of IoT Botnet including scanning for suitable targets, accessing other vulnerable devices, infecting IoT devices, communicating with C&C servers via the IP address of the URL, waiting and executing commands from the C&C server. Therefore, to understand the behavior of IoT Botnet, the sandbox must analyze the collected raw data from the sandbox, including network behavior (send requests and connect to the C&C server, scan other IoT devices via telnet, ssh port), system calls (download the binary file from a server, execute downloaded binary file, brute force account other vulnerable devices, call loop pending connection to the C&C server, etc.), changes to files and directories, requires the use of shared libraries, takes up system resources.

For this reason, the RDP component analyzes collected raw data from the *"Sandbox Engine component"* to serve as input for the rerun decision sandbox algorithm, including:

- Network traffic: Extract necessary information such as *destination IP address, connection protocol, connection port, string data (if not encrypted).*
- System calls: With information from system calls, pinpoint the behavior of parent and child processes generated by the ELF file. This agent supplements the ELF file behavior data that is trying to be performed as executed file (with system call *execve(), etc.*), create subprocess (with system call *fork(), vfork(), clone(), etc.*), connect to C&C server (by a system call *connect(), etc.*), call shared library (system call *access(), open(), read(), etc.*), wait for commands from the C&C server when the connection is successful (system call *wait(), sleep(), etc.*), override or create an infected file version (system call *write()*), disable other processes (by calling *exit(), _exit()*), etc. The information extracted includes *the name of the system call, the parameters passed, and the order of this call.*
- File activity: For information from the activity file, the "Analysis behavior" focuses on identifying the files and folders that ELF files interact (*open, write, rewrite, delete, rename, create, changer permission*) including both shared library files (if applicable). The extracted information includes *the name of the file, the directory path containing the file, and the action.*
- Host performance: Extract CPU and RAM information such as *total resource utilization rate, resource utilization rate* of each process related to the ELF file over time.

The analysis results for this com have been updated to *".config"* file, as shown in Fig. 23.

```
a7192c394957ba17878e3c1f57aca67b.config (~/Desktop) - gedit
Open ▾
ID: a7192c394957ba17878e3c1f57aca67b

QEMU arg:
        command:        qemu-system-mips
        Machine:        malta
        Kernel:         vmlinux-3.2.0-4-4kc-malta
        HDA:            debian_wheezy_mips_standard.qcow2
        RAM:            128

Shared library:
libc.so.0 => /lib/libc.so.0
ld-uClibc.so.0 => /lib/ld-uClibc.so.0

IP C&C server:
195.62.53.43
```

FIGURE 23. **Output *".config"* file after update.**

F. SANDBOX RECOMPUTATION COMPONENT

The "Sandbox Recomputation component" uses data from the output of the RDP component including total system calls (*ts*), total network traffic packets (*tntp*), total file name request (*tfr*), the average percentage of CPU used (*avgCPU*), the average percentage of RAM used (*avgRAM*). Every time we run the *"Sandbox Engine component"*, these values are extracted from behavioral data. If the total of these values does not increase, we will decide to stop running the SE again. The result of this function is whether or not it is necessary to rerun the *"Sandbox Engine component"* again or not to serve the purpose of collecting more information about ELF file behavior. The algorithm of the *"Sandbox Recomputation component"* is described in Algorithm 1.

G. THE C&C SIMULATOR

The command and control (C&C) server is an essential component in a botnet in general and IoT botnet in particular. The C&C server provides the botmaster with a centralized management way to monitor botnet status and order new DDoS attacks to be carried out. The protocol used to connect between the C&C server and the bot is usually IRC, HTTP, P2P, etc. The illustration of the connections of components in the IoT botnet is illustrated by Angrishi as shown in Fig. 24. In order for the IoT bot to display its entire behavior, it is necessary to create a connection between the bot and the

Algorithm 1 Algorithm for Sandbox Recomputation Component

Input: Extract from Behavior data(
Total system calls: *ts*,
Total network traffic packets: *tntp*,
Total file name request: *tfr*,
The average percentage of CPU used: *avgCPU*,
The average percentage of RAM used: *avgRAM*)
Output: Decision of rerun the sandbox: *d*
$d = true, t = 0, i = 0, n = 100$
1: $v_t \leftarrow (ts + tntp + tfr + avgCPU + avgRAM)$
2: **while** d **do**
3: $t \leftarrow t + 1$
4: Run sandbox
5: Extract from Behavior data $(ts, tntp, tfr, avgCPU, avgRAM)$
6: $v_t \leftarrow (ts + tntp + tfr + avgCPU + avgRAM)$
7: **if** $v_t \leq v_{t-1}$ or $t = n$ **then**
8: $d \leftarrow false$
9: **end if**
10: **end while**
11: **return** d

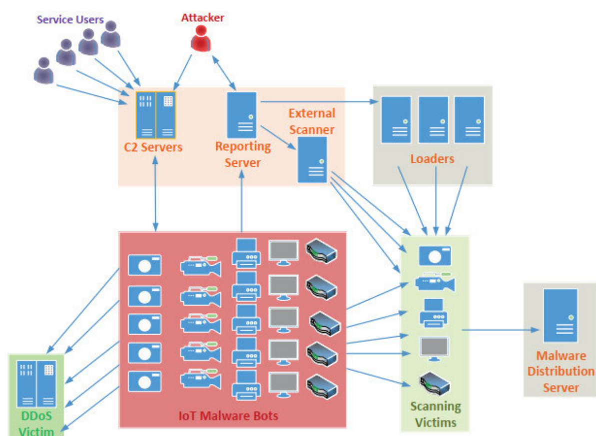

FIGURE 24. Generic structure of an IoT Botnet [8].

C&C server. The C&C server must then transmit the appropriate commands for the IoT bot to execute. These actions are the main contribution of the C&C simulator. The C&C simulator proceeds to generate C&C servers based on the set of IP addresses in the *".config"* file (showed in Fig. 23). A database of C&C commands is gathered from various sources, such as source code for IoT Botnet [73], [74], and studies [36], [42], [48]. If the listed C&C server's IP address is not empty, the automatic script creates C&C servers with these IP addresses. When a bot is connected to this C&C server, C&C commands which form our database will be sent to bot. We are not sure what C&C commands are useful for this bot. We are sending all commands and hope to activate some of the botnets next behavior. Experiment results show this way to be effective.

H. THE SHARED OBJECT DATABASE

The Share Object Database proceeds to add the *"so"* library files from the database to the Sandbox environment based on the set of *"Shared library"* in the *".config"* file. The database of shared objects is collected from shared sources on the Internet and images of IoT device firmware (Router, IP camera, Smart TV, etc.) that manufacturers publish on the network. The C500-Reverse tool [49], published by our team, is used to extract library files from the firmware. Usually, the *"so"* files will be in the *"lib"* directory packaged in the firmware. For devices that do not find the firmware image online, it will be extracted directly from the device with the help of the C500-Extractor [49]. C500-Extractor hardware equipment shown in Fig. 25.

FIGURE 25. C500-Extractor device.

I. REPORT GENERATION

Finally, when the *"Sanbox Recomputation component"* returns *"false"* value, the system will automatically generate a summary report of ELF file behavior from *"Behavior data."* Illustrations of the content of the report can be found in Section IV-C.

IV. EVALUATION

A. DATASET

In order to experiment and evaluate V-Sandbox's analysis results, 9069 samples (6141 IoT botnet and 2928 IoT benign samples) are chosen to analyze. Dataset is collected from IoT-Pot [36], VirusShare,[20] and the Internet.[21] The architecture of samples in the dataset is shown in Table 6.

B. IMPLEMENT

The experiment is conducted on server with an Intel Xeon E5-2689 processor running at 2.6GHz, 32GB memory.

[20] "VirusShare.com." [Online]. Available: https://virusshare.com/. Accessed on: Jun. 05, 2019.

[21] Azmoodeh, "azmoodeh/IoTMalwareDetection," [Online]. Available: https://github.com/azmoodeh/IoTMalwareDetection. Accessed on: Mar. 12, 2020.

TABLE 6. Distribution of dataset.

Arch. CPU	Total	Bashlite	Mirai	Others malware	Benign	Static. sample	Dynamic. sample
ARM	2326	896	484	211	735	1468	853
MIPS	3225	786	479	138	1822	1385	1838
Intel 80386	2260	713	1004	360	183	1993	243
PowerPC	755	402	122	52	179	560	194
x86-64	503	384	47	63	9	474	29
All in dataset	9069	3181	2136	824	2928	5880	3157

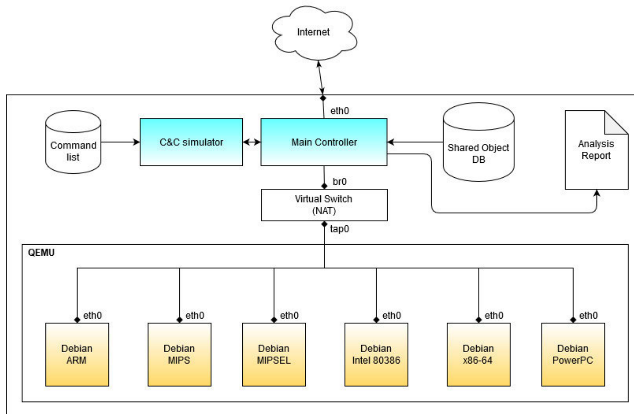

FIGURE 26. V-Sandbox deployment architecture.

V-Sandbox supports many different CPU architectures, basically including ARM, MIPS, MIPSEL, i386, x86-64, PowerPC (can be extended with many other architectures). A view of this implementation is shown in Fig. 26. All of these QEMU virtual machines are connected to a virtual switch for management, providing a simulated network environment as well as the ability to connect to the C&C server, monitor network traffic, and add missing libraries. The Main Controller is responsible for managing tasks including receiving and transferring ELF files to virtual hosts, determining the CPU architecture of the ELF to run the corresponding virtual host, activating the ELF file and monitor agents, providing Share Object (''so'' file) when ELF request, make a connection with dummy C&C server as needed and synthesize analysis report from agent monitor. The C&C simulator provides the ability to communicate between ELF and the C&C server by navigating the connection of the virtual switch combined with a set of command lists gathered from the Internet, paper [36], [42], [48], etc. The Shared Object database is used to provide ''.so'' files when required by the ELF. This Shared Object database collected from sources including the Internet, extracted from IoT device firmware (Router, IP camera, TV-box, etc.) provided by the manufacturer on the internet, OpenWrt images, etc.

C. EXPERIMENTAL RESULTS

The results of running Dataset on V-Sandbox are shown in Table 7. Experimental results show that with Dataset

TABLE 7. The results were successfully executed by V-Sandbox.

Arch. CPU	Static samples	Dynamic samples	Per. Static (%)	Per. Dynamic (%)	Total
ARM	915	283	62.33	33.18	1198
MIPS	936	655	67.58	35.64	1591
Intel 80386	1122	47	56.30	19.34	1169
PowerPC	441	7	78.75	3.61	448
x86-64	361	12	76.16	41.38	373
All in dataset	3775	1004	64.20	31.80	4779

```
{} strace2347.json ●
home > sandboxmips > Desktop > a7192c394957ba17878e3c1f57aca67b_1586004606 > {} strace2347.json > ...
5              "name": "clone",
6              "arguments": "child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
               child_tidptr=0x7712c068"
7          },
8          {
9              "timestamp": 1586004608,
10             "return": "1",
11             "name": "socket",
12             "arguments": "PF_INET, SOCK_RAW, IPPROTO_UDP"
13         },
14         {
15             "timestamp": 1586004608,
16             "return": "0",
17             "name": "setsockopt",
18             "arguments": "1, SOL_IP, IP_HDRINCL, [1], 4"
19         },
20         {
21             "timestamp": 1586004608,
22             "return": "540",
23             "name": "sendto",
24             "arguments": "1,
               \"E\\0\\2\\34\\277\\237\\0\\0@\\21\\275z\\300\\250ze\\300\\250\\0\\1\\212
               [\\266~\\2\\10h\\244t\\262\\260\\202\"..., 540, MSG_NOSIGNAL, {sa_family=AF_INET,
               sin_port=htons(46718), sin_addr=inet_addr(\"192.168.0.1\")}, 16"
25         },
26         {
27             "timestamp": 1586004608,
```

FIGURE 27. Log system-calls.

```
sandboxmips:a7192c394957ba17878e3c1f57aca67b_1586004606$ tcpdump -c 10 -r tcpdump.pcap
reading from file tcpdump.pcap, link-type EN10MB (Ethernet)
19:50:07.858808 IP 192.168.122.101.59487 > 192.168.122.1.12345: Flags [S], seq 3181798
659, win 14600, options [mss 1460,sackOK,TS val 4294920805 ecr 0,nop,wscale 2], length
 0
19:50:07.860616 IP 192.168.122.1.12345 > 192.168.122.101.59487: Flags [S.], seq 356555
1994, ack 3181798660, win 65160, options [mss 1460,sackOK,TS val 2998667613 ecr 429492
0805,nop,wscale 7], length 0
19:50:07.864555 IP 192.168.122.101.59487 > 192.168.122.1.12345: Flags [.], ack 1, win
3650, options [nop,nop,TS val 4294920807 ecr 2998667613], length 0
19:50:08.048936 IP 192.168.122.101.59487 > 192.168.122.1.12345: Flags [P.], seq 1:5, a
ck 1, win 3650, options [nop,nop,TS val 4294920853 ecr 2998667613], length 4
19:50:08.049312 IP 192.168.122.1.12345 > 192.168.122.101.59487: Flags [.], ack 5, win
510, options [nop,nop,TS val 2998667802 ecr 4294920853], length 0
19:50:08.051783 IP 192.168.122.101.59487 > 192.168.122.1.12345: Flags [P.], seq 5:6, a
ck 1, win 3650, options [nop,nop,TS val 4294920854 ecr 2998667802], length 1
19:50:08.051953 IP 192.168.122.1.12345 > 192.168.122.101.59487: Flags [.], ack 6, win
510, options [nop,nop,TS val 2998667805 ecr 4294920854], length 0
19:50:08.052096 IP 192.168.122.1.12345 > 192.168.122.101.59487: Flags [P.], seq 1:15,
ack 6, win 510, options [nop,nop,TS val 2998667805 ecr 4294920854], length 14
19:50:08.053260 IP 192.168.122.101.59487 > 192.168.122.1.12345: Flags [.], ack 15, win
 3650, options [nop,nop,TS val 4294920855 ecr 2998667805], length 0
19:50:08.230657 IP 192.168.122.101.35419 > 192.168.0.1.46718: UDP, length 512
sandboxmips:a7192c394957ba17878e3c1f57aca67b_1586004606$
```

FIGURE 28. Log network traffic.

TABLE 8. The results were executed by LiSa Sandbox with our dataset.

Arch. CPU	Static samples	Dynamic samples	Per. Static (%)	Per. Dynamic (%)	Total
ARM	807	0	54.97	0.00	807
MIPS	918	653	66.28	35.53	1571
Intel 80386	1375	20	68.99	8.23	1395
PowerPC	0	0	0.00	0.00	0
x86-64	293	12	61.81	41.38	305
All in dataset	3393	685	57.70	21.70	4078

containing 9069 samples, V-Sandbox ran and successfully analyzed 4779 samples with different CPU architectures. Notably, the proportion of static samples reached more than 60%, and dynamic samples were more than 30%. This result is an advantage when combining the C&C server and Share Object repositor in V-Sandbox. The results of comparison with other IoT Sandboxes (focusing on LiSa [41] and Cuckoo

TABLE 9. Compare the functions of IoT Sandboxes.

	Support Multi-CPU	Support the C&C Server	Support dynamically libraries	Collected data				Auto generation report
				Network	System calls	File activity	Host perfomance	
DroidScope [30]	N	N	N	N	Y	Y	N	N
AASandbox [31]	N	N	N	N	Y	N	N	N
Cuckoo [35]	Y	N	N	Y	NF	Y	N	Y
IoTBOX [36]	Y	N	N	Y	N	N	N	NS
Limon [38]	N	N	N	Y	Y	Y	N	Y
REMnux [39]	N	N	N	N	Y	N	N	Y
Detux [37]	N	N	N	Y	N	N	N	Y
Padawan [40]	Y	N	N	N	Y	Y	N	Y
LiSa [41]	Y	N	NF	Y	Y	Y	N	Y
V-Sandbox	Y	Y	Y	Y	Y	Y	Y	Y

*N: Not yet, Y: Yes, NF: Not Fully, NS: Not Sure (no open source)

```
ldd.txt
home > sandboxmips > Desktop > a7192c394957ba17878e3c1f57aca67b_1586004606 > ldd.txt
1  libc.so.0 => /lib/libc.so.0 (0x7780e000)
2  ld-uClibc.so.0 => /lib/ld-uClibc.so.0 (0x777f4000)
3
```

FIGURE 29. List of shared libraries required.

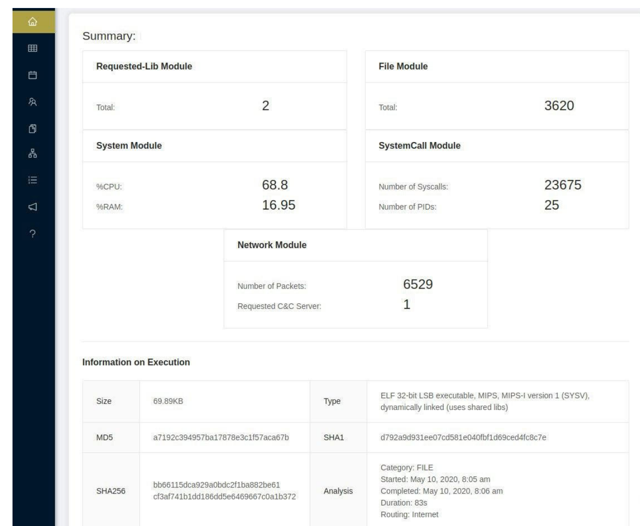

FIGURE 30. Fully reported by V-Sandbox.

[35] sandbox) are presented in Section V. The analysis results collected from the V-Sandbox for samples in the Dataset illustrated in Figs. 27 to 30.

V. DISCUSSION

A comparison of the results of an analysis of samples in our Dataset with LiSa Sandbox [41] is shown in Table 8. The results were executed by Lisa Sandbox with our dataset showing that this sandbox does not support PowerPC architecture. Also, the number of successfully executed statically and dynamically linked files in Lisa Sandbox is mostly less than our results.

TABLE 10. Selected samples to compare the effectiveness of sandboxes.

#	MD5	CPU architecture	Malware family [75]
1	89772d4f8d63117a5af7abd11ef66c5c	x86-64	Gafgyt
2	f70640f966d77234405df7d715f6e494	ARM	Gafgyt
3	571d93ccba8ee531627311fdb0b54c95	ARM	Benign
4	8b269f0eab1e09040c62ce78dff05c01	MIPS	Benign
5	1c7c1763888e0a0b67732db1e8e176ba	ARM	Gafgyt
6	0a982a3fb71dd70c248c107fcf33574f	PowerPC	Gafgyt
7	cf04a95a254a9aada0440281f82d6e9c	MIPSEL	Benign
8	2bb57df01bd06453775472df2098eff1	ARM	Tsunami
9	79b62cfd1975f09e24ce131181c1008a	Intel 80386	Mirai
10	4a832bd4fbb625cd095e9f56d695b047	MIPS	Mirai
11	9505af2cafb5b2bb8d10949543c5c416	MIPSEL	Bashlite
12	a7192c394957ba17878e3c1f57aca67b	MIPSEL	Mirai

Because IoTBOX [36] is not open source, it is not easy to install and test the effectiveness of this sandbox on our data set. Functionally, IoTBOX only focuses on network behavior. Therefore, this sandbox does not provide other behaviors like system calls and host performance. Also, the Cuckoo Sandbox [35] has been tested and installed by us, but it only provides support for X86 architectures and the installation of other CPU architectures is complicated. Therefore, to compare, we selected a few random samples to evaluate the effectiveness of the Cuckoo Sandbox in Table 10, 11. An overview of the functions of the IoT sandbox with V-Sandbox shown in Table 9. Functionally, most of the compared sandboxes do not support the C&C server simulation. This is why the IoT Botnet has not yet revealed adequate behavior. Only Lisa and our sandbox support dynamically libraries. For that reason, it increases our successful execution rate of input files. Our sandbox supports multi-architecture CPU (ARM, MIPS, PowerPC, etc.) like Lisa, Cuckoo, Padawan, and IoTBOX.

In Table 10, we describe 12 randomly selected samples to compare the effectiveness of the proposed sandbox with the two currently highly rated sandboxes, Cuckoo and LiSa sandbox. With the results of running selected samples in Table 11, our sandbox collected more information about system calls and network traffic than Lisa and Cuckoo sandbox.

TABLE 11. Results of running selected samples.

#	Support shared libraries			Total PID			Network behavior		
	V-Sandbox	LiSa	Cuckoo	V-Sandbox	LiSa	Cuckoo	V-Sandbox	LiSa	Cuckoo
1	static	static	static	3	3	3	C&C server connected, transfer command	requests IP C&C server	requests IP C&C server
2	static	static	static	170	3	3	C&C server connected, transfer command, DDoS attack traffic	requests IP C&C server	requests IP C&C server
3	3	0	0	1	1	1	Normal	Normal	Normal
4	1	0	0	1	1	2	Normal	Normal	Normal
5	static	static	static	9	3	3	C&C server connected, transfer command, DDoS attack traffic	requests IP C&C server	requests IP C&C server
6	static	static	static	6	1	6	C&C server connected, transfer command, DDoS attack traffic	No pcap	requests IP C&C server
7	2	0	0	1	1	1	Normal	Normal	Normal
8	static	static	static	2	1	4	Normal	Normal	Normal
9	static	static	static	1	1	1	requests IP C&C server	No pcap	requests IP C&C server
10	static	static	static	10	6	1	C&C server connected, transfer command, DDoS attack traffic	SYN scanning	SYN scanning
11	3	1	0	10	1	1	C&C server connected, transfer command, DDoS attack traffic	No pcap	requests IP C&C server
12	2	1	0	25	1	1	C&C server connected, transfer command, DDoS attack traffic	No pcap	requests IP C&C server

VI. CONCLUSION

This paper presents a V-Sandbox for dynamic analysis of IoT Botnet. This sandbox is capable of supporting multiple CPU architectures, the C&C servers connection, shared libraries so that the IoT Botnet exhibits more behavior than existing IoT sandboxes. The set of agents collects different behaviors (such as system calls and network traffic) of analytical samples that automatically integrated into virtual machines with the corresponding CPU architecture. The V-Sandbox has been tested through our dataset. Experimental results have shown that the proposed sandbox allows collecting behavioral data better than existing sandboxes. The source code for the V-Sandbox is shared on Github [76]. In the future, we will expand the capabilities of supporting more CPU architectures, adding data about C&C commands, and shared object libraries to the database for a more efficient sandbox.

REFERENCES

[1] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "Synthetic structure of industrial plastics, DDoS in the IoT: Mirai and Other Botnets," *Computer*, vol. 50, no. 7, pp. 80–84, Jul. 2017, doi: 10.1109/MC.2017.201.

[2] A. Spognardi, M. D. Donno, N. Dragoni, and A. Giaretta, "Analysis of DDoS-capable IoT malwares," in *Proc. Federated Conf. Comput. Sci. Inf. Syst.*, Sep. 2017, pp. 807–816.

[3] I. Andrea, C. Chrysostomou, and G. Hadjichristofi, "Internet of Things: Security vulnerabilities and challenges," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jul. 2015, pp. 180–187.

[4] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.

[5] K. Moskvitch, "Securing IoT: Your smart home and your connected enterprise," *Eng. Technol.*, vol. 12, no. 3, pp. 40–42, Apr. 2017.

[6] A. Sivanathan, D. Sherratt, H. H. Gharakheili, V. Sivaraman, and A. Vishwanath, "Low-cost flow-based security solutions for smart-home IoT devices," in *Proc. IEEE Int. Conf. Adv. Netw. Telecommun. Syst. (ANTS)*, Nov. 2016, pp. 1–6.

[7] D. Evans, "The Internet of Things: How the next evolution of the Internet is changing everything," CISCO, San Jose, CA, USA, White Paper 2011, vol. 1, 2011, pp. 1–11.

[8] K. Angrishi, "Turning Internet of Things(IoT) into Internet of vulnerabilities (IoV) : IoT botnets," 2017, *arXiv:1702.03681*. [Online]. Available: http://arxiv.org/abs/1702.03681

[9] Kaspersky Lab Report. (2017). *Honeypots and the Internet of Things*. Securelist-Kaspersky Lab's Cyberthreat Research and Reports. Accessed: May 11, 2018. [Online]. Available: https://securelist.com/honeypots-and-the-internet-of-things/78751/

[10] E. Bertino and N. Islam, "Botnets and Internet of Things security," *Computer*, vol. 50, no. 2, pp. 76–79, Feb. 2017, doi: 10.1109/MC.2017.62.

[11] C. Lévy-Bencheton, E. Darra, G. Tétu, G. Dufay, and M. Alattar, "Security and resilience of smart home environments good practices and recommendations," in *Proc. Eur. Union Agency Netw. Inf. Secur. (ENISA)*, Heraklion, Greece, Dec. 2015, pp. 1–77.

[12] G. Kambourakis, C. Kolias, and A. Stavrou, "The mirai botnet and the IoT zombie armies," in *Proc. IEEE Mil. Commun. Conf. (MILCOM)*, Oct. 2017, pp. 267–272.

[13] Kaspersky Lab Report. *IoT: A Malware Story*. Securelist - Kaspersky Lab's Cyberthreat Research and Reports. Accessed: Dec. 19, 2019. [Online]. Available: https://securelist.com/iot-a-malware-story/94451/

[14] C. Kruegel and Y. Shoshitaishvili, "Using static binary analysis to find vulnerabilities and backdoors in firmware," in *Proc. Black Hat USA*, Aug. 2015, pp. 8–15.

[15] P. Celeda, R. Krejci, J. Vykopal, and M. Drasar, "Embedded malware—An analysis of the chuck norris botnet," in *Proc. Eur. Conf. Comput. Netw. Defense*, Oct. 2010, pp. 3–10.

[16] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proc. EURECOM*, Sophia-Antipolis, Biot, France, 2014, pp. 95–110.

[17] A. Azmoodeh, A. Dehghantanha, and K.-K.-R. Choo, "Robust malware detection for Internet of (Battlefield) things devices using deep eigenspace learning," *IEEE Trans. Sustain. Comput.*, vol. 4, no. 1, pp. 88–95, Jan. 2019, doi: 10.1109/TSUSC.2018.2809665.

[18] H. HaddadPajouh, A. Dehghantanha, R. Khayami, and K.-K.-R. Choo, "A deep recurrent neural network based approach for Internet of Things malware threat hunting," *Future Gener. Comput. Syst.*, vol. 85, pp. 88–96, Aug. 2018.

[19] H.-T. Nguyen, Q.-D. Ngo, and V.-H. Le, "A novel graph-based approach for IoT Botnet detection," *Int. J. Inf. Secur.*, vol. 18, pp. 1–11, Oct. 2019.

[20] D. D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards automated dynamic analysis for linux-based embedded firmware," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–16, doi: 10.14722/ndss.2016.23415.

[21] H.-S. Ham, H.-H. Kim, M.-S. Kim, and M.-J. Choi, "Linear SVM-based Android malware detection for reliable IoT services," *J. Appl. Math.*, vol. 2014, pp. 1–10, Sep. 2014.

[22] A.-D. Schmidt, H. G. Schmidt, J. Clausen, K. A. Yuksel, O. Kiraz, A. Camtepe, and S. Albayrak, "Enhancing security of linux-based Android devices," in *Proc. 15th Int. Linux Kongress*, 2008, pp. 1–16.

[23] L. Liu, G. Yan, X. Zhang, and S. Chen, "Virusmeter: Preventing your cellphone from spies," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2009, pp. 244–264.

[24] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 559–570, Jun. 2013.

[25] H.-V. Le, Q.-D. Ngo, and V.-H. Le, "Iot Botnet detection using system call graphs and one-class CNN classification," *Int. J. Innov. Technol. Exploring Eng.*, vol. 8, no. 10, pp. 937–942, Aug. 2019.

[26] D. Davidson, B. Moench, S. Jha, and T. Ristenpart, "FIE on firmware, finding vulnerabilities in embedded systems using symbolic execution," in *Proc. 22nd USENIX Secur. Symp.*, 2013, pp. 463–487.

[27] T. Ronghua, "An integrated malware detection and classification system," Ph.D. dissertation, Deakin Univ., Melbourne, VIC, Australia, Aug. 2011.

[28] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. 23rd Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2007, pp. 421–430.

[29] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *Detection of Intrusions and Malware, and Vulnerability Assessment* (Lecture Notes in Computer Science), vol. 5137. Berlin, Germany: Springer, 2008, pp. 108–125.

[30] L. K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. 21st USENIX Secur. Symp. (USENIX Secur.)*, 2012, pp. 569–584.

[31] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An Android application sandbox system for suspicious software detection," in *Proc. 5th Int. Conf. Malicious Unwanted Softw.*, Oct. 2010, pp. 55–62.

[32] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "'Andromaly': A behavioral malware detection framework for Android devices," *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, Feb. 2012.

[33] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 1–29, Jun. 2014.

[34] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for android," in *Proc. 1st ACM Workshop Secur. Privacy Smartphones Mobile Devices SPSM*, 2011, pp. 15–26.

[35] D. Oktavianto and I. Muhardianto, *Cuckoo Malware Analysis*. Birmingham, U.K.: Packt Publishing Ltd, 2013.

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

[37] *Detux: The Multiplatform Linux Sandbox*. Accessed: Mar. 10, 2020. [Online]. Available: https://github.com/detuxsandbox/detux

[38] K. Monnappa, "Automating linux malware analysis using limon sandbox," in *Proc. Black Hat Eur.*, Aug. 2015, pp. 34–46.

[39] *REMnux: A Free Linux Toolkit for Reverse-Engineering and Analyzing Malware*. Accessed: Mar. 10, 2020. [Online]. Available: https://remnux.org/

[40] *Padawan Live*. Accessed: Mar. 10, 2020. [Online]. Available: https://padawan.s3.eurecom.fr/about

[41] D. Uhrıcek. *LiSa–Multiplatform Linux Sandbox for Analyzing IoT Malware*. Accessed: Mar. 11, 2020. [Online]. Available: http://excel.fit.vutbr.cz/submissions/2019/058/58.pdf

[42] B. Vignau, R. Khoury, and S. Halle, "10 years of IoT malware: A feature-based taxonomy," in *Proc. IEEE 19th Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, Jul. 2019, pp. 458–465.

[43] A. O. Prokofiev, Y. S. Smirnova, and V. A. Surov, "A method to detect Internet of Things botnets," in *Proc. IEEE Conf. Russian Young Researchers Electr. Electron. Eng. (EIConRus)*, Jan. 2018, pp. 105–108, doi: 10.1109/EIConRus.2018.8317041.

[44] J. Ceron, K. Steding-Jessen, C. Hoepers, L. Granville, and C. Margi, "Improving IoT Botnet investigation using an adaptive network layer," *Sensors*, vol. 19, no. 3, p. 727, Feb. 2019.

[45] S. S. Bhunia and M. Gurusamy, "Dynamic attack detection and mitigation in IoT using SDN," in *Proc. 27th Int. Telecommun. Netw. Appl. Conf. (ITNAC)*, Nov. 2017, pp. 1–6.

[46] G. Sagirlar, B. Carminati, and E. Ferrari, "AutoBotCatcher: Blockchain-based P2P botnet detection for the Internet of Things," in *Proc. IEEE 4th Int. Conf. Collaboration Internet Comput. (CIC)*, Oct. 2018, pp. 1–8.

[47] W. Li, J. Jin, and J.-H. Lee, "Analysis of botnet domain names for IoT cybersecurity," *IEEE Access*, vol. 7, pp. 94658–94665, 2019.

[48] C.-J. Wu, Y. Tie, K. Yoshioka, and T. Matsumoto, "IoT malware behavior analysis and classification using text mining algorithm," in *Proc. Comput. Secur. Symp.*, Oct. 2016, pp. 1–7.

[49] N.-P. Tran, N.-B. Nguyen, Q.-D. Ngo, and V.-H. Le, "Towards malware detection in routers with C500-toolkit," in *Proc. 5th Int. Conf. Inf. Commun. Technol. (ICoIC)*, May 2017, pp. 1–5, doi: 10.1109/ICoICT.2017.8074691.

[50] C. Dietz, R. L. Castro, J. Steinberger, C. Wilczak, M. Antzek, A. Sperotto, and A. Pras, "IoT-Botnet detection and isolation by access routers," in *Proc. 9th Int. Conf. Netw. Future (NOF)*, Nov. 2018, pp. 88–95.

[51] R. Doshi, N. Apthorpe, and N. Feamster, "Machine learning DDoS detection for consumer Internet of Things devices," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2018, pp. 29–35.

[52] I. Indre and C. Lemnaru, "Detection and prevention system against cyber attacks and botnet malware for information systems and Internet of Things," in *Proc. IEEE 12th Int. Conf. Intell. Comput. Commun. Process. (ICCP)*, Sep. 2016, pp. 175–182.

[53] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer, "Online passive-aggressive algorithms," *J. Mach. Learn. Res.*, vol. 7, pp. 551–585, Dec. 2006.

[54] I. Alrashdi, A. Alqazzaz, E. Aloufi, R. Alharthi, M. Zohdy, and H. Ming, "AD-IoT: Anomaly detection of IoT cyberattacks in smart city using machine learning," in *Proc. IEEE 9th Annu. Comput. Commun. Workshop Conf. (CCWC)*, Jan. 2019, pp. 0305–0310.

[55] N. Moustafa and J. Slay, "The evaluation of network anomaly detection systems: Statistical analysis of the UNSW-NB15 data set and the comparison with the KDD99 data set," *Inf. Secur. J., A Global Perspective*, vol. 25, nos. 1–3, pp. 18–31, Apr. 2016.

[56] D. Breitenbacher, I. Homoliak, Y. L. Aung, N. O. Tippenhauer, and Y. Elovici, "HADES-IoT: A practical host-based anomaly detection system for IoT devices," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Jul. 2019, pp. 479–484.

[57] A. Azmoodeh, A. Dehghantanha, M. Conti, and K.-K.-R. Choo, "Detecting crypto-ransomware in IoT networks based on energy consumption footprint," *J. Ambient Intell. Humanized Comput.*, vol. 9, no. 4, pp. 1141–1152, Aug. 2018.

[58] M. Ficco, "Detecting IoT malware by Markov chain behavioral models," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Jun. 2019, pp. 229–234.

[59] S. Sachdeva, R. Jolivot, and W. Choensawat, "Android malware classification based on mobile security framework," *IAENG Int. J. Comput. Sci.*, vol. 45, no. 4, pp. 514–522, 2018.

[60] T. Janarthanan and S. Zargari, "Feature selection in UNSW-NB15 and KDDCUP'99 datasets," in *Proc. IEEE 26th Int. Symp. Ind. Electron. (ISIE)*, Jun. 2017, pp. 1881–1886.

[61] Z. Liu, L. Zhang, Q. Ni, J. Chen, R. Wang, Y. Li, and Y. He, "An integrated architecture for IoT malware analysis and detection," in *Proc. Int. Conf. Internet Things Service*, 2018, pp. 127–137.

[62] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf., FREENIX Track*, vol. 41, 2005, p. 46.

[63] D. C. DuVarney, V. N. Venkatakrishnan, and S. Bhatkar, "SELF: A transparent security extension for ELF binaries," in *Proc. Workshop New Secur. Paradigms NSPW*, 2003, pp. 29–38.

[64] W. K. K. Oo, H. Koide, and K. Sakurai, "Analyzing the effect of moving target defense for a Web system," *Int. J. Netw. Comput.*, vol. 9, no. 2, pp. 188–200, 2019.

[65] C. Wiederseiner, V. Garousi, and M. Smith, "Tool support for automated traceability of Test/Code artifacts in embedded software systems," in *Proc. IEEE 10th Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Nov. 2011, pp. 1109–1117.

[66] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan, "DASE: Document-assisted symbolic execution for improving automated software testing," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, May 2015, pp. 620–631.

[67] *Debian Squeeze and Wheezy Images for QEMU*. Accessed: Mar. 28, 2019. [Online]. Available: https://people.debian.org/~aurel32/qemu/

[68] G. Verma, M. Imdad, S. Banarwal, H. Verma, and A. Sharma, "Development of cross-toolchain and linux device driver," in *System and Architecture*. Singapore: Springer, 2018, pp. 175–185.

[69] M. A. Aydın, A. H. Zaim, and K. G. Ceylan, "A hybrid intrusion detection system design for computer network security," *Comput. Electr. Eng.*, vol. 35, no. 3, pp. 517–526, May 2009.

[70] S. O. Amin, M. S. Siddiqui, C. S. Hong, and S. Lee, "RIDES: Robust intrusion detection system for IP-based ubiquitous sensor networks," *Sensors*, vol. 9, no. 5, pp. 3447–3468, May 2009, doi: 10.3390/s90503447.

[71] R. Agarwal and M. V. Joshi, "PNrule: A new framework for learning classifier models in data mining (a case-study in network intrusion detection)," in *Proc. SIAM Int. Conf. Data Mining*, Apr. 2001, pp. 1–17.

[72] A. A. Gendreau and M. Moorman, "Survey of intrusion detection systems towards an end to end secure Internet of Things," in *Proc. IEEE 4th Int. Conf. Future Internet Things Cloud (FiCloud)*, Aug. 2016, pp. 84–90, doi: 10.1109/FiCloud.2016.20.

[73] B. Krebs. *Source Code for IoT Botnet 'Mirai' Released*. Accessed: Sep. 30, 2016. [Online]. Available: https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/

[74] S. Edwards and I. Profetis, "Hajime: Analysis of a decentralized Internet worm for IoT devices," *Rapidity Netw.*, vol. 16, pp. 1–18, Oct. 2016.

[75] *VirusTotal—Free Online Virus, Malware and URL Scanner*. Accessed: Apr. 25, 2018. [Online]. Available: https://www.virustotal.com/en/

[76] H. V. Le. (2020). *V-Sandbox*. GitHub Repository. Accessed: Apr. 25, 2020. [Online]. Available: https://github.com/ndhpro/V-Sandbox

HAI-VIET LE received the master's degree in computer systems and networking from the Irkutsk National Research Technical University, Irkutsk, Russia, in 2014. He is currently pursuing the Ph.D. degree with the Graduate University of Science and Technology, Vietnam Academy of Science and Technology (VAST), Hanoi, Vietnam. He is also working as a Lecturer with the Department of Information Technology and Information Security, People's Security Academy, Hanoi. He has been doing research, application, and teaching since then in the fields of network security, artificial intelligence, machine learning, and more recently in malware analysis.

QUOC-DUNG NGO received the Ph.D. degree in informatics applied in automation and manufacturing from the Grenoble Institute of Technology, Grenoble, France. He is currently working as a Lecturer with the Department of Information Technology, Posts and Telecommunications Institute of Technology, Hanoi, Vietnam. He has actively participated in all the research activities. He has many books and has more than ten research articles to his credit. He has also guest-edited several edited books. His research interests include network security, malware analysis, artificial intelligence, and optimal energy.