International Journal of Information Security
https://doi.org/10.1007/s10207-019-00475-6

REGULAR CONTRIBUTION

# A novel graph-based approach for IoT botnet detection

Huy-Trung Nguyen[1,2] · Quoc-Dung Ngo[4] · Van-Hoang Le[3]

**Abstract**

The Internet of things (IoT) is the extension of Internet connectivity into physical devices and everyday objects. These IoT devices can communicate with others over the Internet and fully integrate into people's daily life. In recent years, IoT devices still suffer from basic security vulnerabilities making them vulnerable to a variety of threats and malware, especially IoT botnets. Unlike common malware on desktop personal computer and Android, heterogeneous processor architecture issue on IoT devices brings various challenges for researchers. Many studies take advantages of well-known dynamic or static analysis for detecting and classifying botnet on IoT devices. However, almost studies yet cannot address the multi-architecture issue and consume vast computing resources for analyzing. In this paper, we propose a lightweight method for detecting IoT botnet, which based on extracting high-level features from function–call graphs, called PSI-Graph, for each executable file. This feature shows the effectiveness when dealing with the multi-architecture problem while avoiding the complexity of control flow graph analysis that is used by most of the existing methods. The experimental results show that the proposed method achieves an accuracy of 98.7%, with the dataset of 11,200 ELF files consisting of 7199 IoT botnet samples and 4001 benign samples. Additionally, a comparative study with other existing methods demonstrates that our approach delivers better outcome. Lastly, we make the source code of this work available to Github.

**Keywords** Information security · IoT botnet · PSI-Graph · Static analysis · Deep learning

## 1 Introduction

The rapid growth of Industry 4.0, especially the development of Internet of things (IoT), has led to an unprecedented revolution in the cyberphysical systems and has brought rich utilities to users. The amount of IoT devices is increasing every day since they provide comfort in human life and perform work with better outcomes. For example, voice assistants, sensors and automation tools, smart home and smart transportation have offered various promising applications to improve life quality. According to [1], in 2019, the amount of IoT devices has quadrupled to 42.1 billion since 2012 and will continue the upward trend. However, in parallel with the development of IoT technology, there exist the security issues of information leaking, disruption to the operation or, in some scenarios, even loss of life [2] when anything can become a spy device to collect information and interact with users anytime, anywhere.

Therefore, the IoT devices are becoming more popular as a powerful tool of cybercriminals. According to Gartner analysts, 25% of cyberattacks will have involved IoT devices by 2020 [3]. Besides, according to Kaspersky Lab report, the number of IoT malware has than tripled those seen throughout 2017 [4]. The threats posed by the IoT malware are predicted to increase in the future. One of the most dangerous threats to IoT devices is botnets. Botnet is a network consisting of the infected IoT devices, referred to as bots, controlled by one or more attackers (botmasters) that remotely control the bots to perform malicious activities. This can be exemplified by late 2016 attack against the Domain Name System (DNS) provider Dyn, in which the combination of distributed denial of service (DDoS) and insecure IoT culminated with the explosion of the largest DDoS attack ever recorded. The massive botnet (network of infected devices) made by Linux.Mirai changed the world's perception of IoT

✉ Huy-Trung Nguyen
huytrung.nguyen.hvan@gmail.com

1 Institute of Information Technology, Vietnam Academy of Science and Technology, Hanoi, Vietnam

2 Graduate University of Science and Technology, Vietnam Academy of Science and Technology, Hanoi, Vietnam

3 People's Security Academy, Hanoi, Vietnam

4 Posts and Telecommunications Institute of Technology, Hanoi, Vietnam

Published online: 23 October 2019

security by infecting millions of connected devices such as web cameras, home routers and digital video recorders over the Internet. On October 21, 2016, this incident was known as the largest DDoS attack ever seen, reaching a magnitude of about 1.1 Terabits per second (Tbps) [5].

Due to the security incident of Mirai, researchers have also found that more botnets have emerged such as Torii, Slingshot in 2018, and Persirai, BrickerBot in 2017. Angrishi et al. [5] and De Donno et al. [6] have provided a detailed analysis and correlation between IoT botnet malware since 2008. This analysis clearly shows the evolution of these IoT botnets through out the year, as well as a rise in the number of the new malware families per year. Therefore, there is an urgent need to provide a protection mechanism on IoT devices to prevent or mitigate botnet infections when IoT becomes a major element of the future Internet and is incorporated in critical national infrastructures.

To deal with this problem, it is important to detect malicious files among benign ones. Several works are focusing on collecting files on IoT devices such as IoTPOT [7] to determine whether they are malicious or not. In recent years, researchers tend to take advantages of machine learning and deep learning method to apply in malware detection problems. This approach requires the dataset consisting of both malicious and benign samples. Benign files could be collected from the repository Web sites or directly extracted from the firmware of IoT devices [8]. Following surveys in IoT devices [5,9], 95% of the deconstructed firmware was based on Linux operating system. Therefore, in this paper, we focus on analyzing files on Executable and Linkable Format (ELF) which is the common standard file format in Linux.

Recently, IoT malware detection field has attracted much attention from researchers and organizations such as [7, 10–14]. These methods can be divided into two major approaches, namely static (code) and dynamic (behavioral) analysis. Dynamic approach [13,14] consists of monitoring executable files during its runtime and detecting abnormal behaviors. This approach is performed by collecting information such as API calls, network behavior, instruction traces, registry changes and memory writes [13,14] during the actual process. To perform this approach, the most crucial part is to build an adequate virtual emulator for malware to reveal behaviors. However, monitoring process is time-consuming, which requires resources and, in some cases, is not secure because malware could infect the real platform. In addition, the main limitation of this analysis method is only in single-path mode and cannot explore all of execution paths of the malware [15]. This is caused by some malware behaviors that only execute when certain trigger conditions [16] such as time, system events and network inputs are met (e.g., C&C host commands). Besides the limitations inherent from dynamic analysis approach, performing dynamic approach on executable files in IoT devices is a daunting process which needs a virtual sandbox supporting various processor architectures (e.g., MIPS, ARM, PowerPC, Sparc) and configuring the suitable environment for execution of IoT botnet malware. In this perspective, Pa et al. [7] found that there are a lot of telnet connection since 2014. Therefore, they developed IoTPOT based on telnet protocol to capture IoT malware. After that, they developed IoTBOX sandbox (with an OpenWRT-based build root), which allows them to analyze the captured IoT malware on 8 different CPU architectures, but the disadvantage of this method is only focusing on network behaviors. Afterward, in the same approach, Chang et al. [13] proposed an IoT sandbox which can support 9 kinds of CPU architectures including ARM (EL, HF), MIPS, MIPSEL, POWERPC, SPARC x86, x64 and sh4. In this work, they used both host behaviors and network behaviors in order to improve IoT botnet detection rate. However, building a simulation environment which allows the IoT botnet malware files to entirely exhibit behaviors is not simple, so the collected information is incomplete. In addition, dynamic analysis method is especially useful in analyzing obfuscated malware. Thus, this approach is considered as a part of the support to static approach to enhance botnet detection on IoT devices.

Static approach is expressed by analyzing and detecting malicious files without executing them. In static malware analysis, analysts reverse the executable file into assembly code to achieve a deeper understanding of the malware activities. Static analysis [10,17] relies on the extraction of a variety of characteristics such as printable string information (PSI), function length frequency (FLF), operation codes (opcodes), byte N-grams and header section from the executable file [18–20]. One of the major advantages of static analysis is the ability to observe the structure of malware, all possible execution paths in malware sample without considering the diversity of processor architecture. While static analysis has its advantages, it also has limitations. The key drawback of this approach is being unable to detect complex and polymorphic malware. However, the functionality of IoT malware is largely constrained by the limited resources of IoT devices (CPU, RAM, etc.), and IoT malware tends to be simple and forthright without using complicated customization, obfuscation or evasive techniques [10]. Therefore, static approach for detecting IoT malware is appropriate. Based on this approach, we propose a novel method to extract static feature, named PSI-Graph. The PSI-Graph shows the structure of the relationship among functions containing printable strings in the executable file which have not been mentioned in the previous studies. Our method could reduce computational complexity and costing time but still ensure high accuracy in IoT botnet detection. To perform the experiment, we use 11,200 ELF files (7199 IoT botnet of different variants and 4001 benign files). This paper is an extension of our preliminary work [21], and the whole materials of this work can be found at (https://github.com/dung4883/PSI_graphIoTBotnet.git/). The main contributions can be summarized as:

- Showing the advantages of static approaches for IoT malware botnet and describing the main challenges associated with this activity.
- Proposing a novel graph-based method for IoT botnet detection without the consideration of multi-architecture problem.
- Proving that the proposed method is effective and efficient for detecting IoT botnet malware files with an accuracy of 98.7%.

The rest of the paper is structured as follows. Related works will be described in Sect. 2. In Sect. 3, we will introduce the methodology and approach of this paper, including the characteristics of IoT botnet, PSI-Graph generation, preprocessing PSI-Graph and neural network. Section 4 will discuss the conducted experiments and used dataset. Finally, conclusions and future works will be presented in Sect. 5.

## 2 Related works

Unlike desktop PCs and Android devices, IoT devices commonly adopt various processor architecture. Meanwhile, static analysis is often used to deal with the multi-architecture problems. Static approach relies on the extraction of known characteristics or patterns from ELF files such as operation code (opcode) sequences, printable string information (PSI), grayscale image and control flow graph (CFG). Some related studies based on this approach are as follows.

Alhanahnah et al. [10] combined different static features such as PSI, CFG and statistics of file structure to generate distinguishable signatures to classify multi-architecture IoT malware. The signature generation process is composed of three main steps which are: coarse-grained clustering, fine-grained clustering and cluster merging. In coarse-grained clustering phase, the statistical features from the assembly codes including total number of functions, total number of instructions, number of redirect instructions, etc. are used to improve the fine-grained clustering phase. After that, the last step aims at generating N-gram string vectors to compute the signature for each IoT malware cluster afterward, even with 95.5% of accuracy based on a database of 5150 IoT malware samples. At the second phase concerning the fine-grained clustering phase, Bindiff is used for performing the pairwise structural similarity between every malware CFG by generating an $N*N$ matrix ($N$ is the number of malware sample). However, there are some potential limitations to this work. Firstly, this study is based on CFG graphs analysis with high complexity and time-consuming. Secondly, this method can deal with ELF files containing encrypted strings, but it depends greatly on the value of these strings and cannot describe the complex semantic information (such as data dependency in IoT botnet).

Haddadpajouh et al. [22] proposed an approach that uses deep recurrent neural network (LSTM structure) to detect IoT botnet based on the operation code (opcode) sequences such as *inc, add, mov, dec*. The process of generating opcode sequences consists of three steps: unpacking ELF executable files with Debian installer bundle; extracting opcodes with object dump tool; and obtaining sequential opcode orders. Afterward, three different long short-term memory (LSTM) configurations are used to detect IoT malware. The experimental results achieved a detection accuracy of 98.18% for IoT malware based on the dataset of 551 ARM-based IoT excutable samples. In the same work, Azmoodhe et al. [19] presented a deep learning-based method to detect Internet of battlefield things (IoBT) malware based on device's opcode graph. Firstly, they utilized Objdump to extract opcode sequence and chose 2-g opcode as a feature by using class-wise information gain for the classification task. Then, they constructed an opcodes' graph from selected opcodes based on control flow graph for each sample. Next, they embedded opcodes' graphs into a vector space and applied a deep eigenspace learning approach to classify malware and benign samples. They evaluated the proposed method with a dataset comprising 128 malware and 1078 benign samples (all ARM-based IoT executable files) and achieved accuracy, precision and recall rate of 99.68%, 98.59% and 98.3%, respectively. Unfortunately, these works are not robust since their dataset had a limitation in terms of the number of samples, and the multi-architecture problem was not considered in this study because the experimental result only takes the evaluation for the ARM-based IoT dataset. Besides, opcode sequences could be disrupted by simple code variations resulted from different compilation options for multi-architecture on IoT.

Nataraj et al. [23] also proposed a malware classification method using texture features from visualized malware binary file as grayscale images in the range [0,255]. In this perspective, Su et al. [12] proposed a lightweight method for detecting IoT botnet based on image grayscale recognition. In particular, the proposed approach reformatted a binary as an 8-bit sequence and then converted them into a grayscale image with the size of $64 \times 64$ pixels which has one channel and the value of a one-channel pixel in the range [0,255]. Then, these images serve as inputs to a convolutional neural network and one-class SVM classifier which determines whether a binary file is malicious or not. To evaluate the performance of their study, the authors used IoTPOT malware dataset [7]. The experiment based on dataset comprising 500 IoT malware samples and equivalent the number of benign samples demonstrates that their method obtains a good performance in malware detection with 94% accuracy for two-class classification and 81% accuracy for three-class classification (i.e., benign, Mirai or Gafgyt). However, in these studies, due to the difference in file sizes, the rescaling method is necessary to standardize the image sizes. Additionally, malware samples have distinct byte patterns in the padded areas and unused data sections, though the padded areas and unused data sections can be easily modified by malware writers without affecting behaviors of malware [24]. Therefore, it is a significant change in grayscale images, which greatly affects the efficiency of detecting malware.

Overall, existing IoT botnet detection approaches employ a wide range of tools and exploit different botnet characteristics to achieve their goal. However, there are still a series of limitations that are more or less common among most existing works. These limitations are: (1) consuming a great amount of computational resources to accurately analyze the control flow graph of IoT binaries in studies [10,19,22]; (2) cannot address the multi-architecture problem due to relying on low-level features (opcode) in studies [19,22]; (3) requiring reformating method which could change natural structure of a sample to standardize input in the study [12]. Therefore, in this paper, we aim to detect IoT botnet using a high-level feature called PSI-Graph which represent life cycle of IoT botnet. The main advantages of our approach are:

1. Effectively addressing the challenges of data encryption in the case of IoT botnet detection.
2. Tackling the multi-architecture problem on IoT devices.
3. Reducing computational time as well as avoiding the complexity of analyzing control flow graphs.

## 3 The proposed method

In this section, we introduce our approach and model which are based on PSI-Graph.

### 3.1 The IoT botnet life cycle

The existence of IoT botnet has been a known fact since 2008 [5,6] with Linux.Hydra botnet. After that, IoT botnets have been evolving in sophistication and impact with various variants. However, their behaviors often have similar patterns as well as life cycle. According to recent observations and preliminary analysis [5,7,25], although IoT botnets have a number of other functions, they share resemble life cycle with existing botnet on computer platforms. In this paper, we summarize the common mode of operation for most IoT botnets as follows:

1. Scanning for the appropriate target: IoT botnets scan other IoT devices containing security vulnerabilities which can be infected. This could be done randomly by scanning IP addresses with commonly targeted service ports, such as Telnet (port 23 or 2323) and SSH (port 22).
2. Gaining access of other vulnerable devices: The bot engages in brute force attack to discover the default credentials of weakly configured IoT devices with possible hard-coded username–password pairs (e.g., root/root, admin/123, etc.).
3. Infecting IoT devices: After logging into a device, the bot infects a loader which is used to download and execute the corresponding binary version of the botnet, typically via FTP, HTTP or Trivial File Transfer Protocol.
4. Communicating with C&C server over IP address or URL: IoT botnets try to connect and forward various device characteristics, such as IP address and hardware architecture to the report server through a different port.
5. Waiting commands: The bot enters in the main foreground execution loop in which it basically establishes the connection with the C&C server and keeps it alive waiting for further commands. If an attack command is received, the corresponding routine is invoked and the attack is performed.

Compared to other similar botnets [26], almost all stages of the IoT botnet's life cycle leave footprints which can be recognized and can be used as the features to detect IoT botnet in our approach. These signatures include IP addresses, username–password patterns, attack commands and so on.

### 3.2 Overview of the proposed method

According to the IoT botnet life cycle mentioned above, an IoT botnet is characterized by five steps of its life cycle. It is important to note that each stage in an IoT botnet's life cycle usually relates to the information represented as a string (i.e., IP address of other vulnerable devices, username–password patterns, malicious code injection commands, IP address of C&C server and command to perform malicious behaviors), which is called printable string information (PSI) [10]. In fact, although there are some encryption techniques used to confuse these pieces of information as in the case of Linux.Mirai botnet [6], a botnet always has to go through a cycle of phases so that compromised devices are useful as botmaster may desire [27]. Based on this idea, we make a assumption that there should exist a static feature which can reflect the "life cycle behavior" of an IoT botnet and can apply data mining techniques for IoT botnet detection.

Several studies have shown that it is possible to detect botnet symptoms by observing the stages of a bot life cycle. Khoshhalpour et al. proposed a technique for detecting botnet behaviors based on analyzing network activities of an individual bot-infected host, which can unfold three different phases of a bot [28]. With this idea, Prokofiev et al. [29]

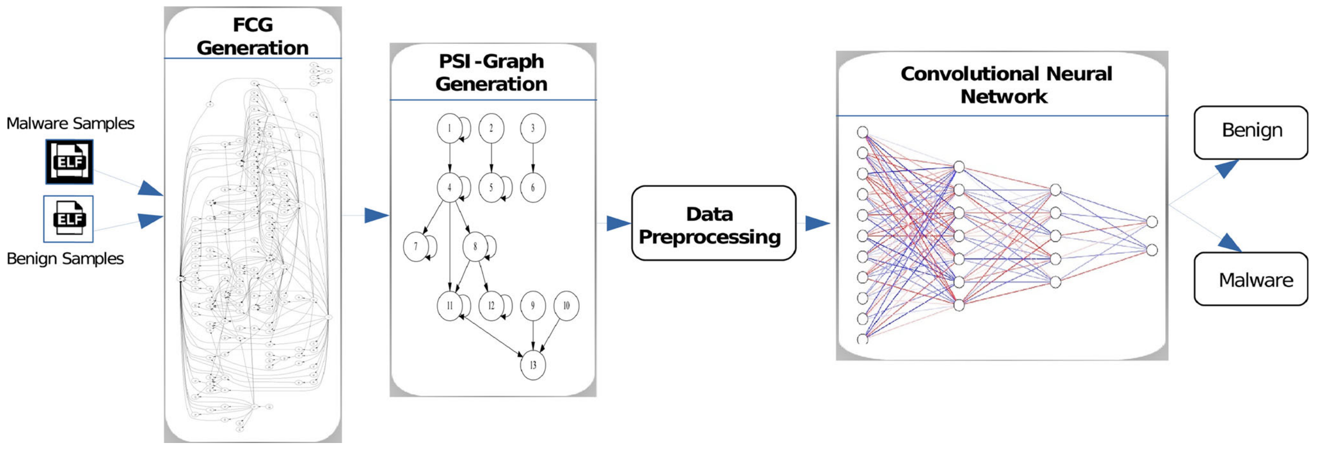

**Fig. 1** Overview of the proposed method

detect IoT botnets by monitoring the propagation stage of a bot following several parameters, such as destination ports, open source ports and number of requests. These studies do not approach the advantages of static analysis; therefore, they could confront similar limitations to other dynamic analysis methods such as only describing one run of a program; they nevertheless show that it is possible to inspect and detect several parts in a botnet's life cycle.

For our method, we consider PSIs as the characteristic of a stage in a bot's life cycle and construct a specific graph called PSI-Graph. PSI-Graph intends to represent IoT botnet's life cycle through the relationships between these PSIs which reflects the operations in any steps of a bot. Before describing the technique proposed, we first define a PSI-Graph as follows:

**Definition 1** PSI-Graph is a directed graph defined as $G_{PSI} = (V, E)$ where:

- $V$ is the vertices set composed of functions which are in function–call graph and contain PSIs,
- $E$ is the set of directed edges $\{(V_i,V_j), (V_k,V_h), \ldots\}$ reflecting the caller–callee relationship between two functions.

Our method is composed of four steps: function–call generation, constructing PSI-Graph, preprocessing and classifying as shown in Fig. 1. First, binary programs are unpacked and disassembled with UPX and IDA Pro. We then build function–call graphs and PSI-Graphs from disassembled codes and the caller–callee relationships. After that, the PSI-Graphs are processed and converted into numeric vector data by a graph embedding technique called graph2vec [30]. Finally, a convolutional neural network is used to classify these samples into two classes which are botnet or not.

## 3.3 Function–call graph generation

A function–call graph (FCG) is a control flow graph, which represents calling relationships between subroutines in a program. A function–call graph is defined as a directed graph $G = (V, E)$, which is composed of a vertex set $V$ and an edge set $E$, where the vertex in the graph is corresponding to the function included in the program, and the edge means the caller–callee relationship between two functions [31]. There are two types of functions which are used in a program, named local functions and external functions. Local functions are executable and written by a writer to perform a specific task; while external functions are system or library functions written by the author, local functions have various names in different programs even though they have similar function; at the same time, the names of external functions are consistent throughout all programs. Moreover, contrary to local functions, external functions never invoke any local function because they are all from the systems or libraries.

Given an executable file, we first use a packer identifier named DiE—Detect It Easy [32]—to check whether a file is packed and what kind of packing tool is exploited. Then, if this is identified, several unpackers are used to unpack the file accordingly such as UPX [33]. The executable files which cannot be unpacked by these tools are not considered in our approach. After unpacking step, we utilize IDA Pro [34]—a well-known disassembler and debugger for binaries of Linux, MacOS and Windows to disassemble the sample. At this step, we use IDAPython to create a plug-in which extracts the assembly code of binary and a set of identified functions. Once this relevant information has been extracted, the function–call graph is constructed.

As in Shang et al. [35], we use breadth first search to build the function–call graph. From the entry point functions, the caller–callee relationships between two functions are identi-

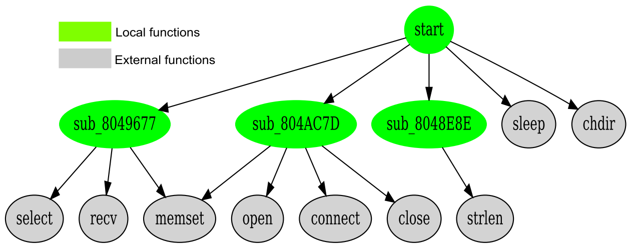

**Fig. 2** A part of the function–call graph of a Linux.Mirai sample

fied and added to the graph. A graph which consists of local functions and external functions is shown in Fig. 2.

## 3.4 Constructing the PSI-Graph

A function–call graph is a useful data representation for control flow of a program. This type of graph intends to represent every possible run of a program. Call graphs are usually complex with a large number of nodes and edges, so that it takes longer to compute and more memory to store. Although every call relationship of a program is represented in the graph, possibly some call relationships would never occur in actual runs of the program. In order to minimize the complexity of computing, in this paper, we construct PSI-Graph by selecting the functions and relationships which are close to the operations of IoT botnet's steps mentioned in Sect. 3.1.

Before constructing the PSI-Graph, we first extract all PSIs existing inside a binary by a plug-in of IDA Pro. These PSIs can be encrypted and often contain important semantic information which can reveal the attacker's intent. For instance the, strings "/dev/watchdog; /dev/misc/watchdog" always exist in Linux.Mirai, implicating that botnet tries to prevent the devices' rebooting process [36]. In some cases, the extracted strings are obfuscated or encrypted as shown in Fig. 3 by using the same obfuscation mechanisms. Finally, we use a simple algorithm to generate the PSI-Graph for each executable sample.

After obtaining the PSIs of executable samples, we construct the PSI-Graph that represents each executable sample. The process of PSI-Graph generation is shown in Algorithm 1.

Considering the trade-off between classification accuracy and computational complexity, we choose functions which contain PSIs with the length of at least three characters. Figure 4 shows an example of the function–call graph and PSI-Graph extracted from the decompiled code of a Linux.Bashlite sample. It is clear from the figure that the PSI-Graph is much simpler than the function–call graph. On average, a PSI-Graph contains only about 16 vertices and 60 edges, compared to 156 vertices and 360 edges of a function–call graph.

| Address | Length | Type | String |
|---|---|---|---|
| .rodata:8000E172 | 00000006 | C | eGAIM\v |
| .rodata:8000E179 | 00000007 | C | aJPMOG\r |
| .rodata:8000E18E | 00000007 | C | qCDCPK\r |
| .rodata:8000E19D | 00000008 | C | oMXKNNC\r |
| .rodata:8000E1A9 | 00000008 | C | \nuKLFMUQ |
| .rodata:8000E1C1 | 0000000C | C | cRRNGuG@iKV\r |
| .rodata:8000E1D4 | 00000006 | C | \nijvon |
| .rodata:8000E1E1 | 00000006 | C | eGAIM\v |
| .rodata:8000E1E8 | 00000007 | C | aJPMOG\r |
| .rodata:8000E1FD | 00000007 | C | qCDCPK\r |
| .rodata:8000E20C | 00000008 | C | oMXKNNC\r |
| .rodata:8000E293 | 00000005 | C | kLVGN |
| .rodata:8000E2AB | 0000000C | C | cRRNGuG@iKV\r |
| … | … | … | … |

**Fig. 3** Encrypted/obfuscated strings in a Linux.Mirai sample

**Algorithm 1** PSI-Graph Generation (FCG)

```
1:  $V = [\,]$, $E = [\,]$
2:  For each vertice $v_i$ in FCG do:
3:      If exist psi in $v_i$ and $v_i \notin V$ do:
4:          $V = V \cup v_i$
5:      End If
6:      For each edge $e_j(v_i,v_k)$ do:
7:          If exist psi in $v_k$ and $v_k \notin V$ and $e_j(v_i,v_k) \notin E$ do:
8:              $V = V \cup v_k$
9:              $E = E \cup e_j(v_i,v_k)$
10:         End If
11:     End for
12: End for
13: Return $V$, $E$
```

## 3.5 Preprocessing PSI-Graph

This section describes the employed method for transforming PSI-Graphs into the numeric format vectors which are used as an input for the classifier.

As mentioned in Sect. 3.1, IoT botnet always has to go through a cycle of sequential operation. To decide whether a sample is botnet or not, we expect to find the malicious cycle which exists inside the executable files. In PSI-Graph, each chain of IoT botnet's actions usually corresponds to a subgraph and all behaviors of a bot are represented via these subgraphs. Therefore, considering all possible subgraphs to represent the entire graph is significantly important in our method. In this paper, we use graph2vec [30], an unsupervised representation learning technique, to transform a graph into the numeric vector format.

Graph2vec is based on the idea of the doc2vec [37] approach which uses the skip-gram network. Graph2vec learns the representations of graphs by treating an entire graph as a document and the subgraphs as words which com-

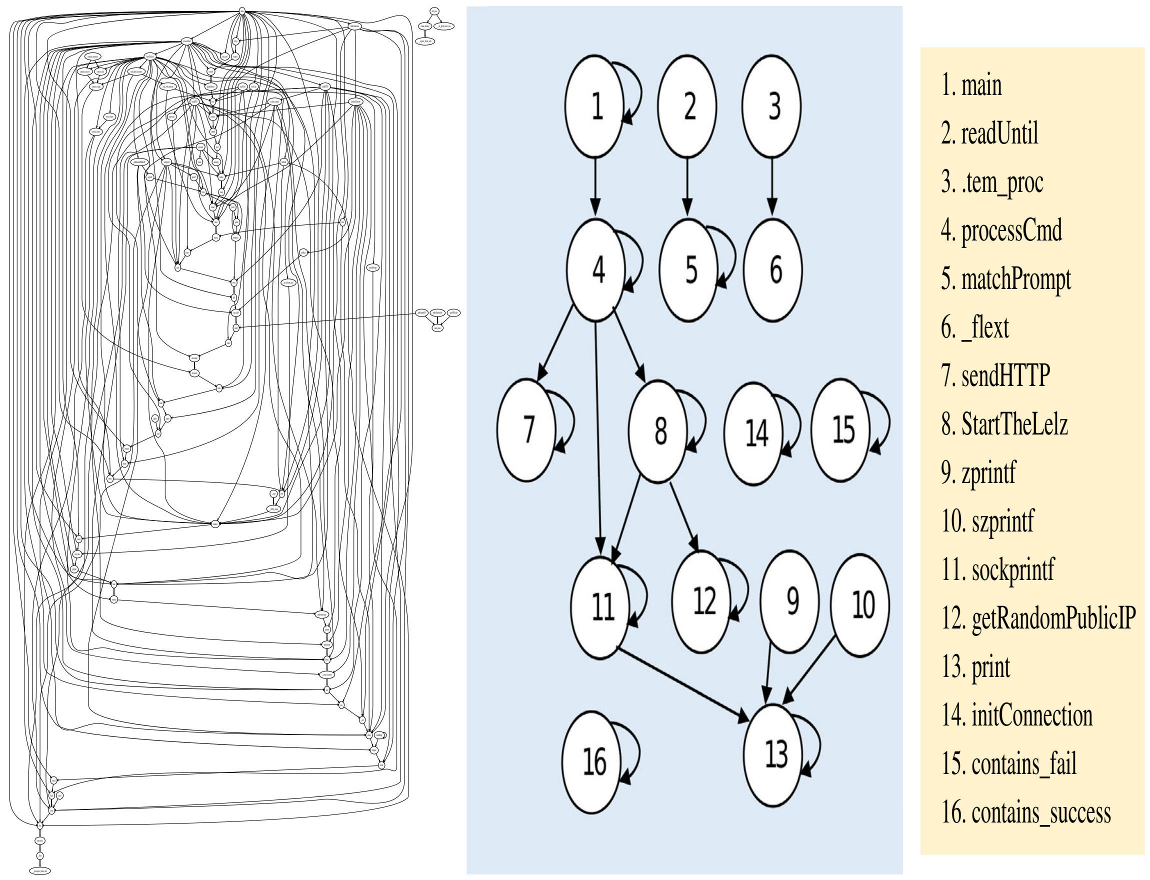

**Fig. 4** Function–call graph (left) and PSI-Graph (right) of a Linux.Bashlite sample

pose the document. Graph2vec approaches consist of three steps:

– Sampling and relabeling all subgraphs from the graph. Subgraph is a set of nodes that appear around the selected node. Nodes in subgraph are not further than the selected number of edges away.
– Training the skip-gram model. Graphs are similar to documents. Thus, documents are set of words and graphs are set of subgraphs. In this phase, the skip-gram model is trained. It is trained to maximize the probability of predicting subgraph that exists in the graph on the input. The input graph is provided as a one-hot vector.
– Computing embeddings to provide a graph ID as a one-hot vector at the input. Embedding is the result of the hidden layer.

Since subgraphs are predicted by the task, graphs with similar subgraphs and similar structure have similar embedding. The result of this step is the set of one-hot vectors of arbitrary length representing the set of graphs. In this paper, we represent graphs as numeric vectors of length 1024 for the classification task.

## 3.6 Classifier based on CNNs

The data obtained after the preprocessing step are used for deciding whether a sample is malicious or not by using a deep neural network classifier.

After preprocessing PSI-Graphs, we obtain vector data and choose an algorithm capable of analyzing vector data. Making use of the vector data means that the rate and direction of change in features, as well as the raw values themselves, are all input to the model. Convolutional neural network (CNN) and recurrent neural network (RNN) are both able to effectively analyze numeric vector data. In comparison, RNNs usually are effective at predicting the values of the next state, while CNNs have been successful in various classification tasks [14]. Besides, CNNs tend to be much faster than RNNs with about 5 times faster, due to the fact that convolutions are a central part of computer graphics and implement on a hardware level on GPUs [38]. Therefore, in this paper, we use CNN as the classifier for detecting IoT botnet based on learning the data from PSI-Graph preprocessing step.

In order to build the convolutional neural network, we apply the network which is originally used for sentence clas-

**Table 1** Description of the experimental dataset

| Family name | Variants | Samples |
|---|---|---|
| Mirai | 7 | 1765 |
| Bashlite | 5 | 3720 |
| Other IoT botnets | 9 | 680 |
| Benign | – | 3845 |
| Total | | 10,010 |

sification proposed by Kim [39]. The first layer of the neural network is an input layer. The next layer performs convolutions using multiple filter sizes. The output of this layer is forwarded to a nonlinear function, called ReLU activation which is defined as $f(x) = \max(0, x)$ since ReLUs are much computationally simpler, compared to the sigmoid activation, which requires computing an exponent. Next, the max-pooling layer is used for down-sampling the data produced from convolution layer, so that processing data can be reduced and computational resources can handle the scale of the data. Finally, the fully connected layer performs classification on the output generated from convolution layer and pooling layer.

## 4 Experiments and results

### 4.1 Description of experimental datasets

To validate our proposed method, we used the IoT botnet dataset collected from two different sources: The first one was from IoTPOT team [7] within a 1-year collected period between October 2016 to October 2017, and the second one was from VirusShare [40] (an online malware repository for research purpose). Because our method aims to classify samples into 2 classes which are IoT botnet and benign, we generated a set of benign samples by extracting the firmware of IoT SOHO devices with binwalk tool [41]. In this study, these firmware files are downloaded from the Web site of brands such as TP-Link, D-Link and Asus. In order to cover the diversity of benign dataset, we also collected firmware binaries from OpenWRT [42] for other brands such as Buffalo, Netgear or Tenda. Benign files were filtered by Virustotal [44] one more time to discard samples which received true positive, which is being detected as malicious by any anti-virus vendors. The dataset in this paper contains 11,200 samples, among which 7199 are botnet samples and 4001 are benign samples. After removing non-binary files and non-readable files by IDA Pro, the dataset consists of 6165 botnet samples and 3845 benign samples. The detail of dataset is shown in Table 1.

In addition, we check the CPU architecture on which each sample runs with the ***file*** command of Linux. Figure 5 shows the diverse processor architectures of malware sample in our

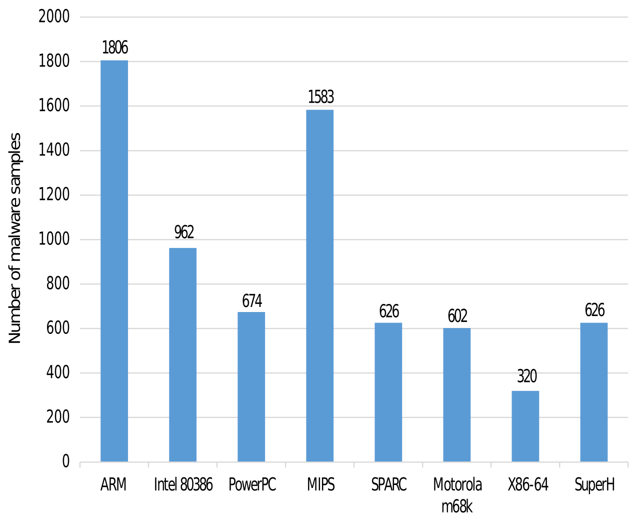

**Fig. 5** IoT malware distribution based on processor types

dataset, where MIPS, ARM and Intel 80386 are three most popular architectures for IoT malware.

For our experiments, the dataset is divided into two subsets: training and testing. The training subset contains equally 2690 samples for both botnet class and benign class. The testing subset contains the rest of dataset which consists of 4630 samples. The experiment is conducted with Python and Pytorch framework [43] on Ubuntu OS 16.04, with an Intel Core i5-8500 processor running at 3.00GHz , 12GB NVIDIA GeForce GTX1080Ti graphic card and 32 GB memory. The learning rate is set at 0.001 for training, which consists of 100 epoch runs.

### 4.2 PSI-Graph feature

As explained in Sect. 3, we represent the control flow of a program as the PSI-Graph and then employ a preprocessing technique to generate the numeric vectors used in the classification. PSI-Graphs are constructed from function–call graphs which usually are complex and require large computing resources. Table 2 shows the comparison between the size of PSI-Graphs and function–call graphs.

As it can be seen, the size of PSI-Graphs is much smaller than the size of function–call graphs, so that using PSI-Graphs as the feature to detect malware could be faster than using function–call graphs. To understand better the PSI-Graph feature, we investigate the distribution of the number of vertices and edges in PSI-Graph of IoT botnets, compared with benign samples in our dataset. As shown in Table 3, the majority of the number of vertices is in the range [1,300] for both botnet and benign files. Although there is a slight difference in the distribution, this distinction is not clear enough to set a threshold for distinguishing botnet samples and benign samples. Figure 6 shows the distribution of the number of vertices in the PSI-Graph of IoT botnet and benign samples.

**Table 2** Comparison between the size of PSI-Graphs and FCGs

| Family name | Avg. vertices in PSI-Graph | Avg. edges in PSI-Graph | Avg. vertices in FCGs | Avg. edges in FCGs |
|---|---|---|---|---|
| Malware | 147.1 | 1110.5 | 254.5 | 3075.5 |
| Benign | 167.8 | 1693.9 | 530.9 | 2962.2 |

**Table 3** Detailed number of vertices and edges for each class

| Family name | Mirai | Bashlite | Other botnets | Benign |
|---|---|---|---|---|
| Avg. vertices | 39.1 | 172.5 | 288.8 | 167.8 |
| Avg. edges | 233.4 | 624.3 | 1110.5 | 1693.9 |
| Min. vertices | 3 | 1 | 2 | 1 |
| Max. vertices | 1813 | 637 | 2313 | 11,901 |
| Min. edges | 15 | 7 | 0 | 0 |
| Max. edges | 18,093 | 2816 | 9882 | 65,817 |

## 4.3 Evaluation metrics

To evaluate the effectiveness of the proposed method, we compare with two related works using IoTPOT dataset, in which we use these measures such as precision, recall, F1-score and accuracy. In these, TP, TN, FP and FN are described as follows:

- *True positive (TP)* indicates that a malware is correctly identified as a malicious application.
- *True negative (TN)* indicates that a benign is detected as a non-malicious application correctly

**Table 4** Detection results by using PSI-Graph and function–call graph

| | Accuracy (%) | FNR (%) | FPR (%) | Time (m) |
|---|---|---|---|---|
| PSI-Graphs | 98.7 | 1.83 | 0.78 | 88 |
| FCGs | 95.3 | 5.81 | 4.13 | 545 |

- *False positive (FP)* indicates that a benign is falsely detected as a malicious application.
- *False negative (FN)* indicates that a malware is not detected and labeled as a non-malicious application.

Based on the above criteria, the following metrics will then be used to quantify the effectiveness of a given system. These evaluation metrics are adopted frequently in the research community to provide comprehensive assessments of imbalanced learning problems. These metric are defined as follows:

- Accuracy: Acc = (TP + TN)/(TP + FP + FN + TN)
- False positive rate: FPR = FP/(FP + TN)
- False negative rate: FNR = FN/(TP + FN)

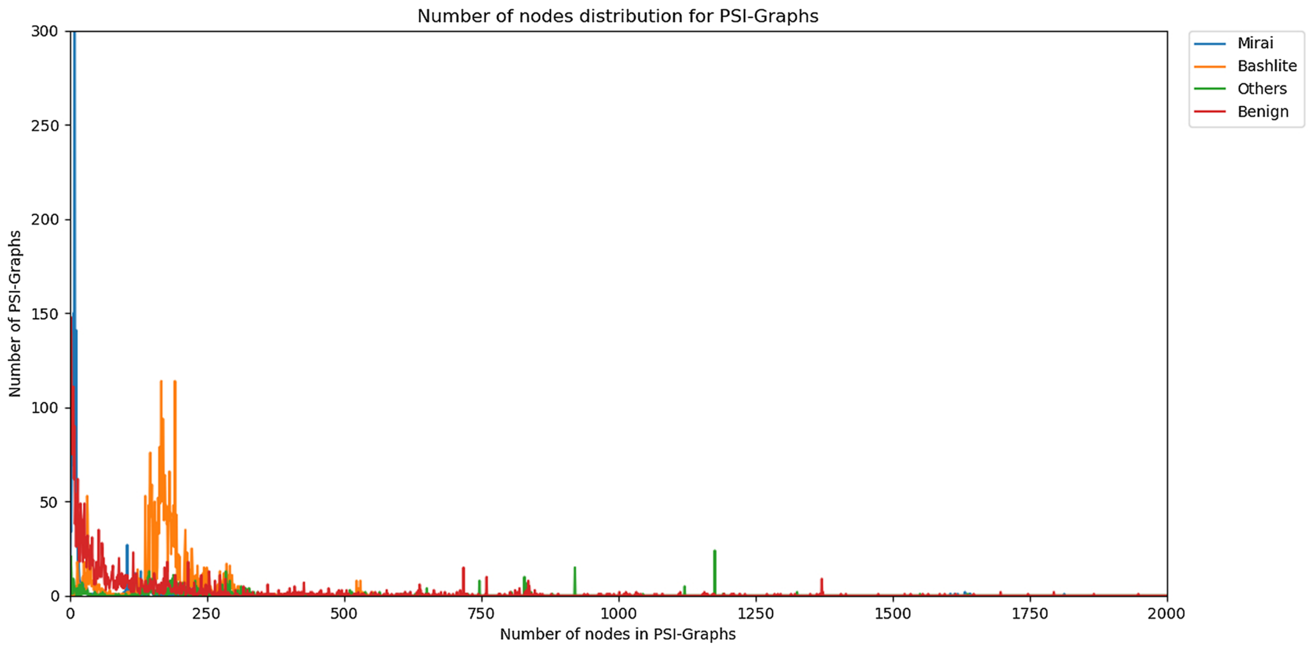

**Fig. 6** Number of vertices and edges between classes

**Table 5** Comparison between IoT botnet detection methods

| | Algorithm (s) | Samples | Accuracy (%) |
|---|---|---|---|
| Alhanahnah et al.[10] | N-gram | 4130 (include 4000 malware IoTPOT) | 95.5 |
| Su et al. [12] | Deep neural network (CNN) | 865 (include 500 malware IoTPOT) | 94 |
| HaddadPajouh et al. [22] | Recurrent neural network | 551 (include 281 malware ARM-based) | 98.18 |
| Azmoodeh et al. [19] | Deep eigenspace learning | 1206 (include 128 malware ARM-based) | 99.68 |
| Proposed method | Deep neural network (CNN) | 6943 (include 3098 malware IoTPOT) | 98.7 |
| | | 10,010 (6165 malware IoTPOT and VirusShare) | 97.8 |

## 4.4 Results

As shown in Table 4, from the results, it is evident that the proposed approach using PSI-Graphs feature performs better in comparison with using function–call graphs. The results show that our work achieves better accuracy score than using FCGs with 1.70 percent higher as well as consumes less time than using FCGs with 457 min less. We also observe the false negative rate in our proposed method (1.83%) compared with using FCGs (5.81%). It should be noted that, in malware detection, the smaller the false negative is, the less like the classifier wrongly labels a malicious sample as benign.

As shown in Table 5, the measured accuracy scores are noticeably different between studies. This is understandable due to the diversity of independent datasets as well as model specifications between related work. It is also not possible for us to recreate the same dataset with identical train/test split for direct comparison. We therefore find similar between datasets and algorithms that other studies use. Particularly, Alhanahnah et. al. [10] and Jiawei Su et al. [12] also collected malicious samples from IoTPOT [7]. The results show that our work achieves better accuracy score with 3.20% and 4.00% higher, respectively. Especially in Mohannad Alhanahnah's approach, the experimental designs and results indicated that this work only achieved 85.2% accuracy in detecting IoT botnet. In the work of Azmoodeh et. al. [19] which used opcode graph feature, we found it is difficult to justify the result because the dataset is relatively small, which contained 128 ARM-based malicious samples although the accuracy score is comparatively high (99.68%). Similarly, Hamed et. al. [22] proposed a recurrent neural network approach which used the dataset consisted of 281 ARM-based malicious samples and 270 ARM-based benign samples. Their work achieves 98.18%, which is slightly less than ours (0.52%).

Finally, over-fitting is a common problem that most deep learning algorithms run into. This occurs when the model is too fit for the training set but does not perform as well as on the extended subset. To evaluate the over-fitting issue in the proposed method, we added 3067 malicious samples collected from VirusShare into testing subset and recalculate the accuracy score. As shown in Table 5, there is a slight reduction in accuracy score when including malicious samples from VirusShare in the dataset (0.9%). From the experimental results, we conclude that the proposed method could address the over-fitting issue within an acceptable range.

## 5 Conclusion and future works

IoT devices rapidly increase in the foreseeable future. Thus, it is of importance to secure these devices against botnet attacks. In this study, we discussed the characteristics of IoT botnet and its effect on the detection of botnets. To address the limitations of previous methods, we proposed a lightweight approach which is based on high-level feature called PSI-graph for detecting IoT botnet. Our method discovered IoT botnet life cycle and took the advantages of deep learning methods to achieve an accuracy of 98.7%. Regarding future work, we plan to continue our research in discovering the aspects of botnet's life cycle. We expect the improvement of the current work when taking into account runtime information of the executable files.

## Compliance with ethical standards

**Conflict of interest** The authors declare that they have no conflict of interest.

**Ethical approval** This article does not contain any studies with human participants by any of the authors.

## References

1. Burhan, M., Rehman, R.A., Khan, B., Kim, B.-S.: IoT elements, layered architectures and security issues: a comprehensive survey. Sensors **18**(9), 2796 (2018)
2. Tankard, C.: Digital pathways, the security issues of the internet of things. Comput Fraud Secur **2015**(9), 11–14 (2015)
3. Gartner. https://www.gartner.com/newsroom/id/3291817. Accessed 10 Feb 2019
4. New trends in the world of IoT threats. https://securelist.com/new-trends-in-the-world-of-iot-threats/87991. Accessed 10 May 2019

5. Angrishi, K.: Turning Internet of Things (IoT) into Internet of Vulnerabilities (IoV): IoT Botnets, preprint (2017). arXiv:1702.03681
6. De Donno, Michele, Dragon, Nicola, Giaretta, Alberto: DDoS-Capable IoT Malwares: Comparative Analysis and Mirai Investigation. Journal Security and Communication Networks. Wiley, London (2018)
7. Pa, Y.M.P., Suzuki, S., Yoshioka, K., Matsumoto, T., Kasama, T., Rossow, C.: IoTPOT: a novel honenypot for revealing current IoT threats. J. Inf. Process. **24**, 522–533 (2016)
8. Tran, N.-P. et al.: Towards malware detection in routers with C500-toolkit. In: 5th International Conference on Information and Communication Technology (ICoIC7). IEEE, pp. 1–5 (2017)
9. Hampton, N., Szewczyk, P.: A survey and method for analysing SOHO router firmware currency. In: 13th Australian Information Security Management Conference, pp. 11-27 (2015)
10. Alhanahnah, M., Lin, Q., Yan, Q.: Efficient signature generation for classifying cross-architecture IoT malware. In: Conference on Communications and Network Security (CNS). IEEE, pp. 1–9 (2018)
11. Isawa, R.: Evaluating disassembly-code based similarity between IoT malware samples. In: 2018 13th Asia Joint Conference on Information Security (AsiaJCIS). IEEE, pp. 89–94 (2018)
12. Su, J., Vasconcellos D., Prasad, S., Sgandurra, D., Feng, Y., Sakurai, K.:Lightweight classification of IoT malware based on image recognition. In: 2018 42nd Annual Computer Software and Applications Conference (COMPSAC). IEEE, pp. 664–669 (2018)
13. Chang, K.-C., Tso, R., Tsai, M.-C.: IoT sandbox: to analysis IoT malware Zollard. In: Proceedings of the Second International Conference on Internet of things and Cloud Computing. ACM, pp. 4–12 (2017)
14. McDermott, C.D., Majdani, F., Petrovski, A.V.: Botnet detection in the internet of things using deep learning approaches. In: International Joint Conference on Neural Networks (IJCNN). IEEE, pp. 1–8 (2018)
15. Nath, H.V., Mehtre, B.M.: Static malware analysis using machine learning methods. In: Security in Computer Networks and Distributed Systems (SNDS). Springer, Berlin, pp. 440–450 (2014)
16. Kang, B., Yang, J., So, J., Kim, C.Y.: Detecting trigger-based behaviors in botnet malware. In: Proceedings of the 2015 Conference on research in Adaptive and Convergent Systems. ACM, pp. 274–279 (2015)
17. Kapoor, A., Dhavale, S.: Control flow graph based multiclass malware detection using bi-normal separation. Def. Sci. J. **66**(2), 138–145 (2016)
18. Cozzi, E., Graziano, M., Fratantonio, Y., Balzarotti, D.: Understanding Linux Malware. In: Symposium on Security and Privacy. IEEE, pp. 870–884 (2018)
19. Azmoodeh, A., Dehghantanha, A., Choo, K.K.R.: Robust malware detection for internet of (battlefield) things devices using deep eigenspace learning. IEEE Trans. Sustain. Comput. **4**(1), 88–95 (2018)
20. Islam, R., Tian, R., Batten, L.M., Versteeg, S.: Classification of malware based on integrated static and dynamic features. J Netw. Comput. Appl. **36**(2), 646–656 (2013)
21. Nguyen, H.T., Ngo, Q.D., Le, V.H.: IoT botnet detection approach based on PSI-graph and DGCNN classifier. In: International Conference on Information Communication and Signal Processing (ICICSP). IEEE, pp. 118–122 (2018)
22. HaddadPajouh, H., Dehghantanha, A., Khayami, R., Choo, K.K.R.: A deep recurrent neural network based approach for internet of things malware threat hunting. Future Gener. Comput. Syst. **85**, 88–88 (2018)
23. Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S.: Malware images: visualization and automatic classification. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security. ACM, pp. 4–11 (2011)
24. Jung, B., Kim, T., Im, E.G.: Malware classification using byte sequence information. In: Proceedings of the Conference on Research in Adaptive and Convergent Systems. ACM, pp. 143–148 (2018)
25. Hachem, N., et al., Botnets: lifecycle and taxonomy. In: 2011 Conference on Network and Information Systems Security. IEEE, pp. 1–8 (2011)
26. Silva, S.S.C., et al.: Botnets: a survey. Comput. Netw. **57**(2), 378–403 (2013)
27. Sudhakar, K., Kumar, S.: Botnet detection techniques and research challenges. In: International Conference on Advances in Energy-Efficient Computing and Communication (2019)
28. Khoshhalpour, E., Shahriari, H.R.: BotRevealer: behavioral detection of botnets based on botnetlife-cycle. ISC Int. J. Inf. Secur. **10**(1), 55–61 (2018)
29. Prokofiev, A.O. et al.: A method to detect internet of things botnets. In: Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus). IEEE, pp. 105–108 (2018)
30. Narayanan, A. et al.: graph2vec: Learning Distributed Representations of Graphs, preprint (2017). arXiv:1707.05005
31. Xu, M., et al.: A similarity metric method of obfuscated malware using function-call graph. J. Comput. Virol. Hacking Tech. **9**(1), 35–47 (2013)
32. Detect-It-Easy. https://github.com/horsicq/Detect-It-Easy. Accessed 12 Feb 2019
33. The Ultimate Packer for eXecutables. https://github.com/upx. Accessed 12 Feb 2019
34. Eagle, C.: The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler. William Pollock, Clifton (2011)
35. Shang, S., Zheng, N., Xu, J., Xu, M., Zhang, H.: Detecting malware variants via function-call graph similarity. In: International Conference on Malicious and Unwanted Software. IEEE, pp. 113–120 (2010)
36. Hallman, R., Bryan, J., Palavicini, G., Divita, J., Romero-Mariona, J.: IoDDoS-the internet of distributed denial of sevice attacks. In: Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security (IoTBDS), pp. 47–58 (2017)
37. Le, Q., Mikolov, T.: Distributed Representations of Sentences and Documents. In: Proceedings of the 31st International Conference on Machine Learning, pp. 1188–1196 (2014)
38. DeepBench. https://github.com/baidu-research/DeepBench. Accessed 10 Feb 2019
39. Kim, Y.: Convolutional neural networks for sentence classification. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). Association for Computational Linguistics, pp. 1746–1751 (2014)
40. VirusShare. https://virusshare.com. Accessed 13 Jan 2019
41. Firmware Analysis Toolkit. https://github.com/ReFirmLabs/binwalk. Accessed 13 Jan 2019
42. OpenWrt. https://openwrt.org/. Accessed 13 Jan 2019
43. Pytorch. https://github.com/pytorch/pytorch. Accessed 13 Jan 2019
44. VirusTotal. https://www.virustotal.com. Accessed 14 Jan 2019