An Efficient Algorithm to Extract Control Flow-based Features for IoT Malware Detection

TRAN NGHI PHU[1], NGUYEN DAI THO[2], LE HUY HOANG[1], NGUYEN NGOC TOAN[1] AND NGUYEN NGOC BINH[3]

[1]*People's Security Academy (PSA), Hanoi, Vietnam*
[2]*VNU University of Engineering and Technology, Hanoi, Vietnam*
[3]*The Kyoto College of Graduate Studies for Informatics (KCGI), Kyoto, Japan*
Email: tnphvan@gmail.com

Control flow-based feature extraction method has the ability to detect malicious code with higher accuracy than traditional text-based methods. Unfortunately, this method has been encountering with the NP-hard problem, which is infeasible for the large-sized and high-complexity programs. To tackle this, we propose a control flow-based features extraction dynamic programming algorithm (CFD) for fast extraction of control flow-based features with polynomial time O(N^2), where N is the number of basic blocks in decompiled executable codes. From the experimental results, it is demonstrated that the proposed algorithm is more efficient and effective in detecting malware than the existing ones. Applying our algorithm to an IoT dataset gives better results on 3 measures: Accuracy (AC) = 99.05%, False Positive Rate (FPR) = 1.31% and False Negative Rate (FNR) = 0.66%.

Keywords: IoT malware detection; Control flow-based features; Dynamic programming; CFD; Embedded malware

Received 18 October 2019; revised 20 December 2019

1. INTRODUCTION

Internet of Things (IoT) devices are increasingly targeted by adversaries due to their unique characteristics such as constant online connection, lack of protection, and full integration in people's daily life. IoT devices such as routers, cameras, TVs, and VoIP phones are now everywhere. As the number of IoT devices increases exponentially every year, IoT malware also grows accordingly in number and diversity. IoT botnets generated more than 750,000 spam emails per day.[4] Especially in October 2016, the malware Mirai infected and controlled more than 100,000 IoT devices worldwide and created the largest DoS attacks in history with a capacity exceeding 1.5 Tbps [1].

IoT devices use hardware components with small to medium-size software drivers and applications to enable a limited interface to those components [2]. One of the most challenging problems in designing IoT malware detector is that the extracted features should be compact and accurate, and the detection algorithm should be light-weight and energy-efficient to be deployed locally [3].

[4]Kaiser T. (2014) Hackers Use Refrigerator. *Other Devices to Send 750,000 Spam Emails.* http://www.dailytech.com/

There have been many results of IoT malware detection recently [3, 4, 5, 6]. In [4], 46 pieces of active mobile malware were identified and classified by payload behaviors. Pa *et al.* [5] constructed a sandbox environment for dynamic analysis of malware attacks against Telnet-based IoT devices running on different CPU architectures. Alhanahnah *et al.* [3] proposed a novel signature generation method for cross-architecture IoT malware. Novom *et al.* [6] utilized the fuzzy and fast fuzzy pattern tree methods to detect and categorize malware of IoT edge nodes after transmuting executable's OpCodes into a vector.

Malware detection and classification can be implemented using a static approach [7], in which the pieces of the malware are analyzed and examined for reason about their behavior without actually running them. There have been many static malware detection methods such as header information, Control Flow Graphs, OpCodes, API call graphs, etc. Davidson *et al.* [8] employed symbolic execution to automatically detect vulnerabilities and malware in the firmware of embedded devices. Nguyen *et al.* [9] proposed another static analysis method to detect botnet malware in IoT devices, based on convolution neural networks with Printable

Strings Information (PSI) extracted from malware samples. Angr [10] is a typical binary analysis open-source toolkit [5], with the capability to perform many state-of-the-art static analysis techniques such as control flow recovery, flow modeling, data modeling, concrete execution and symbolic execution. Kruegel *et al.* [11] used several techniques including Control Flow Graph (CFG), Data Flow Graph (DFG), Symbolic Execution (SE) to analyze every single file found in firmware and identify malware characteristics such as bytecode, headers, system calls or PSI.

According to our experiments, Table 1 shows IoT malware rarely used packing techniques. There are only less than 7.0% number of samples were packed, and most of them used the simple packer UPX [6]. The reason may be limited resources and weak secure policies of the packaged environment of embedded devices. Static malware analysis can ensure complete code coverage and reveal all possible actions that a malware may carry out. Therefore, it is an efficient method for IoT malware analysis when obfuscation techniques are not commonly used by this malware.

An operation code (opcode) is a part of a machine instruction that specifies the operation to be performed. It is a common type of features extracted by static analysis and widely used for malware detection. The opcode sequences extracted from disassembled executable files represent the essential behaviors of a program and can be obtained by static analysis. It was firstly proposed by Bilar [12], and then developed by Robert *et al.* [13], Santos *et al.* [14] and Ding *et al.* [15].

Santos *et al.* [14] suggested the Idea method to detect variants of known malware families based on the appearance frequency of opcode sequences. They developed a vector representation of executable files for machine-learning algorithms to detect unknown malware variants [16]. Ding *et al.* [15] claimed that above opcode-based extraction methods, called text-based methods, only show of file information but not the structures or behaviors of the program. Therefore, they proposed a new feature extraction method based on combining opcode and CFG, namely Control flow-based opcode features extraction method, which achieved higher accuracy than the text-based methods due to extracting more features of the decompiled executable codes through CFG's structure.

Traditional computing environments mainly based on X86 and/or X64 architecture are of Complex Instruction Set Computer (CISC). In heterogeneous IoT networks, IoT devices use embedded processors such as ARM, MIPS, X86-64, PowerPC, SPARC, etc [17], which are of Reduced Instructions Set Computer (RISC). CISC has more instructions than RISC, for example, in our experiments there are 340 Intel 80386 instructions but only 140 MIPS ones. Besides, each instruction of CISC contains more functions than that of RISC. Therefore, with the same function, an Intel executable file has more opcodes than a MIPS one. Our experiments also show that the average number of vertices, edges and opcodes per one basic block of the CFGs extracted from Intel samples is less than from MIPS ones. There are only 13.8% of CFGs of Intel malware samples consisting of more than 11,000 vertices, while for MIPS malware samples, more than a quarter of the CFGs having this characteristic. It means that the CFGs of MIPS executable files tend to be more complex than the CFGs of Intel executable files.

Ding *et al.*'s problem was solved by listing all execution paths from a root vertex to leaf vertices in the CFG of an executable [15]. The root vertex is the entry point of the executable and leaf vertices are endpoints of the executable. The graph has one root, many leafs and a large number of paths from the root to leafs. The Depth First Search(DFS) is used to enumerate all paths, but a large memory is needed to store these paths. However, the DFS-based solution is not efficient due to the repeated calculations. A complete graph with N vertices has $N!$ paths, thus finding all paths of this graph by DFS is an NP-hard problem. Besides, the Ding *et al.*'s experiment used only a small-sized dataset of 650 benign executable files and 650 malicious executable files of the MS Windows PE format. As the experiment on a larger-sized IoT dataset, with CFG has 11,000 vertices, it takes 40 seconds to find only one path, and sometimes no path can be found. It is possible that, Ding *et al.*'s method might be too slow to apply to larger-size available Intel datasets, especially for MIPS ones, which is one of the most popular IoT samples set [17]. Thus, Ding *et al.*'s method can only be applied to a simple CFG of decompiled executable file, which has few vertices and edges. In case of many vertices, it cannot find all the paths, even it is impossible to find any path within the specified time, which leads to the lack of information about CFG and low detection capacity.

Costin *et al.* [18] reported that among the embedded operating systems in the investigated dataset Linux suffered the most, accounting for more than three quarters of 32,000 analyzed firmware images. Pa *et al.* [5] proposed IoTPOT, a honeypot collecting about 4,000 IoT malware samples such as Tsunami, Mirai, Bashlite etc. Another IoT malware database is Detux [19] with more than 9,000 samples. Beside IoT malware samples, it is also crucial to collect benign files to be able to implement detection algorithms. Azmoodeh *et al.* [20] has collected 1,078 benign and 128 malware samples for ARM-based IoT applications. Alhanahnah *et al.* [3] stated that the IoT malware dataset provided by IoTPOT was the largest IoT malware dataset available at that time. In their experiments, only a set of 130 benign IoT samples is collected, which is insufficient to a learning-based malware detector.

[5]Https://angr.io
[6]Packer UPX - https://upx.github.io

In this paper, we further design efficient heuristic algorithms, which could extract control flow-based features by a complexity of O(N^2), improve the NP-hard algorithm of Ding *et al.*'s method, and we call it as control flow-based features extraction dynamic programming algorithm (CFD). The proposed algorithm is based on dynamic programming to build the weighted graph, whose each vertex's label having a number of execution paths passing over. From the vertices' label, control flow-based features are extracted. This study contains the following contributions:

- We propose the CFD for extracting control flow-based features within O(N^2) time, where N is the number of basic blocks in decompiled executable codes from the considered program. CFD allows to process large files without large memory to extract more feature information at a reasonable amount of time and achieves high accuracy.
- We build a huge dataset consisting of IoT samples and PC samples to compare our method with previous approaches. Supporting the research activities, we publish our dataset at address *https://gitlab.com/Nghiphu/c500iotdataset* with registered access.

2. RELATED WORK

The directed graph is related to many related fields such as language processing [21], online social networks [22], and malware detection [15]. One of the main problems is identifying, recognizing and removing the presence of cycles. It is necessary to have a principle technique to reduce a directed graph into a directed acyclic graph (DAG), which only includes acyclic relationships [21]. However, finding an automatic solution for this problem has been challenging. Existing approaches for this problem fall into the following categories: the simple DFS or the Breadth First Search (BFS) based heuristics [15, 23, 24] to eliminate and remove cycles; theoretical solutions that model the problem as variants of minimum-feedback arc set problem [25] or other NP-hard optimization problems; complex domain-specific algorithms [26, 27] that eliminate cycles based on many criteria, including redundancy and confidence of sources asserting the relations. Depending on the optimal purpose, heuristic algorithms have been proposed to construct DAGs. Sun *et al.* [21] proposed techniques to remove the cycles while preserving the logical structure (hierarchy) of a directed graph as much as possible to break cycles in noisy hierarchies. Canh *et al.* [22] proposed the maximizing misinformation restriction problem with the purpose of finding a set of nodes whose removal from a social network maximizes the influence reduction from a source of misinformation within time and budget constraints.

Angr [10] is an open source tool and binary analysis framework that integrates many of the state-of-the-art binary analysis techniques. . The framework has been encouraging the development of next-generation binary analysis techniques by implementing, effective techniques from current research efforts in an accessible and reusable methodology, so that they can be easily compared with each other. Angr provides building blocks for various analysis methods including both static and dynamic ones, so that proposed research approaches can be easily implemented and their effectiveness can be compared to each other. Additionally, the building blocks enable the composition of different analysers to leverage their different strengths. Angr supports two CFG extraction methods as CFGFast and CFGEmulated. The CFGFast, based on using symbol and heuristics to determine file functions, has the same CFG extraction algorithm as IDA[7].Besides, while CFGEmulated uses force execution to add basic blocks, backward slicing, and symbolic back-traversal, CFGFast employs light-weight analysis to calculate indirect jump commands [10]. Angr's CFGFast (or IDA) is as good as CFGEmulated if the binary file is well structured. The CFGEmulated provides a step-by-step simulation of the execution of a file and traces all the states, thus it can give the most accurate CFG which is constructed from the basic blocks.

The Ding *et al.*'s method extracted control flow-based features of a decompiled executable by calculating n-gram frequency of all execution paths on its CFG. Firstly, it constructed a CFG from a program, traversed the CFG to obtain all possible execution paths, and then concatenated them together. Cycles in CFG were detected and deleted by the DFS method when enumerating execution paths. From that, the CFG is converted into a tree, which is called an execution tree. In fact, each vertex of the execution tree is a basic block, which consists of an opcode sequence. Secondly, each vertex in an execution path is replaced with another opcode sequence, so that an opcode stream is generated from the execution path. Finally, a feature vector is calculated by counting n-gram opcode frequency of the opcode stream.

As an example, in Figure 1, a control flow graph is represented as the directed graph G, which has 5 vertices. The root is vertex 0 and there is only one leaf which is vertex 5. By Ding *et al.*'s method, control flow-based feature vectors are constructed by calculating n-gram frequency vectors from all found paths. Firstly, it searches on this graph to find possible paths. The results are three paths, namely:

(0, 2, 5), (0, 1, 4, 5), and (0, 1, 3, 4, 5)

Secondly, all the found paths will be concatenated together to become a stream, such as:

(*0, 2, 5*, 0, 1, 4, 5, *0, 1, 3, 4,5*)

Thirdly, each vertex in the stream is replaced with an opcode sequence, such as:

[7]Hex-Rays SA. IDA pro Introduction *http://www.hex-rays.com/products.shtml/*

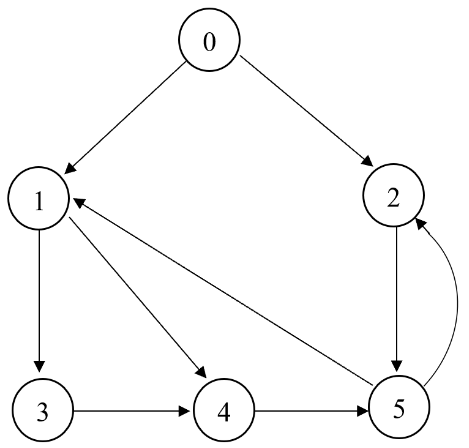

FIGURE 1. A Control Flow Graph - G

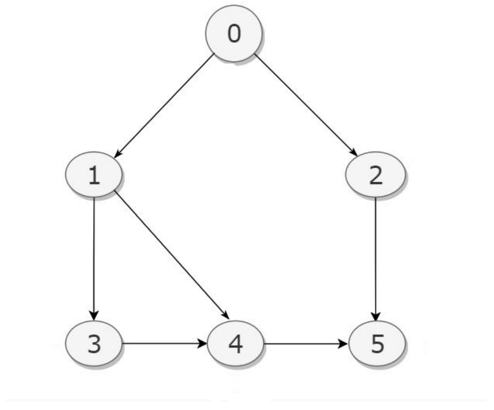

FIGURE 2. The acyclic control flow graph - ACFG

$0 \rightarrow$ "*push add jcxz*",
$1 \rightarrow$ "*push add*",
$2 \rightarrow$ "*call pop pop jbe*",
$3 \rightarrow$ "*add jcxz push jcxz*",
$4 \rightarrow$ "*mov inc add jbe*",
$5 \rightarrow$ "*mov inc mov push*".
This results in the opcode graph in Figure 4 and the opcode stream:

push add jcxz call pop pop jbe mov inc mov push push add jcxz push add mov inc add jbe mov inc mov push push add jcxz push add add jcxz push jcxz mov inc add jbe mov inc mov push

3. OUR METHOD

3.1. Preliminaries

Definition 1 (*Control Flow Graph - CFG*): A CFG of a decompiled executable program is a directed graph $G = (V, E, r, L)$, where:

- V is a set of vertices, each vertex is a basic block in the decompiled executable program;
- E is a set of directed edges, which are used to represent jumps/calls/rets opcode between two basic blocks in the decompiled executable program. With a directed edge (u,v), u is the head and v is the tail. When traversing graph, u is v's parent node, and v is u's child node.
- r is the root vertex of in-degree 0 which contains the entry point of a decompiled executable program;
- L is a set of leaf vertices of out-degree 0, which contains the end points of the decompiled executable program.

Definition 2 (*Acyclic Control Flow Graph - ACFG*): A ACFG denoted by $GA = (V, A, r, L)$ is a directed graph without any cycle built from the control flow graph $G = (V, E, r, L)$ after removing some edges, where $A \subset E$.

Definition 3 (*Execution path*): An execution path on ACFG $GA = (V, E, r, L)$ is a path $P(v) = \{r, v_1, v_2, ..., v\}$, where $v_i \in V, v \in$ L.

Definition 4 (*Execution Directed Acyclic Graph - EDAG*): An EDAG denoted by $GC = (V, A, r, L, C, D)$ is a labeled directed acyclic graph built from a ACFG $GA = (V, A, r, L)$, where:

- C denotes the label set of vertices. $C[u]$, $u \in V$, is the number of execution paths on GA visiting vertex u;
- D denotes the weight set of edges. $D[u,v]$, $\{u, v\} \in E$, is the number of execution paths on GA traversing edge (u,v).

3.2. The main CFD algorithm

```
Algorithm 1 CFD algorithm
    Input: Decompiled executable: exeFile
    Output: Control flow-based features with n-gram: feature_vector
1: graph = GetCfgFromFile (exeFile)
2: acfg = ConstructDag(graph)
3: edag = ConstrucEdag(acfg)
4: feature_vector = ExtractControlFlowBasedFeature(edag)
    Return: feature_vector
```

We propose CFD, a dynamic programming algorithm for building an EDAG. The main flow of CFD is described in Algorithm 1 with the following three sub-algorithms. Firstly, a ACFG is constructed from the CFG of a decompiled executable program by Algorithm 2. Secondly, an EDAG is built from the ACFG by Algorithm 3. In constructing the EDAG, the current number of paths is calculated based on the previous results, therefore it performs more quickly than Ding *et al.*'s method. Finally, control flow-based features are extracted from the EDAG by Algorithm 4.

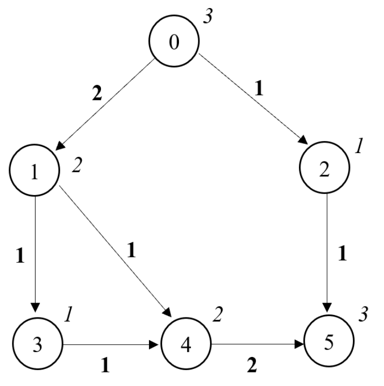

FIGURE 3. EDAG of G

3.3. Algorithm for constructing a ACFG

Algorithm 2 is based on the idea of DFS to build ACFG. Starting from the root vertex, it visits other ones, selects non back-edges to build a ACFG. When traversing to vertex u, it checks all of edges starting from u. If u doesn't belong to any path from the root to the edge's tail vertex, this edge will be updated to the ACFG and its tail vertex will be pushed to a *stack* (line numbers 9, 10). In each loop, the visited edge will be removed from G (line number 14). When the *stack* is empty, it means all edges are traversed, and these edges don't belong to any cycle that is added to ACFG. In addition, this algorithm can choose all vertices connected to the root, removes all unconnected vertices and guarantees connected component starting from the root. In Algorithm 2, function *getChildOfNodeList*(u) finds all vertices which are the child nodes of vertex u in graph G.

Algorithm 2 Constructing ACFG

```
    Input: CFG $G = \{V', E', r, L'\}$
    Output: ACFG G'= $\{V, E, r, L\}$
 1: allVertexBefore[i] = {i} $\forall i \in$ V'
 2: E = $\emptyset$
 3: V = {r}
 4: stack.push(r)
 5: while Not stack.IsEmpty() do
 6:     u = stack.pop
 7:     for v in getChildOfNodeList(u) do
 8:         if not v $\in$ allVertexBefore[u] then
 9:             E = E $\cup$ u,v
10:             stack.push(v)
11:             V = V $\cup$ v
12:             allVertexBefore[v] = allVertexBefore[v] $\cup$ allVertexBefore[u]
13:         end if
14:         E' = E' \ {u,v}
15:     end for
16: end while
17: L = L' $\cap$ V
    Return: G'
```

3.4. Algorithm for constructing an EDAG

Algorithm 3 Constructing EDAG from the ACFG

```
    Input: ACFG $GA = (V, A, r, L)$
    Output: EDAG $GC = (V, A, r, L, C, D)$
 1: #backward
 2: D = 0 #initialize 0 to all elements of matrix D
 3: P = 0 #initialize 0 to all elements of matrix P
 4: for u in V do
 5:     outDegree[u] = getOutDegree(u)
 6:     if outDegree[u] = 0 then
 7:         P[u] = 1
 8:         Stack.push(u)
 9:     end if
10: end for
11: while Not Stack.Empty() do
12:     currentNode = Stack.pop()
13:     for u in getParentOfNodeList(currentNode) do
14:         P[u] = P[u] + P[currentNode]
15:         D[currentNode,u] = P[currentNode]
16:     end for
17:     outDegree[u] = outDegree[u] - 1
18:     if outDegree[u] = 0 then
19:         Stack.push(u)
20:     end if
21: end while
22: #forward
23: C = 0 #initialize 0 to all elements of matrix C
24: for u in V do
25:     inDegree[u] = getInDegree(u)
26:     if inDegree[u] = 0 then
27:         Stack.push(u)
28:     end if
29: end for
30: C[r] = P[r]
31: while Not Stack.Empty() do
32:     u = Stack.pop()
33:     temp = C[u] / P[u]
34:     for v in getChildOfNodeList(u) do
35:         C[v] = C[v] + temp * D[v,u]
36:         D[u,v] = D[v,u] * temp
37:         inDegree[v] = inDegree[v] - 1
38:         if inDegree[u] = 0 then
39:             Stack.push(u)
40:         end if
41:     end for
42: end while
```

An EDAG has a matrix D that contains the weights of edges. If (u,v) is an edge, $D[u,v]$ is the number of execution paths containing the edge (u,v), called the weight of the edge (u,v), and $D[v,u]$ is the number of backward paths from leaves to the root containing the edge (u,v). Algorithm 3 for constructing an EDAG of an executable program has two phases: the backward phase computes the number of backward execution paths from leaves back the root and the toward phase computes weight of edges. In Algorithm 3, the function *getOutDegree*(u) counts the number of edges incoming to vertex u of the ACFG GA. The function *getParentOfNodeList(u)* returns all vertices which are parent vertices of vertex u. The function *getInDegree(u)* counts the number of edges outcoming to the vertex u. And the function *getChildOfNodeList(u)* returns all vertices which are child vertices of vertex u.

The EDAG in Figure 3 is the result of constructing EDAG from the ACFG in Figure 2. For the above directed acyclic graph DAG G in Figure 2, D and C are determined as follows:

$\{C[0] = 3,\ C[1] = 2,\ C[2] = 1,\ C[3] = 1,\ C[4] = 2,\ C[5] = 3\}$

$\{D[0,1] = 2;\ D[0,2] = 1;\ D[1,3] = 1;\ D[1,4] = 1;\ D[3,4] = 1;\ D[4,5] = 2;\ D[2,5] = 1\ \}$

3.5. Algorithm for extracting control flow-based features from an EDAG

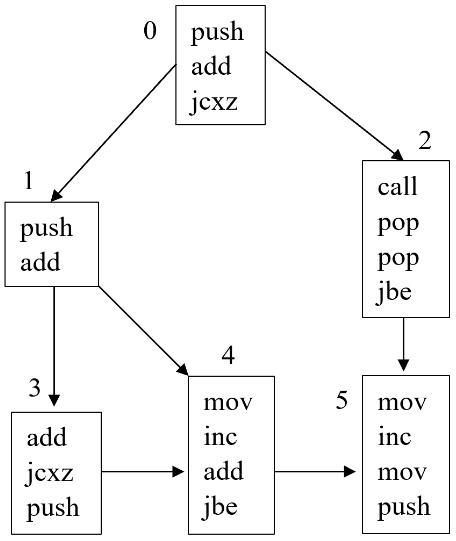

FIGURE 4. ACFG with opcode

A control flow-based feature vector is extracted from an EDAG with the n-gram method by Algorithm 4. Each n-gram is composed of n sequential opcodes. Control flow-based feature vectors can be extracted with different lengths of opcode n-grams, and the common length of n-gram is 2, 3, and 4 [16]. A length-fixed sliding window moves from the beginning to the end of an opcode stream to extract its n-grams. When extracting n-grams from the EDAG, the frequency of n-grams is determined by the weights of the edges and the vertices. Figure 5 shows an example of 3-gram features from vertices 4, 5 of the EDAG in Figure 3 and the ACFG with the opcode in Figure 4, the length of the sliding window in this example is three.

In Algorithm 4, function *getNgramConnect*(u,v) computes an n-gram opcode frequency vector of the opcode sequence that includes *(n-1)* opcodes at the end of the vertex u and *(n-1)* opcodes at the beginning of the vertex v, where n is the length of n-gram. Function *getNgramOfVertex*(u) calculates an n-gram opcode frequency vector of the opcode sequence of the vertex u.

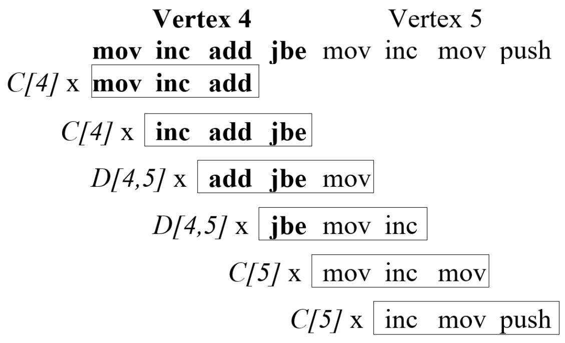

FIGURE 5. 3-gram opcode sequence from EDAG

Algorithm 4 Extracting control flow-based features with n-gram from EDAG

```
Input: EDAG GC = (V, A, r, L, C, D)
n: the length of n-gram

Output: A control flow-based features with n-gram vector
1: feature = 0
2: for u in V do
3:     sumU = 0
4:     for v in getChildNodesOf(u) do
5:         sumU = sumU + D[u,v]
6:         feature = feature + D[u,v] *getNgramConnect(u,v)
7:     end for
8:     feature = feature + getNgramOfVertex(u) * sumU
9: end for
```

3.6. Complexity analysis

The input of CFD is a graph $G = (V, E, r, L)$ obtained from a decompiled executable program, where N is the number of basic blocks in the program or the number of vertices in V; M is the number of edges in E. The complexity of the constructing ACFG (Algorithm 2) is O(M), because each edge in E is traversed only once and this algorithm stops when it traverses all edges.

The complexity of the constructing EDAG (Algorithm 3) is O(N^2). At the backward phase, there are two main loops at the 11th line and the 13th line. Each loop has the maximum complexity of O(N). Therefore, the complexity of this step is O(N^2). Similarly to the forward phase, the two main loops are in the 31th and 34th lines, and the complexity of this step is also O(N^2).

Extracting control flow-based features from Algorithm 3 with n-gram is also O(N^2), because it is determined by the number of *for* loops at line 2 and line 4 in Algorithm 4.

In summary, the complexity of CFD algorithm is O(M) + O(N^2) + O(N^2). M is less than N^2 in a directed graph, so its complexity is O(N^2).

4. EXPERIMENTAL SETTINGS AND RESULT ANALYSIS

4.1. Measures

There are many methods to evaluate a machine learning model. Our paper focuses on improving Ding *et al.*'s algorithm, thus the experiments use the same measures. In addition, we also use an integrated measure F1-Score, based on Precision and Recall as the basic measure of the detection model.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

where:

- TP: The number of malware samples truly predicted to be malicious applications.

- FP: The number of benign samples truly predicted to be malicious applications.

- FN: The number of malware samples truly predicted to be non-malicious applications.

- TN: The number of benign samples truly predicted to be non-malicious applications.

In order to assess the relationship between Precision and Recall, F1-score, which is defined as follows, is commonly employed:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3}$$

According to the traditional assessment, Accuracy (AC), False Positive Rate (FPR) and False Negative Rate (FNR) are used:

$$AC = \frac{TP + TN}{TP + TN + FP + FN} \tag{4}$$

$$FPR = \frac{FP}{TN + FP} \tag{5}$$

$$FNR = \frac{FN}{TP + FN} \tag{6}$$

4.2. Experiment Setup

Our experiments were run on a 64-bits Ubuntu 16.04.3 system, with 2x12-core CPUs, Intel Xeon E5-1600 family, and 64GB RAM. We extracted the CFGs of decompiled executable files by using Angr's CFGEmulated method. A Suport Vector Machine (SVM) machine learning model [28] was installed on the Scikit-learn Python library 0.19.2.

Comparing to the other machine learning models, SVM is highly efficient in binary classification and SVM is also used by Ding *et al.*. We employed SVM with Sigmoid kernel function in our experiments. Ding *et al.* suggested to fix C=100 and *gamma*=0.05 for Sigmoid kernel function. With these parameters, the proposed CFD algorithm is shown better than Ding *et al.*'s method in our experiments. But expecting more fairly, all parameters in the experiments are determined by the grid search method to choose the optimal values: C is selected in the range [1-100,000], the *gamma* is selected in the range [0.0000001-100]. In the experiments, we use the 5-folk cross-validation method [29] with the selected parameter set in training and evaluating steps. Data are divided into five different sections, four of them are used as training data and the remaining is used as testing data for each experiment. AC, FPR, FNR, and F1 are calculated as the average of results from five times of running experiments.

4.3. Dataset and test scripts

MIPS architecture is one of the most popular architectures in embedded systems, while Intel 80386 architecture is prevalent in PC, sometimes in embedded systems [17]. Embedded Linux is known to be the most widely used operating system of IoT devices [18, 30]. Samples of Intel 80386 and MIPS architectures are representative for two popular platforms: PCs and IoT devices; therefore, we focused on Executable and Linkable Format (ELF) files of Linux operating system.

The malicious MIPS samples were collected from IoTPoT [5], Detux [19], and Virus Share [8].The benign MIPS samples were extracted from more than 23,000 firmwares [31] such as Asus, Belkin, Tenvis, D-Link, TP Link, LinkSys, Trendnet, CenturyLink, Zyxel, OpenWrt, etc. The Intel 80386 benign samples were collected from a new Ubuntu Operating System setup on a PC with several common applications installed. The malicious Intel 80386 samples were collected from Virus Share and IoTPoT. After collecting, we filtered to keep only the executable ELF files and double-checked in Virus Total[9] to ensure the right labels. The dataset consists of 5,476 MIPS (1,896 benign and 3,580 malicious) samples and 6,560 Intel 80386 (1,428 benign and 5,132 malicious) ones. About 6.9% of samples are found being packed by Detect it Easy tool[10], in which most of them use UPX pack. The number of packed samples is presented in the *Packed* row of Table 1.

After unpacking, we extract CFGs from the samples, there are about 20% samples getting 'No Path Found' error. The number of packed samples is presented in the *CFG* column of Table 1. Most of the extracted CFGs

[8]https://virusshare.com/

[9]http://virustotal.com

[10]https://github.com/horsicq/Detect-It-Easy

TABLE 1. Statistical analysis of the dataset

Number of sample	MIPS Mal	MIPS Benign	Intel 80386 Mal	Intel 80386 Benign
Total	3,580	1,896	5,132	1,428
Packed	250		1,363	
CFG	2,780	1,720	4,056	1,367
T1 (Max: 40s)	1,996	1,128	3,308	1,017
T2 (CFD)	2,725	1,714	4,028	1,360

TABLE 2. Comparison between MIPS and Intel 80386 decompiled executable samples in term of the number of vertices in their CFGs

	MIPS	Intel
Average of opcode per a block	6.6	5.5
Average of edges	14,107	9,442
Average of vertices	8,310	5,578
Less than 5,000 vertices (%)	17.7	53.9
Less than 11,000 vertices (%)	73.7	86.2
More than 11,000 vertices (%)	26.3	13.8

have the number of vertices from 300 to 14,000.

With the collected CFGs, our method has the average time to calculate the control flow-based features of a CFG with 10 seconds, while the longest amount of time is 40 seconds. The CFD algorithm calculated successfully control flow-based features of all CFGs. Ding *et al.*'s method failed to find the final result for all CFGs within 40 seconds, even within 60 minutes for CFGs having more than 11,000 vertices. Thus, the experimental case with the threshold of 40 seconds as the slowest time to compute the number of paths. When 40 seconds were over, Ding *et al.*'s algorithm stopped looking for paths, and then gave the number of discovered paths.

We compared the ability to detect malware of two methods based on set T1, which is the set of samples as results after 40 seconds by both methods. Because our method extracted features successfully with all of the samples, it is also assessed the ability to detect malware on the all set, called T2. The number of samples in T1 and T2 of each architecture is presented in the *T1 (Max: 40s), T2 (CFD)* rows of Table 1 respectively.

With the same reason that we mentioned in our previous research [32], our experiment was carried out to reduce the dimension by Chi-Square [33], with feature numbers after decreasing respectively $K = 50$, 100, 150, 200, 250, 300 and 350.

4.4. Performance comparison of Ding *et al.*'s method and CFD

We compare Ding *et al.*'s method and our method on set T1 with both 2-gram and 3-gram. The comparison of Ding *et al.*'s method and CFD method in terms of Accuracy and False Positive Rate are shown in Figures 6 and 7. In the two figures, the dotted line shows accuracy of Ding *et al.*'s method, the solid line shows accuracy of CFD method, the dotted bars show FPR of Ding *et al.*'s method, the solid bars show FPR of CFD method, the horizontal axis represents K - the feature number after reducing by Chi-Square, and the left/right vertical axis represent the accuracy/FPR.

TABLE 3. Comparing between CFD and Ding *et al.*'s method

	Ding *et al.*'s method	CFD
Complexity of algorithm	NP-hard	$O(N^2)$
RAM consuming	6.0 GB	0.4 GB
Average of found paths	10^4	10^{303}
Rate of extracted features	86.2%	100%

Figure 6 shows the accuracy and FPR of the two methods with 2-gram. Figure 7 shows their accuracy and FPR with 3-gram. From Figures 6 and 7, we can draw the following results: the accuracy of CFD method is higher and its FPR is lower than Ding *et al.*'s method at almost value of K. In Figure 6, the highest accuracy of CFD method is 99.38% while the number for Ding *et al.*'s method is 99.29%. The lowest FPR belongs to our method. Figure 7 also demonstrates the same result as Figure 6. In summary, experiment results show that AC and FPR of our method are better than Ding *et al.*'s method with both 2-gram and 3-gram.

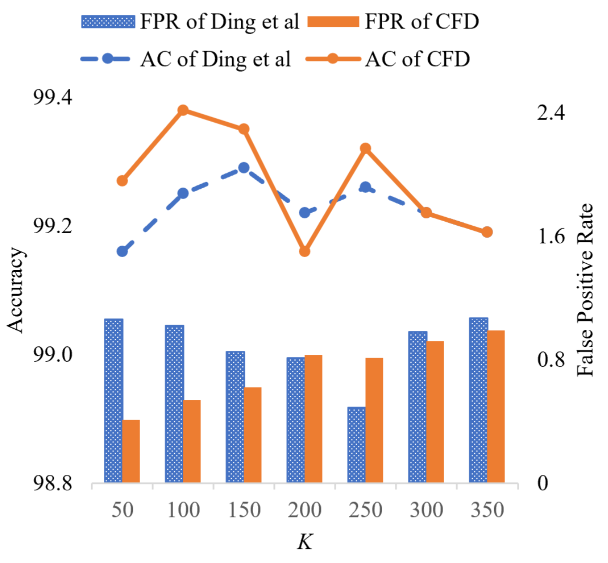

FIGURE 6. Comparing Ding *et al.*'s method and CFD based on 2-gram in term of accuracy and FPR

Table 3 gives comparison of CFD with Ding *et al.*'s method on the complexities of algorithms, RAM consuming, average of found paths and rate of extracted features. The average of found paths row shows that the CFD could get more and more the number of paths compare with Ding *et al.*'s method. The average number of paths found by Ding *et al.*'s method is of 10^4,

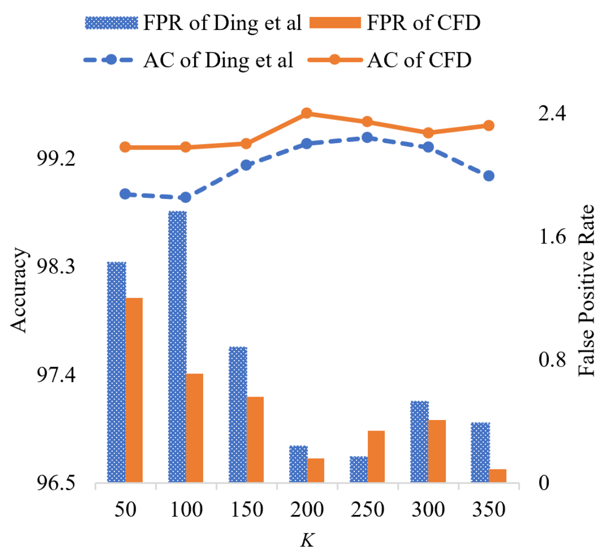

FIGURE 7. Comparing Ding *et al.*'s method and CFD based on 3-gram in term of accuracy and FPR

while the average number of paths found by our method is of 10^{303}. It means that CFD will extract a lot of features from CFG than the old method. There are 24% of CFGs could not extract any paths by Ding *et al.*'s method in 40 seconds. It means that it is impossible to detect these samples by Ding *et al.*'s method. Average of RAM consuming for extracting a feature vector from a CFG of the decompiled file, CFD only used 0.4 GB, but Ding *et al.*'s method consumed 6.0 GB, which is 15 times higher than ours.

4.5. Experimental evaluation of CFD algorithm

We evaluate the CFD method on set T2 of MIPS and Intel with both 2-gram and 3-gram. The comparison of 2-gram and 3-gram based CFD method in terms of Accuracy and False Positive Rate are shown in Figures 8 and 9. Figure 8 shows comparison on T2 of the Intel dataset, and Figure 8 shows results on the T2 of MIPS dataset. In Figure 8, the highest accuracy of 99.0% belongs to 2-gram based CFD at K = 200, going with low FPR of 2.25%. The accuracy of CFD method is higher than the accuracy of Ding *et al.*'s method at every value of K, and its FPR is lower for most of K values. In Figure 9, there are 5 values of K that 2-gram based CFD achieved higher accuracy and lower FPR than the one based on 3-gram. The lowest FPR belongs to 2-gram based CFD, but the highest accuracy of 99.0% going with low FPR of 1.53% belongs to 3-gram based CFD at K = 200. In summary, 2-gram based CFD is better than 3-gram based CFD on the Intel dataset, and still gives good results on the MIPS dataset although not reaching the highest accuracy.

Tables 4 and 5 give performance evaluation of 2-gram based CFD on MIPS and Intel samples of T2

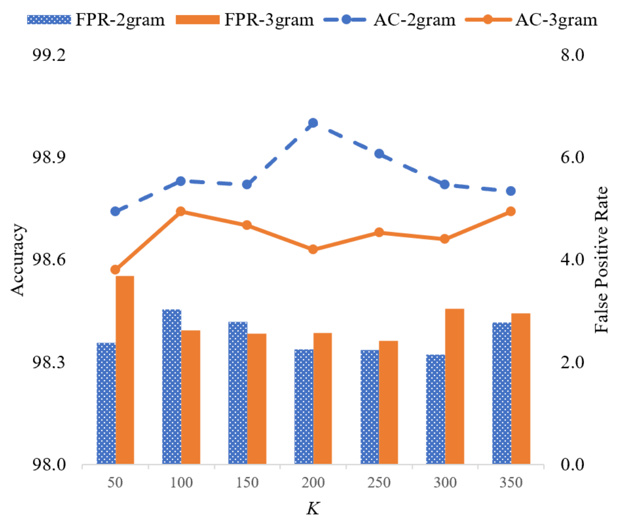

FIGURE 8. Comparing CFD base on 2-gram and 3-gram on the Intel dataset

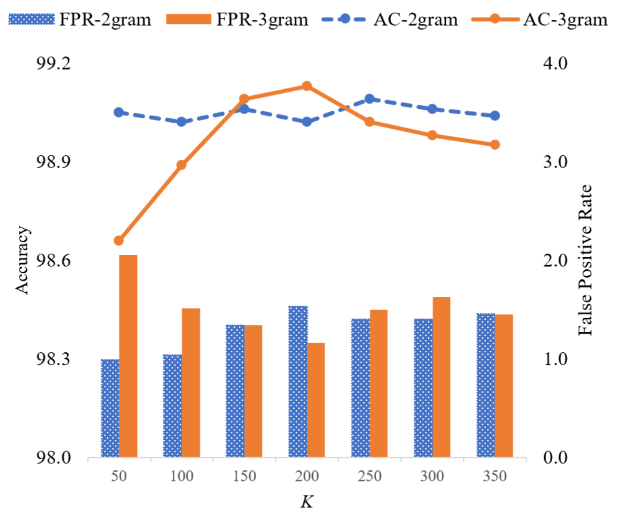

FIGURE 9. Comparing CFD base on 2-gram and 3-gram on the MIPS dataset

respectively. Figure 10 displays the comparison between 2-gram based CFD on MIPS and Intel samples of T2 in term of the average of measures. Results on the MIPS dataset are better than on the Intel dataset with higher accuracy, higher F1 and lower FPR, except slightly higher FNR.

5. CONCLUSIONS AND FUTURE WORK

Ding *et al.* proposed the control flow-based opcode feature extraction method to extract n-gram from an opcode stream concatenating all execution paths on the CFGs of executable. This method achieved higher accuracy compared with the former text-based feature extraction method. However, one of the main limitations of their algorithm is the use of the DFS strategy to find all paths in the execution tree of the program, which is an NP-hard problem. Hence, it can

TABLE 4. Evaluation CFD on the MIPS dataset

	FPR	***FNR***	***AC***	***F1***
50	0.99	0.73	99.05	99.27
100	1.04	0.91	99.02	99.15
150	1.34	0.64	99.06	99.19
200	1.53	0.60	99.02	99.15
250	1.40	0.57	99.09	99.20
300	1.40	0.60	99.06	99.19
350	1.46	0.60	99.04	99.17
Avg	***1.31***	***0.66***	***99.05***	***99.19***

TABLE 5. Evaluation CFD on the Intel dataset

	FPR	***FNR***	***AC***	***F1***
50	2.38	0.72	98.74	98.97
100	3.03	0.55	98.83	99.06
150	2.79	0.65	98.82	99.04
200	2.25	0.52	99.00	99.19
250	2.24	0.56	98.91	99.10
300	2.15	0.70	98.82	99.07
350	2.77	0.58	98.80	99.02
Avg	***2.52***	***0.61***	***98.85***	***99.06***

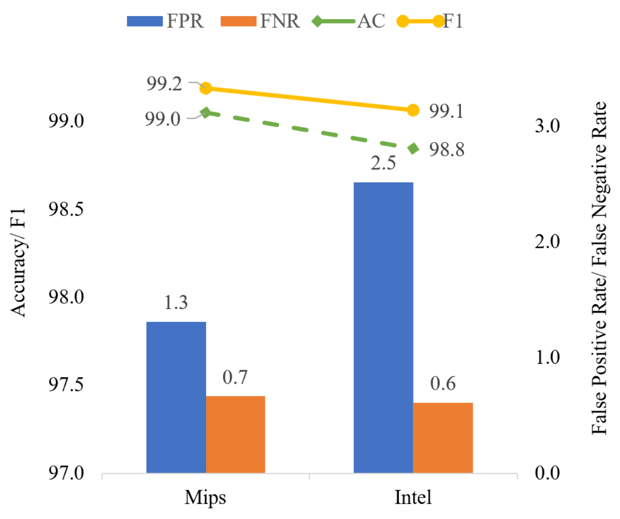

FIGURE 10. Comparing CFD on Intel and MIPS dataset

not be applied to large and complex files, especially for executable running on RISC architecture such as MIPS, which is used popular in IoT devices, that its CFG trends to more complex than CFG of CISC architecture such as Intel. To overcome this, we proposed CFD algorithm based on the idea of dynamic programming. The proposed algorithm can extract control flow-based features within O(N^2) time, where N is the number of basic blocks of decompiled executable codes.

Our experimental results demonstrated that the CFD algorithm successfully extracted control flow-based features of all the samples, while Ding *et al.*'s method was able to work with less than 75% of them and gives less information in the same time limit. The CFD has a higher speed, a less consuming memory and can find more paths than the old one many times. Therefore, it has a better ability to extract information from CFGs and capable of good grade more between malware and benign codes. In addition, it is also clarified that our proposed CFD algorithm can detect IoT malware better than PCs malware. We provided the IoT dataset used in our experiments to researchers for academic purposes.

Our future work will focus on (1) improving the efficiency of extracting CFGs from decompiled executable codes; and (2) eliminating cycles in graphs more efficiently.

REFERENCES

[1] Roger Hallman, Josiah Bryan, Geancarlo Palavicini, Joseph Divita, and Jose Romero-Mariona. (2017) *IoDDoS - The Internet of Distributed Denial of Sevice Attacks.* Proceedings of 2nd International Conference on Internet of Things, Big Data and Security, Porto, Portugal, 24-26 April, pp. 47-58, SCITEPRESS.

[2] Ralf Huuck. (2015) *IoT: The internet of threats and static program analysis defense.* Proceedings of Embedded World, Exibition & Conferences, Nuernberg, Germany, 15-17 Feb, pp. 493-495.

[3] Alhanahnah, Mohannad, Qicheng Lin, Qiben Yan, Ninh Zhang, and Zhenxiang Chen. (2018) *Efficient Signature Generation for Classifying Cross-Architecture IoT Malware.* Proceedings of Conference on Communications and Network Security (CNS), Beijing, China, 30 May-1 June, pp. 1-9, IEEE .

[4] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. (2011) *A Survey of Mobile Malware in the Wild.* Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, Chicago Illinois USA, October, pp. 3-14, ACM, New York, USA.

[5] Pa Yin Minn Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. (2016) *IoTPOT: A Novel Honeypot for Revealing Current IoT Threats.* Journal of Information Processing, 24, 522-533.

[6] Ensieh Modiri Dovom, Amin Azmoodeh, Ali Dehghantanha, David Ellis Newton, Reza M. Parizi, and Hadis Karimipour. (2019) *Fuzzy Pattern Tree for Edge Malware Detection and Categorization in IoT.* Journal of Systems Architecture, 97, 1-7.

[7] Mohsen Damshenas, Ali Dehghantanha Ali, and Ramlan Mahmod. (2013) *A Survey on Malware Propagation, Analysis, and Detection.* International Journal of Cyber-Security and Digital Forensics, 2, 10-29.

[8] Drew Davidson, Benjamin Moench, Somesh Jha, and Thomas Ristenpart. (2013) *FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution.* Proceedings of the 22nd USENIX conference on Security, August, pp. 463-478, USENIX Association 2560 Ninth St. Suite 215 Berkeley, CA, United States.

[9] Huy Trung Nguyen, Quoc Dung Ngo, and Van Hoang Le. (2018) *IoT Botnet Detection Approach Based on PSI Graph and DGCNN Classifier.* Proceedings of In-

ternational Conference on Information Communication and Signal Processing (ICICSP), Singapore, 28-30 Sept, pp. 118-122, IEEE.

[10] Yan Shoshitaishvili et al. (2016) *State of The Art of War: Offensive Techniques in Binary Analysis.* Symposium on Security and Privacy (SP), San Jose, CA, USA, 22-26 May, pp. 138-157, IEEE.

[11] Christopher Kruegel and Yan Shoshitaishvili. (2015) *Using static binary analysis to find vulnerabilities and backdoors in firmware.* Black Hat USA, New York, NY, USA, 5-6 August.

[12] Daniel Bilar. (2007) *Opcodes as Predictor for Malware.* International Journal of Electronic Security and Digital Forensics, 1, 156-168.

[13] Robert Moskovitch, Clint Feher, Nir Tzachar, Eugene Berger, Marina Gitelman, Shlomi Dolev, and Yuval Elovici. (2008) *Unknown Malcode Detection Using OPCODE Representation.* Proceedings of First European Conference on Intelligence and Security Informatics, Esbjerg, Denmark, 3-5 Dec, pp. 204-215, Springer, Berlin, Heidelberg.

[14] Igor Santos, Felix Brezo, Javier Nieves, Yoseba K. Penya, Borja Sanz, Carlos Laorden, and Pablo Garcia Bringas. (2010) *Idea: Opcode-Sequence-Based Malware Detection.* In Engineering Secure Software and Systems, Second International Symposium, Pisa, Italy, 3-4 Feb, pp. 35-43, Springer, Berlin, Heidelberg.

[15] Yuxin Ding, Wei Dai, Shengli Yan, and Yumei Zhang. (2014) *Control Flow-Based Opcode Behavior Analysis for Malware Detection.* Computers & Security, 44, 65-74.

[16] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo Garcia Bringas. (2013) *Opcode Sequences as Representation of Executables for Data-Mining-Based Unknown Malware Detection.* Information Sciences, 231, 64-82.

[17] Daming Dominic Chen, Manuel Egele, Maverick Woo, and David Brumley. (2015) *Towards Automated Dynamic Analysis for Linux-based Embedded Firmware.* The Network and Distributed System Security Symposium, California, USA, 21-24 Feb, pp. 1-16, EURECOM.

[18] Andrei Costin, Jonas Zaddach, Aurelien Francillon, and Davide Balzarotti. (2014)*A large-scale analysis of the security of embedded firmwares.* In Proceedings of the 23rd USENIX Security Symposium, San Diego, USA, 20-22 August, pp. 95-110, USENIX.

[19] Detux [Online].
Code Available at *https://github.com/detuxsandbox/detux* (accessed 23 Feb 2019).
Instance of Detux and collected samples at *http://detux.org* (accessed 23 May 2018).

[20] Amin Azmoodeh, Ali Dehghantanha, and Kim-Kwang Raymond Choo. (2018) *Robust Malware Detection for Internet of (Battlefield) Things Devices Using Deep Eigenspace Learning.* IEEE Transactions on Sustainable Computing, 4, 88-95.

[21] Jiankai Sun, Deepak Ajwani, Patrick K. Nicholson, Alessandra Sala, and Srinivasan Parthasarathy. (2017) *Breaking Cycles In Noisy Hierarchies.* Proceedings of The 9th International ACM Web Science Conference, New York, USA, 25-28 June, pp. 151-160, ACM.

[22] Pham Canh Van, Thai Tra My, Duong Van Hieu, Bui Quy Bao, and Hoang Xuan Huan. (2018) *Maximizing Misinformation Restriction within Time and Budget Constraints.* Journal of Combinatorial Optimization, 35, 1202-1240.

[23] Marco Fossati, Dimitris Kontokostas, and Jens Lehmann. (2015) *Unsupervised Learning of an Extensive and Usable Taxonomy for DBpedia.* Proceedings of the 11th International Conference on Semantic Systems (SEMANTICS' 15), Vienna Austria, 16-17 Sep, pp. 177-184, ACM.

[24] Osma Suominen and Christian Mader. (2014) *Assessing and Improving the ability of SKOS Vocabularies.* Journal on Data Semantics, 3, 47-73.

[25] Karp Richard Manning (1972) *Reducibility among Combinatorial Problems.* In: Miller R.E., Thatcher J.W., Bohlinger J.D. (eds) Complexity of Computer Computations. The IBM Research Symposia Series, Springer, Boston, MA.

[26] Bodenreider Olivier Mougin Fleur. (2005) *Approaches to Eliminating Cycles in the UMLS Metathesaurus: Nave vs. Formal.* American Medical Informatics Association Annual Symposium Proceedings, Washington, DC, USA, 22-26 Oct, pp. 550-554, American Medical Informatics Association (AMIA).

[27] Bodenreider Olivier. (2001) *Circular Hierarchical Relationships in the UMLS: Etiology, Diagnosis, Treatment, Complications and Prevention.* Proceedings of the American Medical Informatics Association Symposium, Washington, DC, USA, 3-7 Nov, pp. 57-61, AMIA.

[28] Shunichi Amari and Si Wu. (1999) *Improving support vector machine classifiers by modifying kernel functions.* Neural Network, 12, 783-789.

[29] Ron Kohavi. (1995) *A study of cross-validation and bootstrap for accuracy estimation and model selection.* Proceedings of the 14th international joint conference on Artificial intelligence, Montreal, Quebec, Canada, August, pp. 1137-1143, ACM.

[30] Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. (2015) *IoTPOT: Analysing the Rise of IoT Compromises.* In Proceedings of the 9th USENIX Conference on Offensive Technologies, Washington, USA, 12-14 August, pp. 9-19, USENIX.

[31] Tran Nghi Phu, Nguyen Ngoc Binh, Ngo Quoc Dung, and Le Van Hoang. (2017) *Towards Malware Detection in Routers with C500-Toolkit.* 5th International Conference on Information and Communication Technology (ICoIC7), Malacca City, Malaysia, 17-19 May, pp. 1-5, IEEE.

[32] Tran Nghi Phu, Kien Hoang Dang, Dung Ngo Quoc, Nguyen Tho Dai, and Nguyen Ngoc Binh. (2019). *A Novel Framework to Classify Malware in MIPS Architecture-Based IoT Devices.* Security and Communication Networks, 2019, 13 pages.

[33] Hiroshi Ogura, Hiromi Amano, and Masato Kondo. (2009) *Feature Selection with a Measure of Deviations from Poisson in Text Categorization.* Expert Systems with Applications, 36, 6826-6832.