C500-CFG: A Novel Algorithm to Extract Control Flow-based Features for IoT Malware Detection

Tran Nghi Phu
VNU University of Engineering and Technology
Hanoi, Vietnam
tnphvan@gmail.com

Le Huy Hoang
People's Security Academy (PSA)
Hanoi, Vietnam
hoangle.hvan@gmail.com

Nguyen Ngoc Toan
People's Security Academy (PSA)
Hanoi, Vietnam
ngoctoan.hvan@gmail.com

Nguyen Dai Tho
VNU University of Engineering and Technology,UMI UMMISCO 209
Hanoi, Vietnam
nguyendaitho@vnu.edu.vn

Nguyen Ngoc Binh
VNU University of Engineering and Technology
Hanoi, Vietnam
nnbinh@vnu.edu.vn

***Abstract*—Control flow-based features proposed by Ding, static characteristic extraction method, has the ability to detect malicious code with higher accuracy than traditional Text-based methods. However, this method resolved NP-hard problem in a graph, therefore it is not feasible with the large-size and high-complexity programs. So, we propose the C500-CFG algorithm in Control flow-based features based on the idea of dynamic programming, solving Ding's NP-hard problem by polynomial complexity O(N^2) algorithm, where *N* is the number of basic blocks in decompiled executable codes. Our algorithm is more efficient and more outstanding in detecting malware than Ding's algorithm: fast processing time, allowing processing large files, using less memory and extracting more feature information. Applying our algorithms with IoT data sets gives outstanding results on 2 measures: Accuracy = 99.34%, F1-Score = 99.32%.**

***Index Terms*—IoT, Malware detection, C500-CFG, CFG**

I. INTRODUCTION

The Internet of Things (IoT) appears more and more popular in every aspect of life. IoT devices such as routers, cameras, TVs, and VOIP phones are now everywhere. They constitute an important platform for the fourth industrial revolution. As the number of IoT devices increase exponentially every year, IoT malware also grows accordingly in number and diversity. IoT botnets generate more than 750,000 spam emails a day [1]. Especially in October 2016, the Mirai malware infected and controlled over 100,000 IoT devices worldwide and created the largest DDoS attacks in history with capacity exceeding 1.5 Tbps [2].

Some research works on the detection of malware in IoT devices have been conducted recently. In [3], 46 pieces of active mobile malware were identified and classified by payload behavior. Pa *et al.* [4] constructed a sandbox environment for dynamic analysis of malware attacks against Telnet-based IoT devices running on different CPU architectures.

Static malware analysis, which analyzed and examined for reasoning about their behaviors without actually running them [5], is an efficient method on IoT malware analysis and detection because encryption and obfuscation techniques are not commonly used by IoT malware. Kruegel *et al.* [10] used techniques such as 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 Printable Strings Information (PSI). Davidson *et al.* [6] made use of symbolic execution for automatically detecting vulnerabilities and malware in the firmware running on embedded systems. Another static analysis method was proposed by Trung *et al.* [7], for the detection of botnet malware in IoT devices, based on convolution neural networks applied to PSI extracted from the malware. Angr [8], [9] is a typical binary analysis toolkit, open source, with the capability to perform a variety of state-of-the-art static analysis techniques, from control flow recovery, flow modeling, and data modeling to concrete execution and symbolic execution.

Opcode is part of a machine instruction that specifies the operation to be performed. Opcode sequences describe the essential behaviors of a program and can be extracted through static analysis. Researches on malware detection using opcode and CFG have been interested in by many groups. Using opcode to detect malware, was firstly proposed by Bilar [11], afterward, many researches based on opcode like Robert [12] and Santos [13] have been done. Santos [14] has suggested the Idea method to detect variants of known malware families based on frequency of appearance of opcode sequences. This research is based on opcode sequences to build vector representation of the executable files, however Santos has not tested with longer sequences and other information like system calls. The researchers have also developed a vector representation of executable files used with machine-learning algorithms to detect unknown malware variants [13].

Ding [15] calls the opcode-based extraction methods of these previous researches as "text-based extraction methods", which only show information of files, do not show the characteristic structures or behaviors of a program. Therefore, Ding proposed a new way of detecting malware based on opcode called Control flow-based features. The Ding's method [15] has a higher accuracy than the text-based methods by

extracting more features of the executable file through CFG's structure. Ding' problem was solved by listing all paths from the root to leaf vertex in the graph. The root is the entry point of the program while the end point is leaf vertex. In this case, the graph has one root, many leaf vertices and very large number of paths. Therefore, the complete graph with *N* vertices has $(N!)$ paths, so this is the NP-hard problem.

Thus, Ding's method can only be applied to simple CFG files, with few peaks. In case of many vertices, we cannot find all the paths, even it is impossible to find the path within the specified time, so the lack of information on CFG and low detection capacity. Ding's method of finding characteristics is based on recursion so it uses a lot of memory, which we have experimented with CFG graph on 3,000 peaks, Ding's method used 8GB RAM to find the first path in 40 seconds. We performed statistically with MIPS ELF files, a common type of program in IoT devices using MIPS chips on Embedded Linux platform, the average number of vertices is 6,000 peaks for MIPS ELF static files, while dynamic ones is 1,000 vertices, so the majority of malware are static files.

II. OUR METHOD

Ding calculates Control flow-based features by constructing a control flow graph from a program and then traverse it to obtain all possible execution paths. With cycle in graph, the back edges were detected and deleted, therefore, Ding translated the CFG into a tree, which is called an execution tree. In fact, each vertex of the execution tree is a basic block, each basic block consists of opcode stream. It must be calculated n-gram of opcode stream in the basic block, then replace them for vertex in each path, but this step is completely independent and we don't mention. Now, the main problem is how to calculate n-gram of all execution paths in the CFG.

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

- *V* is a set of vertices, each vertex is a basic block in decompiled file.
- *E* is a set of edges, each edge *u,v* is directed, *u* call head and *v* call tail. When traversing graph, u is still call parent of v, v is call child node of v.
- *r* is the root vertex, which contains the entry point of executable file, has in steps equal to 0.
- *L* is a set of leaf vertices, which contains the end points of executable file, has out steps equal to 0.

Definition 2 (*C500-Graph*): C500-Graph *G = (V, E,r, L, D)* is a acyclic directed graph built from a Control Flow Graph *G* = (*V, E,r, L*), where *D*, label of vertices in *V*, is the number of execution paths go through vertices.

As an example, a control flow graph is presented like the directed graph *G* in Fig. 1, which has 5 vertices, the root vertex is 0 and only one leaf vertex is 5. By Ding's method, to calculate Control flow-based features, it will search on the graph and find all the paths. There are three paths, namely: (0, 2, 5), (0, 1, 4, 5) and (0, 1, 3, 4, 5)

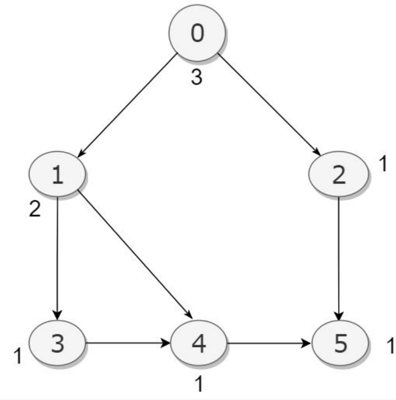

Fig. 1. The directed graph G

From this path set, we will find n-gram base on the set of all paths, which is called Control flow-based features. Specifically, for 2-gram Control flow-based features as follows:

[2-gram(0,1) = 2; 2-gram(1,3) = 1; 2-gram(1,4) = 1; 2-gram(3,4) = 1; 2-gram(4,5) = 2; 2-gram(2,5) = 1; 2-gram(0,2) = 1]

The search algorithm enumerates all paths by the Depth-first search, therefore, it needs a large memory to store paths and has a very slow speed because of repeated calculation. We propose a method based on dynamic programming method for building a C500-Graph which contains a number of paths from leaves to the root of the execution tree. In constructing the C500-Graph, the current path is constructed based on the previous results, so the tree will be formed. Then Control flow-based features are extracted from the C500-Graph. In this paper, Control flow-based features extraction is transformed into two sub-problems: constructing a C500-Graph and extracting Control flow-based features on the C500-Graph.

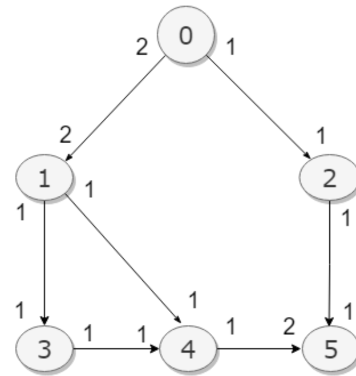

Fig. 2. C500-Graph of G

A. Constructing a C500-Graph

A C500-Graph contains *D* array, where *D[u,v]* is the number of paths between leaves and root. If *(u,v)* is an edge, *D[u,v]* is the number of paths from the root to leaves, *D[v,u]* is the

number of backward paths from leaves to the root. Constructing the C500-Graph of a program algorithm has 2 phases: the go-backward phase to compute number of backward paths, the go-toward phase to compute the number of paths from the root to leaves traversed each node. C500-Graph of the Control Flow Graph in Fig. 1 is presented Fig. 2.

```
    Input: Control Flow Graph GA = (V, A, r, L)
    Output: C500 Graph GC = (V, A, r, L, D)
 1: Fill(D, 0)
 2: Fill(C, 0)
    #go-backward
 3: for u in V do
 4:     step[u] = getStepOut(u)
 5:     if step[u] = 0 then
 6:         C[u] = 1
 7:         Stack.push(u)
 8:     else if  then
 9:         C[u] = 0
10:     end if
11: end for
12: while Not Stack.Empty() do
13:     currentNode = Stack.pop()
14:     for u in getParentOfNodeList(currentNode) do
15:         L[u] = L[u] + L[currentNode]
16:         D[currentNode,u] = L[currentNode]
17:     end for
18:     step[u] = step[u] - 1
19:     if step[u] = 0 then
20:         Stack.push(u)
21:     end if
22: end while
    #go-forward
23: for u in V do
24:     step[u] = getStepIn(u)
25:     if step[u] = 0 then
26:         Stack.push(u)
27:     end if
28: end for
29: while Not Stack.Empty() do
30:     cN = Stack.pop()
31:     temp = L[cN]/sumBackPaths(cN)
32:     for u in getChildOfNodeList(cN) do
33:         L[v] = temp * D[v,cN]
34:         D[cN,v] = D[cN,v] * temp
35:     end for
36: end while
```

Algorithm 1: Algorithm for constructing the C500-Graph of program

At the go-backward phase, there are two main loops at the 12th line and the 14th line. Each loop has the maximum complexity of *N*, where *N* is the number of vertices of Control Flow Graph, which is also the basic block number of program. Therefore, the complexity of this step is O(N^2). Similar to

```
   Input: C500-Graph GC = (V, A,r, L, D)
   Output: Control flow-based features with 2-gram
   two_gram
1: for u in V do
2:     for v in child_nodes_of(u) do
3:         two_gram(u,v) = D[u,v]
4:     end for
5: end for
```

Algorithm 2: Algorithm for Extracting Control flow-based features with 2-gram from C500-Graph

```
   Input: C500-Graph GC = (V, A,r, L, D)
   Output: Control flow-based features with 3-gram
   three_gram
1: for u in V do
2:     for v in get_child_node_of(u) do
3:         for x in get_child_node_of(v) do
4:             three_gram(u,v,x) = min(D[u,v], D[v,x])
5:         end for
6:     end for
7: end for
```

Algorithm 3: Algorithm for Extracting Control flow-based features with 3-gram from C500-Graph

the go-forward phase, the two main loops are in the 29th and 32th lines, and the complexity of this step is also O(N^2). So, the complexity of the C500-Graph construction algorithm is O(N^2).

B. Extracting Control flow-based features on C500-Graph

Control flow-based features could be extracted from a C500-Graph with n-gram feature extraction. The algorithm for extracting Control flow-based features with 2-gram is presented in Algorithm 2, and with 3-gram is presented in Algorithm 3.

Control flow-based features with 2-gram of the C500-Graph in Fig. 2 are:
[2-gram(0,1) = D[0,1] = 2; 2-gram(1,3) = D[1,3] = 1;
2-gram(1,4) = D[1,4] = 1; 2-gram (3,4) = D[3,4] = 1;
2-gram(4,5) = D[4,5] = 2; 2-gram(2,5) = D[2,5] =1;
2-gram(0,2) = D[0,2] =1]

III. The Experiments

A. CFG Extraction

We extract CFG of ELF file by Angr's CFGEmulated method due to high accuracy. Angr, which supports two CFG methods as CFGFast and CFGEmulated, is an open source tool. The CFGFast has the same CFG extraction algorithm as IDA [16] based on using symbol and heuristic to determine file functions. Besides, while CFGEmulated uses force execution to add basic blocks then using backwards slicing and symbolic back-traversal, CFGFast uses light-weight analysis to calculate indirect jump commands [8]. Angr's CFGFast (or IDA) is good as CFGEmulated if the binary file is well structured.

The CFGEmulated method step-by-step simulates program execution and following all states, so it can give the most accurate CFG, which is formed on basic blocks.

B. Dataset and test scripts

The experimental dataset consists of 7,000 MIPS ELF templates, which are Executable files on the Embedded Linux OS that run on the MIPS chip architecture. These include 3,699 MIPS ELF malware samples collected from Detux [17] and Virushare [18] with 3,400 MIPS ELF benign samples, which are extracted from more than 2,000 firmware of routers. Some samples of error templates were unsuccessfully extracted with Angr, so the total number of samples extracted from CFG was 5,560/ 7,000 samples. The results of the CFG extraction indicate that most CFGs have the number of vertices from 300 to 10,000, so the Control flow-based features method with CFGs has a peak of 300 to 10,000 to avoid noise. After filtering by a quantity of vertices, the sample set is 4,430 including 2,940 benign and 1,490 malwares.

With the CFG collected, the proposed method of C500-CFG has an average time to calculate the Control flow-based features of a CFG of 10 seconds, while the latest time is 40 seconds. The C500-CFG method calculates successfully Control flow-based features with any models. The old Control flow-based features extraction method did not calculate the final result for most models in the 40 seconds. For CFGs with above 6,000 vertices, the old method did not result in a 60-minutes. Thus, the experimental case with a threshold of 40 seconds as the slowest time to compute a quantity of paths and to detect malware. When the 40 seconds are over, our method will stop looking for a path and then it outputs the current number of found paths. This is not all the paths but only the quantities of paths found within the 40 seconds time limit of the old method.

The experimental results in 40 seconds indicated that the old method could not find all the paths, the quantities of found path in the average sample was 2.6x10^4 paths, much less than with the average number of paths found in the proposal method of 3.4x10^{303}. During the 40 seconds with 3,586 samples (81% of all samples), the old method could not find anything at all, couldn't calculate Control flow-based features, meaning there was impossibility of detecting these patterns with Control flow-based features. Therefore, the proposed method C500-CFG has a high speed and can find more paths than the old method many times, so it has ability to present information on the CFG, thus capable of good grade more between malware and benign code.

We compared the ability to detect malware of two methods based on set T1, which is a set of samples as results after 40 seconds by both methods. T1 includes 844 samples of 300 benign and 544 malware samples. We also assessed the ability to detect malware on the experimental set by our C500-CFG method. Set T2 is a malware set, which is extracted by our method, containing 2,940 benign and 1,490 malware samples. The experiment was carried out to reduce the dimension by ChiSquare [19] with characteristic number after decreasing respectively 50, 100, 150, 200, 250, 300 and 350.

C. Machine learning method

It can be seen that compared with machine learning methods, SVM is a highly efficient method of binary classification, is also used by Ding. We used SVM with the sigmoid function kernel and grid search method, which can find the best parameter set. With this data set, it is better than Ding's method when fixed suggesting C=100 and gamma=0.05. In the experiments, we use a 5-folk method with the best parameter set. Data are divided into five different sections include four training sections and one testing section for each experiment. The accuracy and F1-Score are calculated as the average of five times in this experiment.

D. Measurements

There are many methods to evaluate a machine learning model because it depends on each model and different data sets. Our paper focuses on algorithmic improvements introduced by Ding, so the experiments use the same measurement accuracy. In addition, here we also use F1-Score integrated measurement, based on Precision. Besides, Recall is the basic measure of machine learning model.

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

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

where:

- TP: The number of malware is predicted to be malware;
- FP: The number of benign is predicted to be malware;
- FN: The number of malware is predicted to be benign;
- TN: The number of benign is predicted to be benign.

In order to assess a relation between Precision and Recall, it often uses F1-Score and are defined as follows:

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

According to the traditional assessment, the accuracy of the model is used:

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

E. Performace comparison of proposed method

The experimental results comparing last methods and C500-CFG method are shown in Fig 3, 4, 5, 6. Fig 3 compares Accuracy while Fig 5 compares F1-Score when featured extraction equal to 2-gram with the two methods. With the method of taking 2-gram features, the old and new identifiers reach Accuracy and F1-Score values at K=300. Furthermore, proposed method is better than the old method at all thresholds with K value. Fig 4 showed Accuracy comparison, Fig 6 compares F1-Score when extracting features by 3-gram of the two methods. With the method of taking 3-gram features, the identifier is based on the old method of achieving Accuracy

and F1-Score values at K=150, while the proposed method remains stable at K=300. We can see that at all thresholds, the proposed method is better than the last one. Experiment results showed that the Accuracy and F1-Score indicators of our method outperform the Ding's method when using the 2-gram or 3-gram characteristic methods.

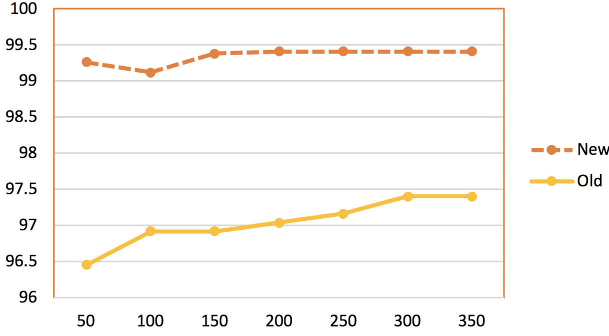

Fig. 3. Compare accuracy based on 2-gram between old and new ways

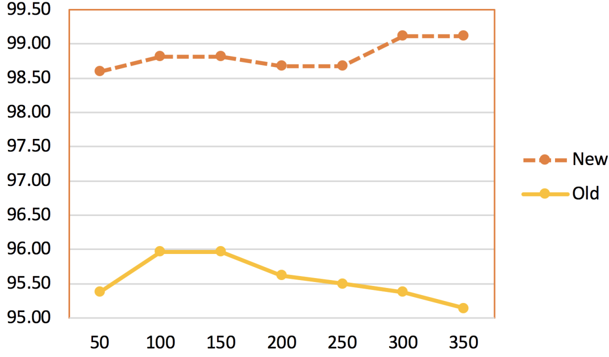

Fig. 4. Compare accuracy based on 3-gram between old and new ways

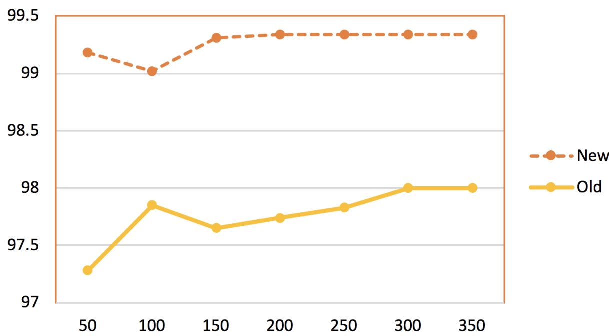

Fig. 5. Compare F1-Score based on 2-gram between old and new ways

F. Evaluation of our C500-CFG method

Fig 7 showed experimental results evaluating C500-CFG method. The best value model Accuracy and F1-Score with K=300, typically, Accuracy = 99.07% and F1-Score=98.65%. The 2-gram feature extraction method gives better results than the 3-gram one.

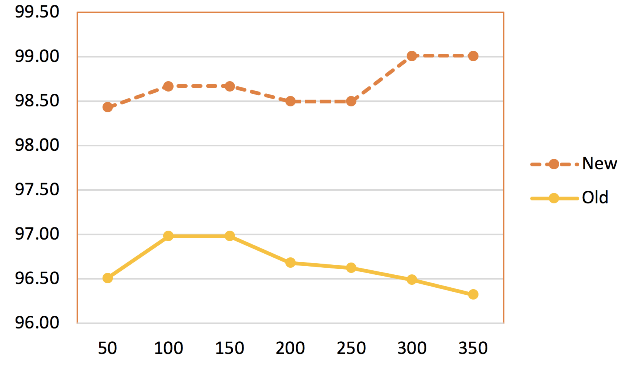

Fig. 6. Compare F1-Score based on 3-gram between old and new ways

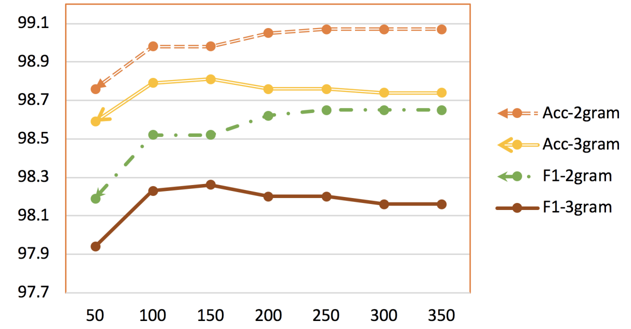

Fig. 7. Evaluation C500-CFG method

G. Evaluate 2-gram and 3-gram method

Ding stated that 3-gram gave the best results and experimented with 3-gram method. However, our experiments with Iot Dataset showed that 2-gram method gives better results. Experimental results comparing the effectiveness of 2-gram and 3-gram between Ding's method and C500-CFG in Figs 8 and 9 showed that the 2-gram method is superior to 3-gram method. In the same way, the optimal result of the model corresponding to K value when extracting features by 2-gram and 3-gram methods are different. With the 3-gram method, the best value was achieved with K=150 then gradually decreasing. Thus, when we choose to add a feature with large K, it will cause data interference to reduce the accuracy. This is also consistent with previous observations, if the greater the K, the higher the level of data sensitivity. Therefore, malicious models with high variability are difficult to detect.

IV. Conclusions and Future Work

Ding proposed the control flow-based features extraction method to extract opcode sequences with a higher accuracy compared with the former text-based features extraction method. However, one of the main limitations of Ding's algorithm is that it uses the deep first search strategy to find all paths in the execution tree of the considered program. This search problem is an NP-hard problem. Hence, it cannot be applied to large and complex files, such as a static MIPS ELF

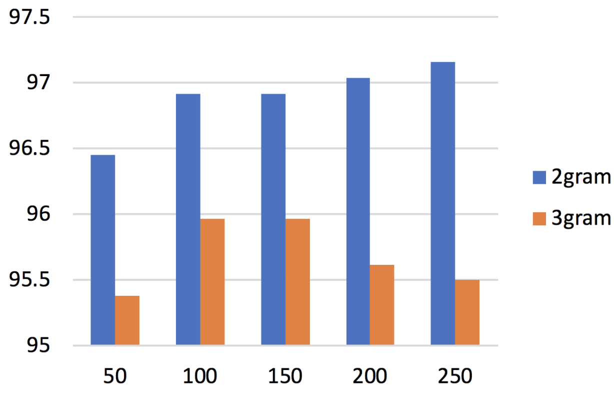

Fig. 8. Accuracy based on our proposal method between 2-gram and 3-gram

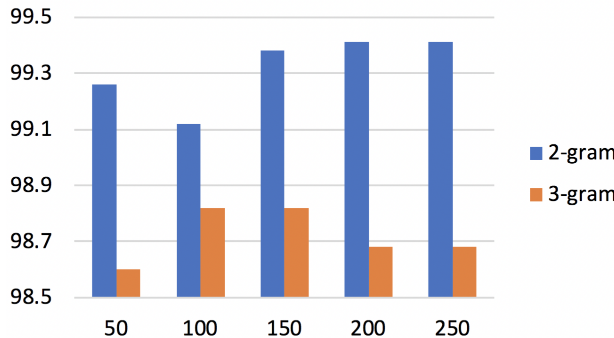

Fig. 9. Accuracy based on old method between 2-gram and 3-gram

with about 6,000 basic blocks in its decompiled executable codes. Our proposed C500-CFG algorithm is based on the idea of dynamic programming and could extract Control flow-based features by a complexity of O(N^2), where N is the number of basic blocks in decompiled executable codes from the considered program. Our experimental results showed that the C500-CFG algorithm could be applied to extract Control flow-based features of all samples, while Ding's method can only extract less than 20% of samples and gives less information in the same time limit. Evaluating on the same sample set, our proposed algorithm is with a higher accuracy (up to Accuracy = 99.34%, F1-Score = 99.32%), with a faster speed, with ability to process large files, and uses less memory. Our experimental results also clarify that the 2-gram feature extraction is better than the 3-gram feature extraction for the same IoT dataset. As our future work, we will (1) verify the C500-CFG algorithm with various data sets to evaluate the performance and effectiveness; (2) improve the efficiency of extracting CFG from decompiled executable codes; and (3) eliminate cycles in graphs more efficiently.

V. ACKNOWLEDGMENT

This research is funded by Vietnam Ministry of Public Security under Grant no. BX.2017.T31.01.

REFERENCES

[1] Hackers Use Refrigerator. *Other Devices to Send 750,000 Spam Emails*. http://www.dailytech.com/

[2] Roger Hallman, Josiah Bryan, Geancarlo Palavicini, Joseph Divita and Jose Romero-Mariona. *IoDDoS - The Internet of Distributed Denial of Sevice Attacks*. 2nd International Conference on Internet of Things, Big Data and Security. SCITEPRESS, p. 47-58, 2017.

[3] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. *A Survey of Mobile Malware in the Wild.* In Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, 3–14. SPSM '11. New York, NY, USA: ACM, 2011. https://doi.org/10.1145/2046614.2046618.

[4] Pa Yin Minn Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. *IoTPOT: A Novel Honeypot for Revealing Current IoT Threats.* Journal of Information Processing 24, no. 3 (2016): 522–33. https://doi.org/10.2197/ipsjjip.24.522.

[5] Damshenas, Dehghantanha Ali and Mahmod. *A Survey on Malware Propagation, Analysis, and Detection.* International Journal of Cyber-Security and Digital Forensics (IJCSDF) (April 2013). https://doi.org/10.5120/11480-7108.

[6] Drew Davidson, Benjamin Moench, Somesh Jha and Thomas Ristenpart. *FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution.* USENIX. Accessed July 17, 2018. ps://www.usenix.org/conference/ usenixsecurity13/technical-sessions/paper/davidson.

[7] Huy Trung Nguyen, Quoc Dung Ngo, and Van Hoang Le. *IoT Botnet Detection Approach Based on PSI Graph and DGCNN Classifier.* In 2018 IEEE International Conference on Information Communication and Signal Processing (ICICSP), 118–122, 2018. https://doi.org/10.1109/ICICSP.2018.8549713.

[8] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel and Giovanni Vigna. *State of The Art of War: Offensive Techniques in Binary Analysis*, IEEE Symposium on Security and Privacy (SP), 2016.

[9] Angr [Online]. Available *https://angr.io*

[10] Christopher Kruegel and Yan Shoshitaishvili. *Using static binary analysis to find vulnerabilities and backdoors in firmware*. in: Black Hat USA, 2015.

[11] Daniel Bilar. *Opcodes as Predictor for Malware.* International Journal of Electronic Security and Digital Forensics 1, no. 2 (2007): 156. https://doi.org/10.1504/IJESDF.2007.016865.

[12] Robert Moskovitch, Clint Feher, Nir Tzachar, Eugene Berger, Marina Gitelman, Shlomi Dolev and Yuval Elovici. *Unknown Malcode Detection Using OPCODE Representation.* In Intelligence and Security Informatics, edited by Daniel Ortiz-Arroyo, Henrik Legind Larsen, Daniel Dajun Zeng, David Hicks, and Gerhard Wagner, 204–215. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008.

[13] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero and Pablo Garcia Bringas. *Opcode Sequences as Representation of Executables for Data-Mining-Based Unknown Malware Detection.* Information Sciences, Data Mining for Information Security, 231 (May 10, 2013): 64–82. https://doi.org/10.1016/j.ins.2011.08.020.

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

[15] Yuxin Ding, Wei Dai, Shengli Yan and Yumei Zhang. *Control Flow-Based Opcode Behavior Analysis for Malware Detection.* Computers & Security 44 (July 1, 2014): 65–74. https://doi.org/10.1016/j.cose.2014.04.003.

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

[17] Shodan [Online]. Available *https://github.com/detuxsandbox/detux*

[18] Virusshare [Online]. Available *https://virusshare.com/*

[19] Hiroshi Ogura, Hiromi Amano and Masato Kondo. *Feature Selection with a Measure of Deviations from Poisson in Text Categorization.* Expert Systems with Applications 36, no. 3, Part 2 (April 1, 2009): 6826–6832. https://doi.org/10.1016/j.eswa.2008.08.006.