IoT Malware Detection Using Function-Call-Graph Embedding

Chia-Yi Wu*, Tao Ban‡, Shin-Ming Cheng*†, Bo Sun§, Takeshi Takahashi‡
* National Taiwan University of Science and Technology, Taipei, Taiwan
{m10915075, smcheng}@mail.ntust.edu.tw
†Research Center for Information Technology Innovation, Academia Sinica, Taipei, Taiwan
‡National Institute of Information and Communications Technology, Tokyo, Japan
{bantao,takeshi_takahashi}@nict.go.jp
§Saitama Institute of Technology, Saitama, Japan
sunshine@nsl.cs.waseda.ac.jp

2021 18th International Conference on Privacy, Security and Trust (PST) | 978-1-6654-0184-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/PST52912.2021.9647806

***Abstract*—In the era of rapid network development, IoT devices are being deployed more and more widely, and various kinds of malware programs are gradually appearing at the deployment level. As a widely adopted static analysis approach, structure based analysis such as graph embedding can capture the semantic features of malware binaries and has received much research attention. In this paper, to further improve the robustness of the graph embedding approaches to IoT malware detection, we propose a novel method that incorporates both local and global characterizing features extracted from Function-Call Graphs (FCG) to perform the detection. The caller-callee relationship represents the local semantic features, and the global statistic feature represents the graph's structural characteristics. The performance of the proposed method is evaluated on a large-scale dataset consisting of 112K malware and 89k benignware samples collected from seven CPU architectures. It shows a 99% accuracy on IoT malware detection, outperforming existing graph embedding solutions. Moreover, when CPU architecture is taken into consideration, the proposed method combined with support vector machine and multilayer perception classifier can yield even higher performance.**

***Index Terms*—Cybersecurity, function call graph, graph embedding, IoT malware, machine learning, static analysis**

I. Introduction

The popularity of IoT applications that have enriched human life has also attracted malware attacks that seek to breach human safety and privacy for potential revenue. Identifying malware and reducing its impacts becomes a critical topic for both industrial and academic research [1]. Meanwhile, the source code release of malware such as Mirai resulted in a tremendous number of malware variants, which renders the protection of IoT devices more challenging [2, 3]. By understanding the behavior of malware binaries using static analysis, we want to elucidate the actions intentionally designed to harm IoT devices and hence facilitate fast detection and mitigation of the potential threats [4, 5].

Recently, the graph-feature-based approach has received extensive research attention in the machine learning (ML) field. In this approach, a graph that captures the semantics of a sample is taken as the input to an ML algorithm []. As a well-known example, graph2vec [6] uses Weisfeiler-Lehmar (WL) [7] relabeling to extract rooted-subgraphs and generates the graph embedding using doc2vec [8]. Among the existing graph-based features extracted from binaries, Function-Call Graphs (FCGs), where the vertices specify function calls and the edges correspond to the caller-callee relationship between functions, are considered appropriate for representing the execution logic of the binary [9, 10, 11]. Nevertheless, an FCG usually has a very complicated structure and huge capacity, which requires an efficient transformation processing before it can be taken as input to a learning algorithm [12].

In this paper, we propose an IoT malware detection model using graph embedding based on FCG. To maintain both call information and function interaction relationship represented by FCG, we introduce some improvements to graph2vec. In particular, we retrieve a graph rooted from a specific function as input to graph2vec. In this way, the semantics of a particular function can be captured by analyzing the related graph structure.

To evaluate the generalization performance of the proposed malware detection method, we collected a large IoT malware dataset and performed a series of numerical investigations. The dataset comprises $112,728$ malware and $89,576$ benignware samples compiled on seven distinct CPU architectures: ARM, MIPS, X86, X86-64, PowerPC, SPARC, and UNKNOWN.

Among the three evaluated classifiers, i.e., random forest (RF), support vector machine (SVM), and Multilayer Perceptron (MLP), SVM demonstrated the best performance and obtained an accuracy rate up to 99.72% on the dataset when evaluated by 5-fold cross-validation. In the case of CPU-specific analysis, an accuracy rate of 99.76% was achieved. These results verify the efficiency of using FCGs to obtain discriminating models to facilitate malware detection.

The main contributions of the paper are summarized as follows.

- We improve the graph2vec performance based on FCG in IoT malware detection.
- We proposed a feature representation which integrates both graph embedding and structural features from FCGs.
- We utilize a large scale dataset to evaluate the performance and compare the proposed method with graph2vec.

- We reveal the necessity to perform CPU-specific analysis on IoT malware and provide solid results as a proof of concept.

The remainder of this paper is organized as follows. Section II reviews previous works related to background and malware analysis. Section III presents the proposed method. Section IV evaluates the performance of the proposed method. Section V concludes the paper.

II. Background and Related work

In this section, we introduce background of IoT malware and previous work about IoT malware detection based on dynamic analysis and static analysis.

A. *IoT malware*

In today's scenario, IoT devices play a significant role and are deployed more and more widely. Because IoT devices have limited computational abilities and hardware limitations and users lack security awareness, they have become the primary target of today's malware attacks. By exploiting vulnerabilities found in an IoT device, attackers can control it and force it to become a part of the IoT botnet. The most notorious IoT malware examples are Mirai and Bashlite [13], which have been involved in several large-scale distributed denial-of-service attacks (DDoS attacks) with widespread impact [14] . The cycle of their attacks is mainly divided into several steps. First, they scan for devices with full vulnerabilities as the primary target of their attack, and then they gain access to other vulnerable devices through some brute force attacks. After login, they use FTP, HTTP, or other protocols to download and install the newest botnet binaries. Finally, they will communicate with the C&C server awaiting the attack command.

The most famous attack of IoT malware is the DDoS attacks caused by Mirai to the DNS service of DNS service provider Dyn in 2016. These attacks made several well-known websites, including GitHub, Twitter, and many other websites, inaccessible. Finally, after the release of Mirai source code, it was discovered to be a Bashlite based mutation.

Another well-known malware family is Hajime. It has many consistent features and functions with Mirai and some new features as well. The most noteworthy one is that it will take multiple steps to hide its running process. The attacker can open the shell script on an infected device, and as the code is modular, they can perform additional functions instantly.

There are also other well-known IoT malware samples, such as Dofloo and Xorddos [15]. They had also launched large-scale DDOS attacks. Therefore, the problems of IoT malware must be dealt with efficiently to prevent further damages.

B. *IoT malware detection via dynamic analysis*

The primary objective of a malware countermeasure can be divided into two parts: identifying malware and defeating malware. The most intuitive way to identify malware is to analyze the execution process of malware. Dynamic analysis is to identify malware based on its execution flow and impact on hardware. As the execution flow will show the behavior of the malware, dynamic analysis has been a good choice for malware analysis.

Hou et al. [16] put the malware into a sandbox to generate the execution flow. The execution flow, i.e., the system call sequence, is converted into a system call graph. The number of occurrences of the system calls, the call relationship of the system calls, and the out-degree and in-degree of each node (system call) on the system call graph become the input features for learning algorithms.

Surendran et al. [17] identified the semantic behavior of the sequence on the system call graph. They calculated the shortest path between system calls and the occurrence rate of the system call itself. Based on these two values, the dependency between each system call and others was calculated as the semantic feature of this malware.

Amer et al. [18] used word2vec to analyze the similarity of individual system calls in the call sequences. The similarity analysis was used to cluster the system calls, and the functions of the calls within the same cluster were considered similar. They then replaced the system call in the system call string with the cluster number of the system call. This conversion made it easier to read the semantic behavior in the system call sequence.

C. *IoT malware detection via static analysis*

Although dynamic analysis can accurately capture the semantic behavior of malware, it requires too much computational time. The time cost and hardware resources for constructing the virtual environment cannot be ignored, especially when a large-scale set of samples are to be analyzed. Therefore, static analysis is also a suitable choice. Static analysis is mainly to disassemble and analyze the internal data of malware executable files. Static analysis can be divided into two subcategories: analyses based on graph features and non-graph features.

1) Graph-feature based approach: Nguyen et al. [11] proposed a lightweight method for detecting IoT botnet, which is based on extracted high-level features from function–call graphs, known as PSI-Graphs. These features showed effectiveness when dealing with multi-architecture problems and avoided the complexity of control flow graph analysis used by most of the existing methods. They also proposed a method named rooted-subgraph [19], which is a novel high-level PSI-rooted subgraph-based feature and generated a limited number of features with precise behavioral descriptions. The proposed method required smaller space and reduced processing time and showed effectiveness and robustness. Xiao et al. [20] proposed a graph repartition algorithm by transforming API call graphs into fragment behaviors based on programs' dynamic execution traces. The proposed algorithm relied on the N-order subgraph (NSG) for constructing the appropriate fragment behavior. Kawasoe et al. [21] proposed a graph-based method for confirming differences in malware behaviors and investigating the actual conditions of malware variants.

2) Non-graph-feature based approach: Wan et al. [22] compared the byte sequences in the executable files using Levenshtein distance similarity. The n-gram was used for feature extraction to obtain the training set required by the model. The similarity comparison allowed malware with similar behavior patterns to be close to each other, which can help n-gram to perform feature extraction more efficiently. Jung et al. [23] used images generated from malware byte information to represent malware behavioral context, and a convolutional neural network-based sentence analysis was used to process the generated images. They performed several experiments to show the efficiency of their proposed method, and the experimental results showed that their approach had higher accuracy than the naive CNN model. The detection accuracy was about 99%. Su et al. [24] proposed a novel lightweight approach for detecting DDOS malware in IoT environments. They extracted the malware images and utilized a lightweight convolutional neural network for detection. The experimental results showed that the proposed system could achieve 94.0% accuracy for detecting malware.

III. Methodology

This section describes the proposed IoT malware analysis method based on FCGs extracted from ELF binaries. As shown in Fig. 3, a given dataset is analyzed following three steps: reverse engineering, feature extraction, and classifier modeling. In reverse engineering, we used radare2 [25], a complete framework for reverse-engineering and analyzing binaries to create the FCGs. In feature extraction, we generate graph-embedding features by improving graph2vec and extract structural graph features introduced in [10], and then concatenate them into a numerical feature vector. In classifier modeling, we label the data with two categories: benignware or malware, and input the feature vector to classification algorithms. We use 5-fold cross-validation to evaluate the performance of the algorithms.

A. Reverse engineering

The proposed method analyzes ELF binary files of IoT malware. First, we use radare2 [25] to extract an FCG from a binary as input for subsequent analysis. As shown in Fig. 1, we can obtain the graphviz dot file for the binary using radare2 [25], and use the script to get the corresponding FCG. As shown in Fig. 2, FCG is a control flow graph that represents the calling relationship between subroutines in a computer program. Each node represents a function, and an edge (f, g) indicates that function f calls function g. Therefore, the loop in the figure represents a recursive procedure call [26].

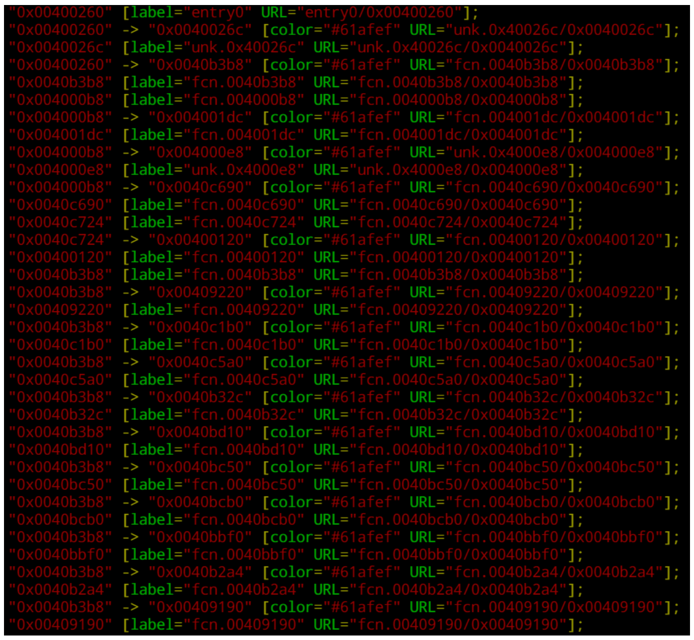
B. Feature extraction

We incorporate two types of features in the learning: graph embedding features and graph structure features.

Many previous works focused on representing the distributed representation of sub-structures in a graph to extract graph embedding features, such as nodes, subgraphs, etc. However, in the analysis tasks of knowledge graphs such as classification and clustering, we need to get the representation of the whole graph if existing means are used. The graph2vec [6] approach is effective in handling these analysis tasks. The main concept of graph2vec is to learn the graph representation by representing documents as graphs and words as rooted subgraphs. Graph2vect is usually implemented following two steps: WL machine and doc2vec.

Graph2vec learns the rooted-subgraph by WL relabeling based on the degree of the vertex. We consider that this approach will lose information about function calls, leading to reduced robustness in detecting malware. And an attacker can easily evade detection by changing the structure of the graph. Therefore, we made the following improvement in WL relabeling. Since the detection is based on FCGs, the vertices represent their functions. We use function names to replace degrees in WL relabeling to improve the robustness of the detector and make it more semantic.

To extract structural graph features, we make use of the statistical characteristics introduced in [9] to gain a more clear understanding of the structure of the overall graph. The graph



```
"0x00400260" [label="entry0" URL="entry0/0x00400260"];
"0x00400260" -> "0x0040026c" [color="#61afef" URL="unk.0x40026c/0x0040026c"];
"0x0040026c" [label="unk.0x40026c" URL="unk.0x40026c/0x0040026c"];
"0x00400260" -> "0x0040b3b8" [color="#61afef" URL="fcn.0040b3b8/0x0040b3b8"];
"0x0040b3b8" [label="fcn.0040b3b8" URL="fcn.0040b3b8/0x0040b3b8"];
"0x004000b8" [label="fcn.004000b8" URL="fcn.004000b8/0x004000b8"];
"0x004000b8" -> "0x004001dc" [color="#61afef" URL="fcn.004001dc/0x004001dc"];
"0x004001dc" [label="fcn.004001dc" URL="fcn.004001dc/0x004001dc"];
"0x004000b8" -> "0x004000e8" [color="#61afef" URL="unk.0x4000e8/0x004000e8"];
"0x004000e8" [label="unk.0x4000e8" URL="unk.0x4000e8/0x004000e8"];
"0x004000b8" -> "0x0040c690" [color="#61afef" URL="fcn.0040c690/0x0040c690"];
"0x0040c690" [label="fcn.0040c690" URL="fcn.0040c690/0x0040c690"];
"0x0040c724" [label="fcn.0040c724" URL="fcn.0040c724/0x0040c724"];
"0x0040c724" -> "0x00400120" [color="#61afef" URL="fcn.00400120/0x00400120"];
"0x00400120" [label="fcn.00400120" URL="fcn.00400120/0x00400120"];
"0x0040b3b8" [label="fcn.0040b3b8" URL="fcn.0040b3b8/0x0040b3b8"];
"0x0040b3b8" -> "0x00409220" [color="#61afef" URL="fcn.00409220/0x00409220"];
"0x00409220" [label="fcn.00409220" URL="fcn.00409220/0x00409220"];
"0x0040b3b8" -> "0x0040c1b0" [color="#61afef" URL="fcn.0040c1b0/0x0040c1b0"];
"0x0040c1b0" [label="fcn.0040c1b0" URL="fcn.0040c1b0/0x0040c1b0"];
"0x0040b3b8" -> "0x0040c5a0" [color="#61afef" URL="fcn.0040c5a0/0x0040c5a0"];
"0x0040c5a0" [label="fcn.0040c5a0" URL="fcn.0040c5a0/0x0040c5a0"];
"0x0040b3b8" -> "0x0040b32c" [color="#61afef" URL="fcn.0040b32c/0x0040b32c"];
"0x0040b32c" [label="fcn.0040b32c" URL="fcn.0040b32c/0x0040b32c"];
"0x0040b3b8" -> "0x0040bd10" [color="#61afef" URL="fcn.0040bd10/0x0040bd10"];
"0x0040bd10" [label="fcn.0040bd10" URL="fcn.0040bd10/0x0040bd10"];
"0x0040b3b8" -> "0x0040bc50" [color="#61afef" URL="fcn.0040bc50/0x0040bc50"];
"0x0040bc50" [label="fcn.0040bc50" URL="fcn.0040bc50/0x0040bc50"];
"0x0040b3b8" -> "0x0040bcb0" [color="#61afef" URL="fcn.0040bcb0/0x0040bcb0"];
"0x0040bcb0" [label="fcn.0040bcb0" URL="fcn.0040bcb0/0x0040bcb0"];
"0x0040b3b8" -> "0x0040bbf0" [color="#61afef" URL="fcn.0040bbf0/0x0040bbf0"];
"0x0040bbf0" [label="fcn.0040bbf0" URL="fcn.0040bbf0/0x0040bbf0"];
"0x0040b3b8" -> "0x0040b2a4" [color="#61afef" URL="fcn.0040b2a4/0x0040b2a4"];
"0x0040b2a4" [label="fcn.0040b2a4" URL="fcn.0040b2a4/0x0040b2a4"];
"0x0040b3b8" -> "0x00409190" [color="#61afef" URL="fcn.00409190/0x00409190"];
"0x00409190" [label="fcn.00409190" URL="fcn.00409190/0x00409190"];
```

Fig. 1. Function call graph graphviz dot in ELF binaries

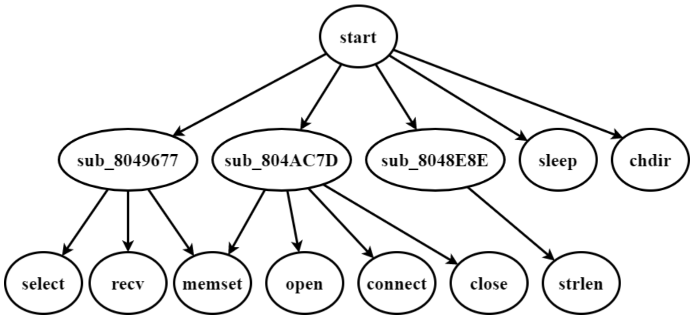


Fig. 2. A part of the function–call graph of a Linux.Mirai sample [11]

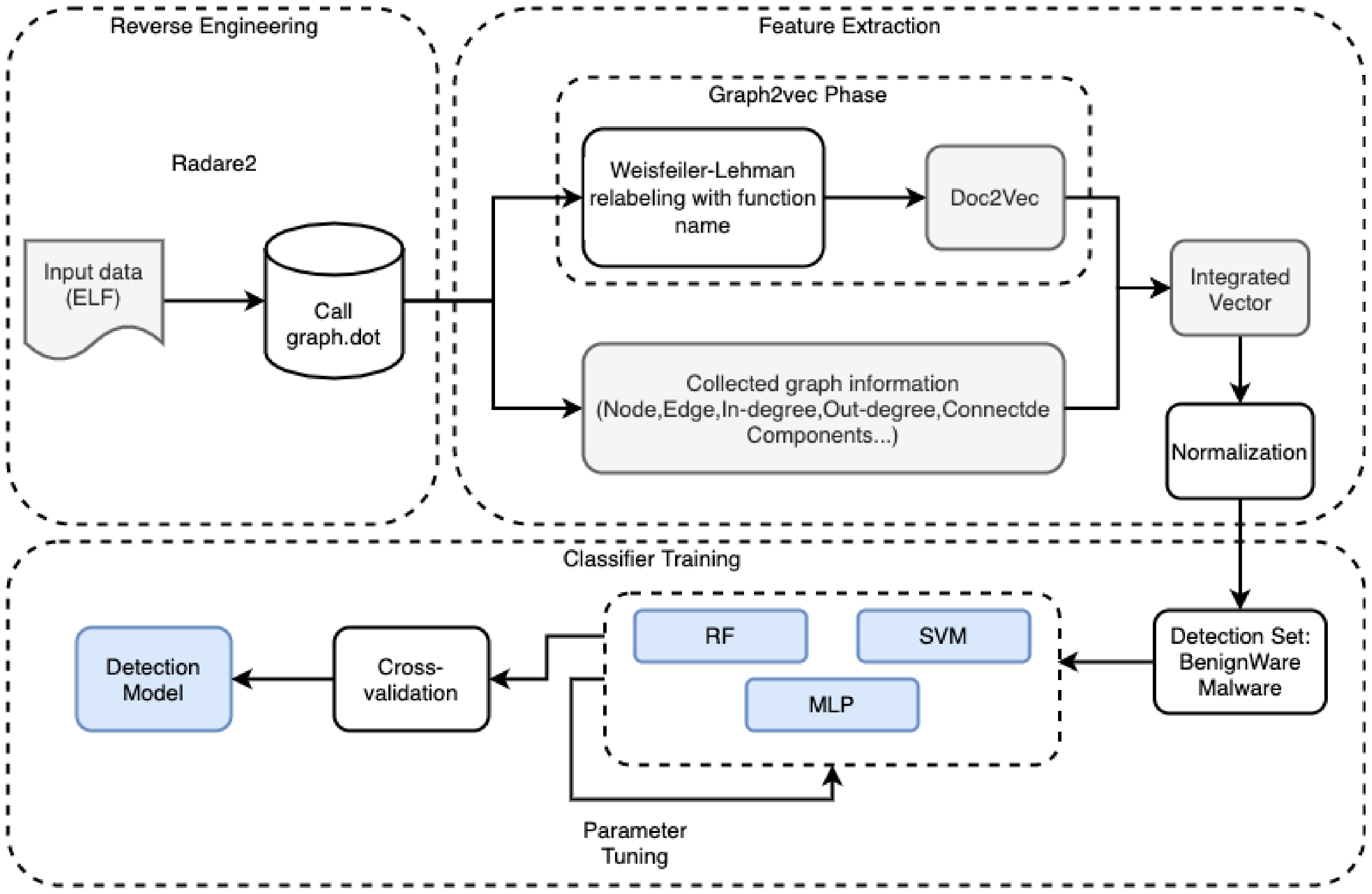


Fig. 3. Overview of the proposed IoT malware analysis framework

features we use include the number of nodes, number of edges, density, components, and file size. The measurements used in the analysis are defined as follows.

1) Definition 1 (**Numbers of nodes and edges**): General characteristics which can use to highlight the structural size of the binary.
2) Definition 2 (**Number of connected components**): In graph G, a connected component is a subgraph in which vertices are connected to each other, but not connected to other vertices in the subgraph. The number of components of G is the cardinality of a set that contains such components.
3) Definition 3 (**Density**): For a graph $G = (V, E)$, its density is defined as the closeness of all its edges to the maximum number of edges. The density of G can be represented as the average normalized degree. $Density = \frac{1}{n}\sum_{i=1}^{n}\frac{deg(v_i)}{n-1}$, where $V = \{v_1, v_2, ..., v_n\}$.

Finally, we combine graph embedding and structural graph features and apply a simple normalization to scale them all to the $[0, 1]$ range.

C. Classifier Training

Since the dimension of the Graph embedding feature may affect the subsequent experimental results, we first determine the training dimension before training.

Machine learning machine-learning-classifiers can automatically find rules by analyzing data and generalizing unknown malware in varying environments. We selected three commonly used classifiers, e.g., RF, SVM, and MLP, to build prediction models for detecting malware. The details are described as follows.

- RF is an ensemble learning method for classification or regression that operates by constructing a multitude of decision trees at training time [27]. For classification tasks, the output of RF is the class selected by most trees. For regression tasks, the mean or average prediction of the individual trees is returned.
- SVM is a supervised learning model that analyzes data for classification and regression analysis [28]. An SVM constructs a hyper-plane in the input or kernel-induced space following the large margin principle. It has been widely applied for classification, regression, or other tasks like outlier detection.
- MLP is a class of feedforward artificial neural networks consisting of at least three layers. It utilizes a optimization technique called back-propagation for training [29]. It can distinguish data that are not linearly separable.

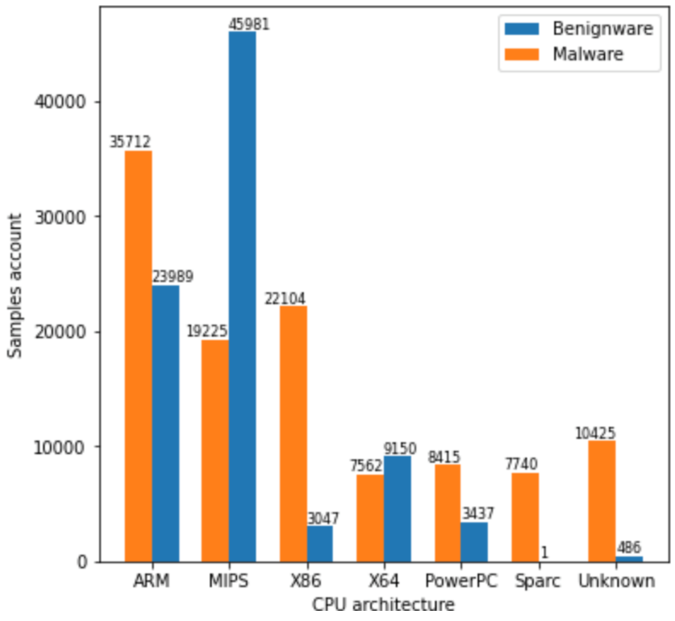


Fig. 4. Sample distribution among different CPU architectures

IV. Experiment

In our experiment, we utilized the dataset presented in Section IV-A to train the prediction model and tuned the parameters to obtain the best model. Finally, we evaluated the performances of RF, SVM, and MLP on IoT malware detection, aiming to differentiate malware and benignware in the dataset. Our experiments are based on the scikit-learn [30] implemented in Python 3.7.

A. Dataset

In the dataset, we collected $202,304$ ELF executable files from IoT devices, from which $112,728$ are malicious, and $89,576$ are benign. The distribution of samples among seven different CPU architectures is shown in Fig. 4. As shown in the figure, ARM and MIPS are the most popular CPU architectures. However, we failed to collect sufficient benignware of SPARC since most of the firmware images of SPARC are encrypted.

B. Visualization

To understand the distribution of the data, we use Uniform Manifold Approximation and Projection (UMAP) [31] to visualize the data in a two-dimensional embedding space. UMAP is a non-parametric graph-based dimensionality reduction algorithm that uses applied Riemannian geometry and algebraic topology to find low-dimensional embeddings of structured data. The UMAP algorithm consists of two steps: (1) computing the graphical representation of the data set (fuzzy simplicial complex) and (2) optimizing the low-dimensional embedding of the graph through stochastic gradient descent. It is computationally efficient and can deal with large high-dimensional datasets. Fig. 5 shows the 2D visualization result of a subset of 200 benignware samples and 200 malware

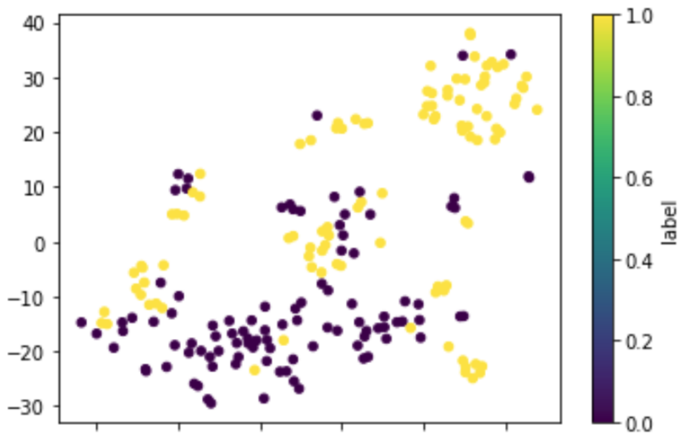


Fig. 5. A visualization of benignware and malware (Dimension=262)

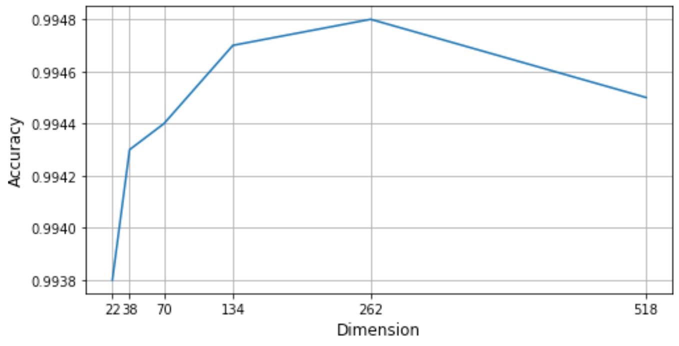


Fig. 6. Tuning parameter D using 5-fold cross validation for RF. D is selected from {16, 32, 64, 128, 256, 512}.

samples with dimension, D, is equal to 262. The figure shows yellow points representing malware sample and purple points representing benignware sample. We can see that despite the low-level presentation, the benignware and malware show a good separability: a benignware sample falls close to other benignware samples but apart from malware samples. High separability implies good prediction performance for malware detection.

C. Parameter tuning

The generalization performance of learning algorithms relies heavily on the hyperparameters used to train the model. Parameter tuning can give us optimized values for hyperparameters, which maximizes our model's predictive accuracy. For each of the parameters of a given model, we define a series of values, i.e., grid values, then all possible combinations of grid values on different parameters form a pool of parameters settings. Table I lists the grid values of all the parameters that we have examined for each classifier.

An example of parameter tuning is shown in Fig. 6. In the figure, the blue line describes the overall prediction accuracy averaged for each D parameter. We can observe that the accuracy rate increases as D increases from 22 to 262, and it decreases after $D = 262$. And we can get the best

TABLE I
SETTINGS FOR PARAMETER TUNING

Parameter	Classifier	Physical Meaning	Grid Values
D	RF, SVM, KNN	Parameter D as for Dimension	{16, 32, 64, 128, 256, 512}
N	RF	Number of trees in the forest	{50, 100, 150, 200, 250, 300, 350, 400, 450}
C	SVM	Penalty parameter	{0.1, 1, 10, 100, 1000}
γ	SVM	The width of the radial basis function kernel	{0.00001, 0.0001, 0.001, 0.01, 0.1}
S	MLP	Size of hidden layer	{10, 20, 30, ...,100}
I	MLP	Maximum number of epochs	{10, 20, 30, ...,100}

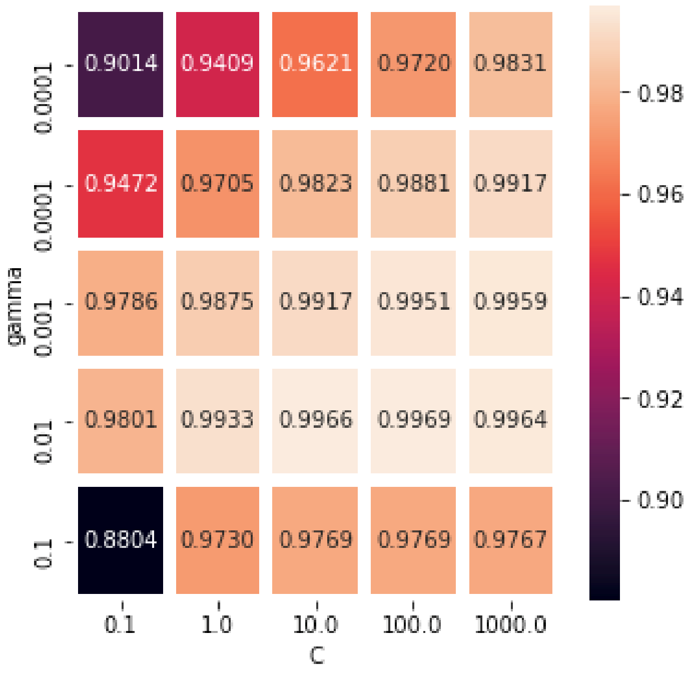


Fig. 7. Tuning parameter C and γ for SVM

accuracy rate at $D = 262$, so the result suggests that 262 is the most appropriate number of dimensions used in follow-up experiments. After selecting the most suitable dimension parameter, we adjust the parameters of the subsequent models accordingly.

The number of estimators, N, is tuned for RF. More estimators can make the model have better performance but at the same time make the prediction slower. We can achieve the highest accuracy when $N = 300$ and $N = 450$. As more estimators will lead to a longer training and prediction time, $N = 300$ is selected as the most suitable parameter for RFs.

As an example in Fig. 7, we tuned the C and γ parameters of the SVM. Here, C is called the penalty coefficient, which is the tolerance for errors. The higher the C, the less error will be tolerated, and the more likely to obtain an overfitting classifier. The smaller C is, the easier it is to obtain an underfitting model. If C is too large or too small, the generalization ability becomes poor. γ is a parameter that comes with the function after selecting the radial basis function (RBF) kernel. It implicitly determines the distribution of the data after mapping to the new feature space. Commonly, the larger the γ, the fewer the support vectors, and the smaller the γ, the more support vectors. The number of support vectors affects the speed of training and prediction. We used grid search to make (C, γ) independent of each other, convenient for a parallel implementation to get the best overall result. The grid search table shows that the best result can be obtained when $C = 100$ and $\gamma = 0.01$. We select $(100, 0.01)$ as the parameter setting for SVMs.

Finally, we tuned the hidden layer size, S, and $max\_iteration$, I, for the MLP. In a neural network, the number of hidden layer nodes and the iterations are tuned, greatly affecting the generalization performance. S determines the model complexity of the network, and $max\_iteration$ controls the overfit to training samples. The grid search table shows that the best result is obtained at $S = 100$, $max\_iters = 80$. The reported experiment result for MLPs is based on this parameter setting.

D. Evaluation matrices

In the evaluation phase, we adopted the common evaluation metrics, namely, accuracy, recall, precision, false-positive rate, and F1-measure, to assess the performance of our proposed method. These metrics are defined based on the following intermediate measures.

- True positive (TP): samples correctly classified as positive.
- False positive (FP): samples incorrectly classified as positive.
- True negative (TN): samples correctly classified as negative.
- False negative (FN): samples incorrectly classified as positive.

Accuracy refers to the proportion of correct judgments of true and false:

$$Accuracy = \frac{TP + TN}{n}. \quad (1)$$

Precision refers to how much is true when the judgment is true:

$$Precision = \frac{TP}{TP + FP}. \quad (2)$$

Recall is the probability of the samples in the positive class being classified correctly:

$$Recall = \frac{TP}{TP + FN}. \quad (3)$$

F1-measure is the weighted average of precision and recall:

$$F1\_measure = \frac{2 * (Recall * Precision)}{Recall + Precision}. \quad (4)$$

E. Performance evaluation

In this section, we design two experiments to demonstrate the performance of the proposed approach for IoT malware detection. The first experiment is conducted on an overall dataset consisting of all of the samples in our dataset. In the second experiment, the evaluation is performed on eight subsets formed by selecting samples from the same CPU architecture. Following section IV-C, we get the optimal parameters for each classifier independently. The selected parameter settings are shown in the rightmost column in Table II, and we use 5-fold cross-validation to verify our experimental results. The results in tables are averaged over five independent runs with the training and test sets determined by 5-fold cross-validation.

The experiment's purpose is to evaluate the performance of the proposed method in distinguishing malware samples and benignware samples. A detector will issue a security alert to warn the user when a downloaded program is recognized as malware. Table II shows the evaluation results of all examined classifiers. All classifiers yield near-optimal results with accuracy greater than 99%, indicating that the FCGs with graph2vec and structure information carry essential information for differentiating malware from benignware on all IoT platforms.

Fig. 6 shows how dimension affects the performance of the classifier. The best accuracy is obtained at $D = 262$, and the accuracy drops when D falls far from this value. In this task, because of the near-optimal performance, precision and recall on the malware class are close to the accuracy values, shown as overlapped lines in the figure.

We have also compared the proposed method with graph2vec. From Table II, we can find that the performance of the proposed method has been significantly improved, so it has also been certified that our approach can be more comprehensive in detecting malware.

In this experiment, we divided the dataset into seven subsets based on the supported CPU architectures of the samples. Since the benignware samples on SPARC are insufficient to form a proper dataset, we omit SPARC experiments.

Fig. 8 shows the distribution of data embedded in the 2D space, with each unique color representing a different CPU architecture. To improve the readability of the graph, we use a subset of the $10,000$ samples for visualization purposes. We have confirmed that using UMAP on the full dataset produces a layout similar to the one shown in the figure. We can see that the samples from the same CPU architecture form tight clusters. The benign and malware samples within the clusters, denoted by colored discs and minus signs, respectively, exhibit significant separability. This result suggests that it may be feasible to pursue CPU-specific classification.

Table III shows the results of malware detection among specific CPU architectures. We only report SVM and MLP, which can get better performance within the overall dataset in the table to enhance readability.

As shown in Table III, SVM and MLP reach accuracy rates close or even above 99.70% on all CPU architectures except x86_64. Performance of malware detection on x86_64 is always slightly lower than on other platforms. An explanation for this result is that the binaries collected for the x86_64 architecture include files from a wide range of Linux distributions, rendering the classification tasks more complicated.

We also compare the results of the two scenarios in Table

TABLE II
PERFORMANCE COMPARISON OF GRAPH2VEC AND PROPOSED METHOD

Classifier		Graph2vec	Proposed method	Parameter
RF	Accuracy	98.64%	99.48%	D=256,
	F1-score	98.77%	99.54%	N=300
SVM	Accuracy	98.91%	**99.71%**	D=256,
	F1-score	99.03%	**99.74%**	C=100, γ=0.01
MLP	Accuracy	98.45%	99.70%	D=256,
	F1-score	98.62%	99.73%	S=100, I=80

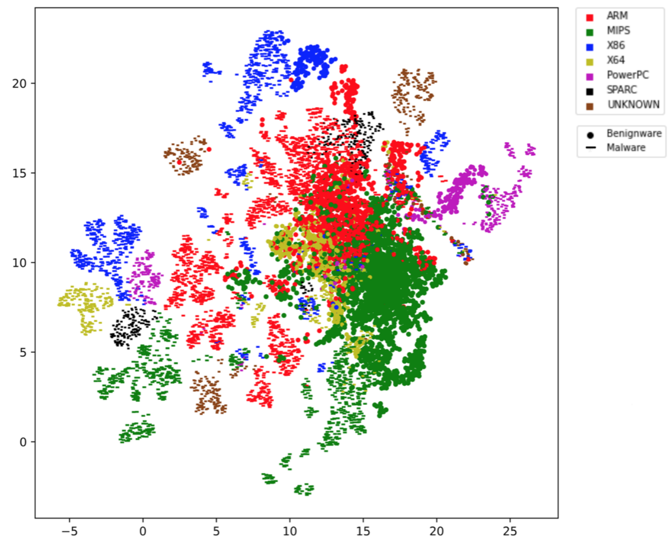


Fig. 8. UMAP 2D visualization of sample distribution for benignware and malware from different CPU architectures.

TABLE III
RESULT OF CPU-SPECIFIC ANALYSIS

CPU Architecture	Classifier	Accuracy	F1-score
ARM	SVM	99.69%	99.68%
	MLP	**99.71**%	**99.70**%
MIPS	SVM	**99.81**%	**99.78**%
	MLP	99.80%	99.75%
X86	SVM	**99.99**%	**99.97**%
	MLP	99.96%	99.91%
X86-64	SVM	99.52%	99.52%
	MLP	**99.68**%	**99.68**%
PowerPC	SVM	99.80%	99.76%
	MLP	**99.89**%	**99.86**%
SPARC	SVM	NaN	NaN
	MLP	NaN	NaN
UNKNOWN	SVM	**99.73**%	**98.35**%
	MLP	99.63%	97.86%

TABLE IV
RESULT COMPARISON IN TWO SCENARIOS

	All Data		Divide CPU	
Classifier	SVM	MLP	SVM	MLP
Accuracy	**99.72**%	99.70%	99.76%	**99.77**%
F1-score	99.71%	**99.73**%	**99.68**%	99.66%

IV. Numbers in the table are the weighted averages of the numbers reported in Table IV according to:

$$WeightedMetric = \frac{1}{M}\sum_{i=0}^{C} m_i * Metric(i), \quad (5)$$

where C is the number of CPU architectures in the experiment, M is the total number of samples in the dataset, and m_i is the number of samples on the i-th CPU architecture.

These results show that CPU-specific analysis can further improve the generalization performance of the classifiers.

V. DISCUSSION

A. Comparison with related work

In this experiment, we compare the proposed method with the methods in the related work of malware detection. We implemented the FCG signature-based method introduced by Vij et al. [32] and the method proposed by Alasmary et al. [9] which uses graph theory information extracted from control flow graph for malware detection. Due to scalability issues with the referred method, we used a subset of 100K ELF files composed of 50K malware and 50K benignware samples for evaluation.

TABLE V
PERFORMANCE COMPARISON WITH RELATED WORK

Research	Feature	Classifier	Accuracy	F1-score
Vij [32]	FCG signature	RF	96.05%	96.04%
		SVM	74.00%	73.96%
		MLP	81.13%	81.13%
Alasmary [9]	CFG graph theory	RF	94.14%	94.13%
		SVM	77.34%	77.34%
		MLP	80.97%	80.41%
Proposed method	FCG embedding	RF	99.17%	99.17%
		SVM	99.37%	99.37%
		MLP	**99.59**%	**99.58**%

The results of each classifier are shown in Table V. The proposed method has the highest accuracy in malware detection (99.59% using MLP). These results indicate that FCG embedding contains considerable discriminating information for malware detection, and these results show that the proposed method is more efficient than existing methods.

B. Limitation of proposed method

Since our proposed method detects malware based on static analysis, it is unavoidable if an attacker deliberately avoids detection. If the attacker adds more obfuscating functions to change the overall structure of the binaries, it may cause error prediction for the detector. This is an inherent difficulty for all static analysis methods, and we plan to further improve on this point and strengthen the stability of our detector.

VI. CONCLUSION

In this paper, we proposed an approach based on FCG to optimize the effectiveness of graph2vec in IoT malware detection and combined it with structural graph features to make the detection more accurate. The experimental results on malware detection showed near-optimal accuracy greater than 99.72%, and the performance can be further improved using CPU-specific models. We believe that promising solutions for IoT security can be developed based on these findings.

ACKNOWLEDGEMENT

This work is supported in part by Ministry of Science and Technology, Taiwan, under contract numbers 110-2218-E-011-004-MBK.

REFERENCES

[1] M. Wazzan, D. Algazzawi, O. Bamasaq, A. Albeshri, and L. Cheng, "Internet of Things botnet detection approaches: Analysis and recommendations for future research," *Applied Sciences*, vol. 11, no. 12, p. 5713, 2021.

[2] H. S. Galal, Y. B. Mahdy, and M. A. Atiea, "Behavior-based features model for malware detection," *Journal*

of Computer Virology and Hacking Techniques, vol. 12, no. 2, pp. 59–67, Dec 2016.

[3] W. Li, Z. Su, K. Zhang, A. Benslimane, and D. Fang, “Defending malicious check-in using big data analysis of indoor positioning system: An access point selection approach,” *IEEE Transactions on Network Science and Engineering*, vol. 7, no. 4, pp. 2642–2655, Aug. 2020.

[4] Y.-T. Lee, T. Ban, T.-L. Wan, S.-M. Cheng, R. Isawa, T. Takahashi, and D. Inoue, “Cross platform IoT-malware family classification based on printable strings,” in *Proc. IEEE TrustCom 2020*, 2020, pp. 775–784.

[5] Q.-D. Ngo, H.-T. Nguyen, H.-A. Tran, and D.-H. Nguyen, “IoT botnet detection based on the integration of static and dynamic vector features,” in *Proc. IEEE ICCE 2020*, 2021, pp. 540–545.

[6] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, “graph2vec: Learning distributed representations of graphs,” *arXiv preprint arXiv:1707.05005*, Jul. 2017.

[7] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, “Weisfeiler-lehman graph kernels.” *Journal of Machine Learning Research*, vol. 12, no. 9, 2011.

[8] Q. Le and T. Mikolov, “Distributed representations of sentences and documents,” in *International conference on machine learning*, 2014, pp. 1188–1196.

[9] H. Alasmary, A. Khormali, A. Anwar, J. Park, J. Choi, A. Abusnaina, A. Awad, D. Nyang, and A. Mohaisen, “Analyzing and detecting emerging Internet of Things malware: A graph-based approach,” *IEEE IoT-J*, vol. 6, no. 5, pp. 8977–8988, July 2019.

[10] Y. Zhang, X. Chang, Y. Lin, J. Mišić, and V. B. Mišić, “Exploring function call graph vectorization and file statistical features in malicious PE file classification,” *IEEE Access*, vol. 8, pp. 44 652–44 660, Mar 2020.

[11] H.-T. Nguyen, Q.-D. Ngo, and V.-H. Le, “A novel graph-based approach for IoT botnet detection,” *IJISS*, vol. 19, no. 5, pp. 567–577, Oct 2020.

[12] K. Vinayaka and C. Jaidhar, “Android malware detection using function call graph with graph convolutional networks,” in *Proc. ICSCCC 2021*, May 2021, pp. 279–287.

[13] A. Marzano, D. Alexander, O. Fonseca, E. Fazzion, C. Hoepers, K. Steding-Jessen, M. H. Chaves, Í. Cunha, D. Guedes, and W. Meira, “The evolution of bashlite and mirai iot botnets,” in *Proc. IEEE ISCC 2018*, Jun. 2018, pp. 00 813–00 818.

[14] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, and M. Kallitsis, “Understanding the Mirai botnet,” in *Proc. USENIX 2017*, Aug. 2017, pp. 1093–1110.

[15] M. J. Bohio, “Analyzing a backdoor/bot for the mips platform,” *Analysis of a MIPS Malware/SANS Institute InfoSec Reading Room*, 2015.

[16] S. Hou, A. Saas, L. Chen, and Y. Ye, “Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs,” in *Proc. IEEE/WIC/ACM WIW 2016*, Jan. 2016, pp. 104–111.

[17] R. Surendran, T. Thomas, and S. Emmanuel, “Gsdroid: Graph signal based compact feature representation for android malware detection,” *Expert Systems with Applications*, vol. 159, p. 113581, Nov. 2020.

[18] E. Amer and I. Zelinka, “A dynamic windows malware detection and prediction method based on contextual understanding of API call sequence,” *Computers & Security*, vol. 92, p. 101760, May 2020.

[19] H.-T. Nguyen, Q.-D. Ngo, D.-H. Nguyen, and V.-H. Le, “Psi-rooted subgraph: A novel feature for IoT botnet detection using classifier algorithms,” *ICT Express*, vol. 6, no. 2, pp. 128–138, Jun. 2020.

[20] F. Xiao, Y. Sun, D. Du, X. Li, and M. Luo, “A novel malware classification method based on crucial behavior,” *Mathematical Problems in Engineering*, Mar. 2020.

[21] R. Kawasoe, C. Han, R. Isawa, T. Takahashi, and J. Takeuchi, “Investigating behavioral differences between IoT malware via function call sequence graphs,” in *Proc. ACM SAC 2021*, Mar. 2021, pp. 1674–1682.

[22] T.-L. Wan, T. Ban, Y.-T. Lee, S.-M. Cheng, R. Isawa, T. Takahashi, and D. Inoue, “IoT-malware detection based on byte sequences of executable files,” in *Proc. AsiaJCIS 2020*, Aug. 2020, pp. 143–150.

[23] B. Jung, T. Kim, and E. G. Im, “Malware classification using byte sequence information,” in *Proc. RACS 2018*, Oct. 2018, pp. 143–148.

[24] J. Su, D. V. Vasconcellos, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai, “Lightweight classification of IoT malware based on image recognition,” in *Proc. IEEE COMPSAC 2018*, vol. 2, Jun. 2018, pp. 664–669.

[25] R. Team, “Radare2 github repository,” https://github.com/radare/radare2, 2017.

[26] Wikipedia, “Call graph.” [Online]. Available: https://en.wikipedia.org/wiki/Call_graph

[27] Wikipedia, “Random forest.” [Online]. Available: https://en.wikipedia.org/wiki/Random_forest

[28] Wikipedia, “Support-vector machine.” [Online]. Available: https://en.wikipedia.org/wiki/Support-vector_machine

[29] ——, “Multilayer perceptron.” [Online]. Available: https://en.wikipedia.org/wiki/Multilayer_perceptron

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, “Scikit-learn: Machine learning in python,” *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[31] L. McInnes, J. Healy, and J. Melville, “Umap: Uniform manifold approximation and projection for dimension reduction,” *arXiv preprint arXiv:1802.03426*, Feb. 2018.

[32] D. Vij, V. Balachandran, T. Thomas, and R. Surendran, “Gramac: A graph based android malware classification mechanism,” in *ACM CODASPY 2020*, Mar. 2020, pp. 156–158.