HawkEye: Cross-Platform Malware Detection with Representation Learning on Graphs

Peng Xu[1], Youyi Zhang[2], Claudia Eckert[1], and Apostolis Zarras[3]

[1] Technical University of Munich
[2] Tongji University
[3] Delft University of Technology

Abstract. Malicious software, widely known as malware, is one of the biggest threats to our interconnected society. Cybercriminals can utilize malware to carry out their nefarious tasks. To address this issue, analysts have developed systems that can prevent malware from successfully infecting a machine. Unfortunately, these systems come with two significant limitations. First, they frequently target one specific platform/architecture, and thus, they cannot be ubiquitous. Second, code obfuscation techniques used by malware authors can negatively influence their performance. In this paper, we design and implement `HawkEye`, a control-flow-graph-based cross-platform malware detection system, to tackle the problems mentioned above. In more detail, `HawkEye` utilizes a graph neural network to convert the control flow graphs of executable to vectors with the trainable instruction embedding and then uses a machine-learning-based classifier to create a malware detection system. We evaluate `HawkEye` by testing real samples on different platforms and operating systems, including Linux (x86, x64, and ARM-32), Windows (x86 and x64), and Android. The results outperform most of the existing works with an accuracy of 96.82% on Linux, 93.39% on Windows, and 99.6% on Android. To the best of our knowledge, `HawkEye` is the first approach to consider graph neural networks in the malware detection field, utilizing natural language processing.

1 Introduction

With the development of 5G and IoT networks, as well as autonomous driving, Linux-based devices are becoming ubiquitous around the world. Meanwhile, the malicious software targeting these systems increasingly attracts the attention of both academia and industry, especially for Linux-based IoT devices and cloud servers. Historically, the security community concentrates on detecting and analyzing malware samples that primarily target Windows-based systems. However, this has changed as embedded systems and cloud servers rely on various architectures and operating systems. Unfortunately, most of the existing malware detection systems target a single platform and cannot recognize the cross-platforms' malware. Take modern ICT environments as an example; it is common to find a mixture of different operating systems on servers (e.g., Linux) and workstations

(e.g., Windows). Without anti-malware products for all the different operating systems, malware can easily infiltrate an organization's premice [12]. As a matter of fact, more and more malware is targeting cross-platforms [5]. Additionally, the native libraries of Android apps, including the malicious native code, are cross-platform (i.e., `x86`, `ARMv7`, `ARMv8`).

There are currently numerous malicious code detection methods using machine learning to characterize and discover the malicious behavior patterns of malware. Although these methods serve the target platforms with an extra security layer, nearly all of them suffer cross-platform issues because they leverage specific features to target a distinct type of malware. For example, the permission-based Android malware detection system cannot detect malicious Windows PE examples. For instance, the Windows Richer Header [14] feature cannot detect Linux ELF malware and Android's DEX files.

To address cross-platform issues in general, Control Flow Graphs (CFG) based method is a solution in the right direction because all programs have CFGs, which makes this approach platform-independent. However, the existing CFG-based methods [4, 16] have two inevitable drawbacks. On the one hand, these approaches are far from scalable because of the expensive graph matching computation and sub-graph isomorphism. These techniques conduct pairwise graph matching for malware search, the complexity of which makes them unusable in large-scale datasets. On the other hand, the fixed graph pattern [4,16] is hard to adapt to different code because of too many manually fixed features.

In this paper, we design and implement `HawkEye`, a cross-platform malware detection framework to tackle these problems. `HawkEye` is based on a hybrid Control Flow Graph and Graph Neural Networks and is inspired by Natural Language Processing. `HawkEye` includes three primary components: (*i*) a graph generator, which extracts both static and dynamic control flow graphs from executable files; (*ii*) a graph-neural-network-based graph embedding method responsible for converting the whole attributed graph to a unique vector; (*iii*) a machine-learning classification system able to classify malware samples. Overall, this framework not only can be used by various platforms but also outperforms numerous malware detection solutions.

In summary, we make the following main contributions:

- We develop a cross-platform malware detection framework, which can detect not only Windows-based malware but also Linux and Android malware. `HawkEye` currently supports Intel, ARM, and MIPS architectures.
- We implement a representation-learning-based feature engineering-based on graph neural networks capable of identifying malware samples. To the best of our knowledge, `HawkEye` is the first approach to leverage graph neural networks (GNN) for cross-platform malware detection with the help of trainable features and transform learning.[4]

[4] Although Devign [17] uses GNN in the cybersecurity field, it only targets the `C` source code of famous CVE.

- We introduce instruction, basic-block, and graph embedding to assist out feature engineering and convert the program code to vectors,
- We evaluate `HawkEye` with real-world applications and various metrics. For Windows and Android malware detection, the results outperform most of the existing works. For Linux, we also retrieve significant results.

2 Related Work

MalConv [10] models the execution sequences of disassembled malicious binaries. It implements a neural network that consists of convolutional and feedforward neural constructs. That architecture embodies a hierarchical feature extraction approach that combines the convolution of n-grams of instructions with plain vectorization of features derived from the *Portable Executable* (PE) files' headers.

Ember [3] presents an open dataset for training static PE malware machine learning models. It extracts eight groups of raw features that include parsed features, format-agnostic histograms, and counts of strings. Those features include features extracting from the header file, imported functions table, exported functions table, raw byte histogram, and string information.

Adagio [4] implements a kernel-hashing-based malware detection system on the function call graph. It is based on the efficient embeddings of function call graphs with an explicit feature map inspired by a linear-time graph kernel. In an evaluation with real malware samples purely based on structural features, Adagio outperforms several related approaches. MAGIN [16], on the other hand, takes the manually fixed 11 features from the attributed CFG, which is the same with Gemini [15].

3 Motivation

3.1 Cross-platform Malware Detection

On the one hand, the reason why we need cross-platform malware detection is primarily stimulated by the increasing native library-based Android malware. Furthermore, those native libraries are not only targeting the ARM32/ARM64 CPUs but also Intel X86/X64 CPUs. So far, most Android malware detection works are only concentrating on Android byte-code (DEX files), especially for the static code analysis based malware detection. Although few works consider the native library-based malware, no work so far takes care of the malicious code introducing by various version native libraries (i.e., for Intel and ARM platforms).

On the other hand, although it seems unimaginable to design a malware detection system targetting not only to Windows platform but also Linux and macOS, there are more and more examples illustrating that the same malicious code or vulnerability affects not only Windows but also Linux Users [5]. Even in cases with similar methods to detect malware on different platforms, it is nearly impossible to adopt one approach directly from one platform to another due

Table 1: Comparsion with previous works

	Approaches	Description
CFG	Gemini [15], MAGIN [16]	*ACFG + Manually indicated features*
	Adagio [4]	*CFG + Manual one-hot embedding features*
Byte Sequences	MalConv [10]	*Convolutional + trainable embedding*
	Ember [3]	*Gradient boosted decision tree + LightGBM*
CFG+NLP	Pektaş et. al. [9]	*Malware Detection + call graph + graph embedding*
	`HawkEye`	*Cross-platforms + Representation learning on graph*

to some differences. For example, the API-based malware detection systems [9] are tightly associated with the underlying operating system. It means that Windows, Linux, Android, and macOS operating systems have significantly different underlying support for those detection systems.

Therefore, a malware detection system for various operating systems is a new topic in the cybersecurity field. To detect malware for cross-platform CFGs are considered a fundamental feature since they are platform-independent and thus are proper to represent a program behavior [4, 16].

3.2 Representation Learning based Feature Engineering

To generate the node attribute in the control flow graph, we take representation learning into consideration. Representation learning, which can automatically learn features from raw data, has increasingly attracted researchers' and engineers' focus. Compared to those manually indicated attributed control flow graph (ACFG) methods, like Xu et al. [15], Adagio [4], and Yan et al. [16], `HawkEye` can extract ACFG automatically without preparing manual features and avoiding the challenge of manual indicated methods (how to pick up the useful features is a challenge) because of the representation learning. For example, to represent the vertex of ACFG, Xu et al. [15] manually indicated six block-level attributes (numbers of calls, instructions, arithmetic instruction and transform instruction, string constants, and number of constants) and two inter-block attributes (numbers of offspring and betweenness). Additionally, `HawkEye` borrows ideas from natural language processing (NLP) to assist feature engineering. `HawkEye` uses the word2vec to convert instructions to vectors. Although MalConv [10] and Ember [3] also take benefits from NLP and leverage the n-gram method to extract features, n-gram also loses the bag-to-word information.

In brief, `HawkEye` introduces representation learning as the fundamental technique to represent code, which is different compared to [4, 15, 16], and use the control flow graph as fundamental to organize the program, which is different with those target-specific methods [9,10]. Additionally, it utilizes NLP to convert the byte sequences (instruction and basic block) to vectors, used to replace the manually indicated features [4,15,16]. The differences between `HawkEye` and the current works are summarized in Table 1.

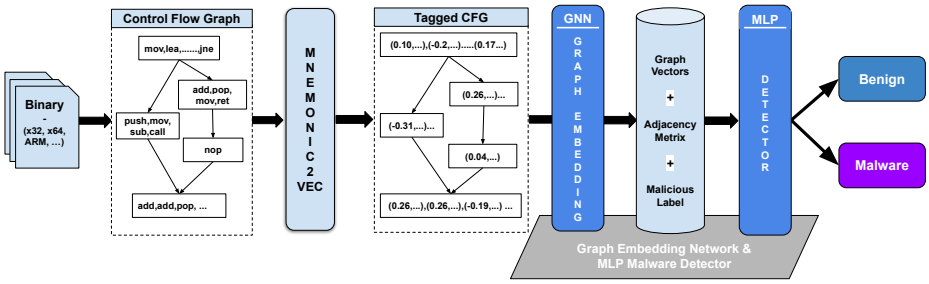


Fig. 1: System architecture

4 System Design

4.1 Architecture

We formalize our cross-platform malware detection system as a binary classification problem. We define the graph sets, which represents the structural information of the executable files (malware and benign), as our input, like $G^D(V^D, E^D)$, where V^D presents the set of graph's nodes, and E^D presents the edges among those nodes and D presents the number of the graph. For each graph g_i, it is encoded as $g_i(V, X, A)$, where $A \in \{0,1\}^{m*m}$ is the adjacency matrix and m is the number of the graph's node, $m = |V|$. $X \in R^{m*d}$ presents a d-dimensional real-valued vector $x_j \in R^d$.

$$loss = \min \sum_{i=1}^{n} \lambda(f(g_i(V, X, A)) + \delta w(f)) - y_i \tag{1}$$

where $i \in n$ and $n = |D|$. The goal of `HawkEye` is to learn a mapping from G^D to Y^D, $f : G \to Y$ to predict whether an executable file is malicious or not. The prediction function f can be learned by minimizing the loss function in Equation 1, where g is the graph embedding and f is the MLP-classifier. In our work, we also add one $w(*)$ function to adjust the stability (reduces the influence of the difference from graph size) of `HawkEye`, and δ is a coefficient to scale the function w. Figure 1 illustrates `HawkEye`'s architecture. The executable binary is the framework's input, and the prediction label (malware or benign) is the output. In summary, it includes three primary modules:graph generator, Feature embedding and MLP-based classifier as our malware detector.

4.2 Graph Generator

Based on the Angr framework (including angr-utils and bingraphvis) [11], we build our own *CFG Generator* to extract flow graphs from the executable binaries. The extracted graphs from cross-platform binaries include the static and dynamic CFG as well as the fCG (function call graph). For the static CFG, `HawkEye` constructs a directed graph $G = (N, E)$ with the basic block addresses

and assembly code inside of basic blocks, where N represents all the nodes of the graph, including the node name and content (assembly instructions), and E represents the set of edges of the graph. The dynamic CFG and the fCG can be considered as a reduced graph of the static CFG. The dynamic CFG only concludes the basic blocks covered by symbolic execution. The fCG, on the other hand, collects the assembly code at the function level.

4.3 Feature Embedding

This component aims to generate a finite-dimensional nonlinear vector for each instruction in basic blocks. We divide this task into three modules: (*i*) mnemonic (opcode) embedding, (*ii*) block embedding, and (*iii*) CFG embedding.

Mnemonic Embedding. All the assembly instructions usually come with the following form: *label:[mnemonics][operands][comment]*. The label is an identifier that is a position marker for instructions and data. The instruction mnemonic is a brief word that marks an instruction, such as *mov*, *add*, and *sub*. The operands represent the value of instruction control and transfer. We only consider the mnemonic part because those mnemonics are linked to behaviors; the operands are the corresponding behaviors objects. As the final operands usually depend on the immediate number, register, and memory, we cannot use them as a feature because of their inherent variability. Therefore, we concentrate on analyzing the sequence of mnemonics. We show the difference between mnemonic and instruction embedding in the evaluation. There exist numerous methods to convert the sequence mnemonics into a vector sequence. In this work, we adjust the skip-gram sampling model of *word2vec* [6] to reduce word to the opcode of instruction to implement the mnemonic embedding.

Block Embedding. The embedding of a basic block is derived from all the instructions contained in the embedded block. The method for producing block embedding is *normalization*. Then, the normalization method is represented as: $x_{normalization} = \frac{x-Min}{Max-Min}$, where Max and Min are the maximal and minimal values among all embedding vectors in the basic block. The first element of vectors is picked up to get the maximal and minimal values. The list of block embedding vectors is forwarded to our adapted graph embedding network and malware detector for training.

Graph Embedding. To calculate the graph embedding from embedding vectors of the generated tagged CFG, we combine the graph structure characteristics and the node features with a settable iteration size (one Hyperparameter of GNN). For the graph embedding, in our case, the vertices (nodes) of a graph are functions, and the edges are connections among those functions. Those vertices (nodes) contain a set of opcodes inside them. The function embedding constructs each node's feature. In essence, we apply the graph embedding network based on *structure2vec* to convert the vectors of one graph to a unique vector. This neural network mainly considers two aspects of information: the instruction sequence in the node and the connection between the basic blocks. Our GNN combines

Table 2: The number of samples in different datasets

Platforms	Training		Validation		Testing		Total	
	Malware	Benign	Malware	Benign	Malware	Benign	Malware	Benign
Windows-x86	17,910	19,043	5,970	6,347	5,970	6,346	29,850	31,736
Android	15,331	15,000	5,111	5,000	5,111	5,000	25,553	25,000
Linux-x86	5,501	5,693	1,834	1,898	1834	1,897	9,169	9,488
Linux-x64	319	923	106	307	106	307	533	1,539
Linux-ARM32	434	446	144	148	144	148	724	744

these two kinds of information to generate our graph embedding vectors through deep neural structures. The realization of graph embedding generation is the execution of a graph neural network with unsupervised feature learning. So far, `HawkEye` can transform the input data into a group of graph embedding vectors containing feature information.

4.4 MLP-based Malware Classifier

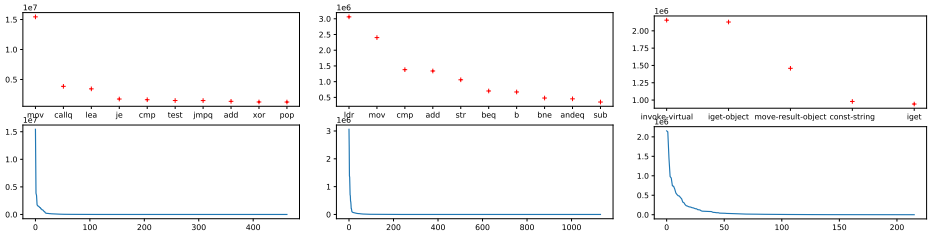
In this step, we establish a machine learning model that distinguishes malware from benign executables following the generated graph embedding vectors. The trained model with the best accuracy is stored as the final malware detection model. We use a multi-layer perceptron (MLP) for classification tasks. More specifically, we use one input layer, two hidden layers with 32 units, and one output layer in our MLP. We take the hinge loss as our loss function, which determines the difference between raw output prediction value $< P = MLP(X) >$ and real value $< R = Y\_label >$.

5 Evaluation

5.1 Dataset and Experimental Setup

Dataset Preparation. We collected our malware samples from VirusShare [1] and AndroZoo [2]. For benign samples, we collected ELF binary samples from libraries and executable files from Ubuntu 18.04-x64, Ubuntu {14.04, 16.04, 18.04}-x86 on Intel, and Raspbian 32-bit on ARM. For Windows-x86, we collected the benign binary samples from Windows XP, 7, 8, and 10. For Android samples, we collected all the samples from AndroZoo and used VirusTotal [13] API to label malicious samples classified as malware by more than five frameworks in VirusTotal. The samples in the training dataset account for 60% of the total binaries, while the samples in the validation dataset account for 20%. Finally, the samples in the testing dataset account for 20% of the total binaries. The detailed statistics are shown in Table 2.

Learning Setup. In our implementation, for the graph embedding network, we selected minibatch as 1, which is analogous to an online learning model. The reason behind our choice is that the input data contains the adjacency matrix,



(a) Intel X86 (b) ARM (c) Android-Dalvik

Fig. 2: Power-law Distribution for Intel, ARM and Dalivk Opcodes.

which consumes a significant amount of memory, especially some APKs have more than 90,000 nodes (basic blocks of control flow graph). The classifier takes an input feature size of 32. The supervised classifier takes a loss value among the mean absolute error, sigmoid cross-entropy, and hinge loss to compare the rate of convergence using the ADAM [7] optimization algorithm with a 0.04 learning rate over 10 to 25 epochs. To avoid over-fitting, we use the model parameters at the minimum validation loss as the final learned weighted matrix θ. We execute the learning process for diverse situations on a Tensorflow computation framework on a server equipped with AMD EPYC Processor (64 cores) and 128 GB DDR4 memory RAM.

5.2 Power Law and Opcode Embedding

Before moving to our evaluation tasks, we use power-law distribution to prove the reasonability of using natural language processing techniques in our work. Figure 2 presents the opcode distribution for various platforms with the above datasets. Figure 2(a) shows the opcode distribution for Intel X86/X64 (we consider 32-bit and 64-bit opcode together since 96% of them are overlapped). In total, we have 463 opcodes; the 10 largest usage opcodes are illustrated. Figure 2(b) shows the ARM's opcode and its distribution. We have 1,131 ARM-opcode in total; the top 10 opcodes are shown. Meanwhile, Figure 2(c) shows Dalvik's opcode distribution, which has 216 opcodes; the top 5 opcodes are presented. All of them follow the power-law distribution, which means borrowing word embedding techniques from natural language processing to do opcode embedding is reasonable.

5.3 Evaluation Tasks

After determining the experimental setup, we evaluate `HawkEye`'s performance on the following tasks: malware detection performance, CFG generation, and training efficiency.

We tested our model's accuracy using the standard *Area Under Curve - Receiver Operating Characteristic* (AUC-ROC) curve. We utilize four metrics:

Table 3: Performance comparison with other approaches

Model	Accuracy (%)	Precision (%)	Recall (%)	F1 (%)	AUC: (%)
WIN-Ge	93.39	94.79	**97.74**	**96.24**	94.61
WIN-MalConv [10]	90.77	**98.88**	34.34	50.97	82.43
WIN-Ember [3]	**98.23**	97.47	89.72	93.43	**96.67**
WIN-MAGIC [16]	82.46	86.63	82.46	81.96	84.78
ANDROID-Ge	**99.85**	**99.74**	**99.74**	**99.74**	**99.57**
ANDROID-Adagio [4]	95.00	91.07	94.0	95.32	91.07

precision, recall, F1, and FPR, often used to describe a binary classification model's testing accuracy.

As Figure 3 and Table 3 reveal, we compared our results with other works on Windows-32 and Android platforms. The reason for this choice is as follows. For Windows-32 and Android samples, there are efficiently labeled malware samples. In addition, there are enough related works to compare the performance of our framework. However, to detect malware on Linux platforms, such as Linux-32, Linux-64, and Linux-ARM32, we cannot find any comparable works to support our achievement.

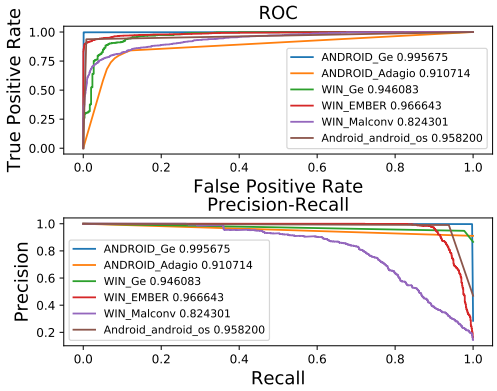
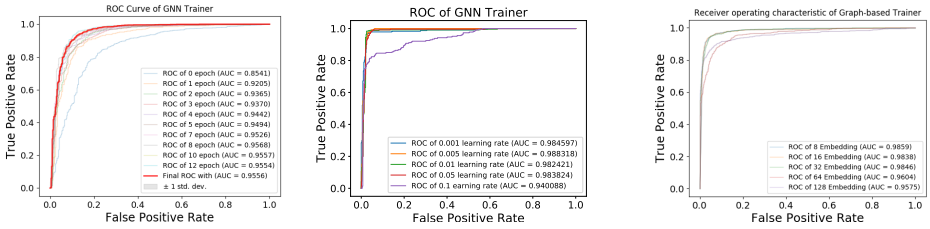


Fig. 3: ROC and precision-recall on Windows-x86 and Android

As Table 3 shows, on the Windows-x86 platform, for accuracy and precision, Ember [3] gets a better result than our graph embedding (GE) approach. However, for recall and F1-score, Ember does not work better. MAGIC gets the worst results among all of our experiments. When compared with MalConv [10], `HawkEye` outperforms its results. Similar to the Windows-x86 platform, we compare our performance on the Android platform with Adagio [4], which is a one-hot embedding-based malware detection with CFG. Our GE-based approach outperforms Adagio's results. In conclusion, our `HawkEye` outperforms other related works in nearly all metrics, especially on the recall and F1-function.

5.4 Hyper-Parameters Selection

Different Number of Epochs. To evaluate our module's convergence feature, we set a different number of epochs, between 10 and 25, in order to test the differences in the performance. The validation is processed every five epochs to



(a) Epoch (b) Learning Rate (c) Embedding Size

Fig. 4: ROC results with different iteration

select the best-weighted matrix. As Figure 4(a) depicts, the accuracy rises, and the loss decreases shapely in the first three epochs in an ordinary situation. After 12 epochs, the ROC value will be maintained at a certain level and only slightly float up and down. In essence, we get the best ROC results after 12 epochs. We then save the model parameters for future restoration in the test process. In conclusion, our model can have convergence quickly and achieve the best performance after about 15 epochs. In this experiment, the other hyper-parameters are fixed as the learning rates equal 12, the iteration size equals 2, and the embedding size is 32.

Different Learning Rates. The learning rate is a configurable hyper-parameter used in the training of neural networks. It is referred to as the step size that the weights are updated during training. The various value of learning rate affects the performance significantly. In this section, the influence of learning rates is studied. `HawkEye` evaluates the various learning rates with values: {0.001, 0.005, 0.01, 0.05, 0.1}. In this experiment, the other hyper-parameters are fixed as epoch equals 12, the iteration size equals 2, and the embedding size is 32. Figure 4(b) illustrates the outcomes. In detail, the AUCs of `HawkEye` achieve more than 98% for learning rate of {0.001, 0.005, 0.01, 0.05}, and when learning rate equals to 0.005, `HawkEye` gets the best of AUC with 98.83%. Only for the 0.1 learning rate, the AUC drops to 94%, which is due to the big step size of weight updating.

Different Embedding Sizes. From Figure 4(c), we can conclude that the embedding size in a specific range does not impact the performance significantly. The ROC curves are similar to each other, with an embedding size from 8 to 32. Considering the embedding size is positive relative to the training time and evaluation time, we decided to use 8 as embedding size for the trade-off between performance and efficiency. It is worth mentioning that if we select a bigger embedding size (e.g., 64) compared to a small input feature size (e.g., 32), the performance will decrease sharply because the features get dilute.

5.5 Detection on Obfuscated Samples

Table 4: Detection rate of obfuscated APK

	ClassEnc.	StrEnc.	Refl.	Triv.	Triv.-Str.	Triv.-Ref.-Str.	Triv.-Ref.-Str.-Class.
PRAGuard[5]	38.0	64.0	96	90.0	50.0	44.0	32.0
Drebin	99.12	98.99	86.58	98.32	98.99	99.32	96.98
Our framework	99.33	98.99	86.58	98.32	98.99	99.32	96.98

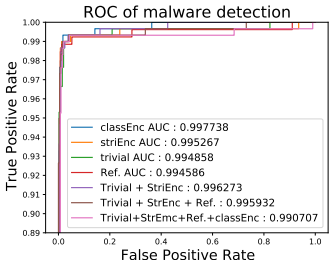


Fig. 5: ROC of obfuscated APK

Here, we present our experimental results for the detection of obfuscated samples. We only take obfuscated Android samples as our input and compare our work with PRAGuard [8]. The obfuscated methods include class encryption, string encryption, reflection, and various combinations. For other platform samples, our work can be extended easily. PRAGuard mentions the influence of obfuscated applications on Android malware detection. It presents seven types of obfuscation techniques and influenced performance. We evaluate our framework on the PRAGuard dataset. The ROC is illustrated in Figure 5. We compare the detection rate with PRAGuard in Table 4. From the extracted results, we identify that obfuscation does not influence our framework.

6 Conclusion

In this paper, we investigate a new methodology that detects malware on cross-platform architectures. We design and implement three separate tools: (*i*) a CFG generator, (*ii*) a feature embedding (includes opcode embedding and graph embedding) networks, and (*iii*) an MLP neural network malware detector. The combination of the above tools allowed us to build `HawkEye`, a combined detector. `HawkEye` solves the classification accuracy by training itself via diverse inner maximization methods, different embedding maps, and a specific type of CFG. The experiments validate that `HawkEye` can fast and accurately classify malware. Overall, it proves that the control flow graph and the graph neural networks can be successfully applied in malware detection.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No 883275 (HEIR) and No. 833115 (PREVISION).

References

1. VirusShare.com. https://virusshare.com/. Accessed: 2019-07-05.
2. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Androzoo: Collecting Millions of Android Apps for the Research Community. In: IEEE/ACM Working Conference on Mining Software Repositories (MSR) (2016)
3. Anderson, H.S., Roth, P.: EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. ArXiv e-prints (2018)
4. Gascon, H., Yamaguchi, F., Arp, D., Rieck, K.: Structural Detection of Android Malware Using Embedded Call Graphs. In: ACM Workshop on Artificial Intelligence and Security (2013)
5. Germain, J.M.: New Security Hole Puts Windows and Linux Users at Risk. `https://www.technewsworld.com/story/86778.html` (2020)
6. Goldberg, Y., Levy, O.: Word2vec Explained: Deriving Mikolov et al.'s Negative-Sampling Word-Embedding Method. arXiv preprint arXiv:1402.3722 (2014)
7. Kingma, D.P., Ba, J.: Adam: A Method for Stochastic Optimization. arXiv preprint arXiv:1412.6980 (2014)
8. Maiorca, D., Ariu, D., Corona, I., Aresu, M., Giacinto, G.: Stealth Attacks: An Extended Insight Into the Obfuscation Effects on Android Malware. Computers & Security **51**, 16–31 (2015)
9. Pektaş, A., Acarman, T.: Deep Learning for Effective Android Malware Detection Using API Call Graph Embeddings. Soft Computing **24**(2), 1027–1043 (2020)
10. Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C.K.: Malware Detection by Eating a Whole Exe. In: AAAI Workshop on Artificial Intelligence for Cyber Security (2018)
11. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In: Network & Distributed System Security Symposium (NDSS) (2015)
12. Stange, S.: Detecting Malware Across Operating Systems. Network Security **2015**(6), 11–14 (2015)
13. Total, V.: Virustotal-Free Online Virus, Malware and Url Scanner. Online: https://www. virustotal. com/en (2012)
14. Webster, G.D., Kolosnjaji, B., von Pentz, C., Kirsch, J., Hanif, Z.D., Zarras, A., Eckert, C.: Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage. In: International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) (2017)
15. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In: ACM SIGSAC Conference on Computer and Communications Security (CCS) (2017)
16. Yan, J., Yan, G., Jin, D.: Classifying Malware Represented as Control Flow Graphs Using Deep Graph Convolutional Neural Network. In: Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (2019)
17. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In: Advances in Neural Information Processing Systems (2019)